WILLEM LOUIS
VAN DER POEL

Den Haag, The Netherlands

Micro-programming and Trickology*)

With 3 Figures

Disposition

(Disposition cont'd)

*) *Acknowledgement.* Many of the tricks described in this contribution have been found by other people. The most prominent among these were DR. G. VAN DER MEY and MR. J. G. VAN LEYDEN who invented the more subtle tricks. It must be further understood that much of the material is a result of close teamwork.
The author wishes to express his warmest thanks to all concerned.

Summary. The growth of automatic programming languages for computers poses certain problems in logical design and machine code programming. Most classical computers are not very well equipped for composite actions such as searching a list, block transfer, sorting etc. There is a marked tendency in computers today to cope for these macro-actions by means of built-in features. The purpose of this article is to show some ways to build up these macro-instructions from a coding system where the programmer has immediate access to the micro-programming of the machine. Unfortunately, this subject cannot be treated without referring to a particular machine code. For this the ZEBRA code has been selected.

After a short introduction into the features of ZEBRA, a survey is given of all sorts of complicated macro-actions and how they can be expressed in this very flexible microcode. One of the key stones is the feature to repeat an instruction. In this way often a multiple use can be made of a single instruction. Another feature is the generation of pieces of coding in fast registers which are subsequently executed. These pieces were not written out in full beforehand. This technique is called "under-water programming". A considerable ingenuity is often required to devise the macro-instructions and this has given rise to the name "trickology" for the art of using this tricky programming.

Zusammenfassung. Die Entwicklung der automatischen Programmsprachen für Rechenautomaten erlegt der logischen Planung und der Festlegung des Maschinencodes gewisse Probleme auf. Die meisten klassischen Rechenautomaten sind in bezug auf zusammengesetzte Befehlsabläufe, wie beispielsweise Durchsuchen von Listen, Blocktransfer, Sortieren usw., nicht besonders gut ausgestattet. Heute besteht bei Rechenautomaten die deutliche Tendenz, solche Makroabläufe vor allem durch besondere, in die Maschine eingebaute Befehle zu bewältigen. In diesem Beitrag sollen einige Wege aufgezeigt werden, wie man solche Makrobefehle auch in einem Programmsystem aufbauen kann, in dem der Programmierer einen direkten Zugriff zum Mikroprogramm der Maschine hat. Unglücklicherweise kann man diesen Gegenstand jedoch nicht behandeln, ohne auf einen bestimmten Maschinencode zurückzugreifen. Es wird der Befehlscode des Rechenautomaten ZEBRA zugrunde gelegt.

Nach einer kurzen Einführung in die besonderen Merkmale von ZEBRA wird ein Überblick gegeben über alle möglichen Arten von komplizierten Makroabläufen und auf welche Weise man sie in diesem sehr flexiblen Mikrocode ausdrücken kann. Hierbei besteht einer der Hauptgedanken in der Möglichkeit zur Wiederholung eines Befehls. Auf diese Weise kann häufig ein einziger Befehl vielfach gebraucht werden. Ein anderes Merkmal besteht in der Erzeugung von Teilstücken des Programms in schnellen Registern, die nachher ausgeführt werden. Diese Teilstücke waren vorher nicht voll ausgeschrieben. Dieses Verfahren wird als „Unterwasserprogrammierung" bezeichnet. Da es jedoch häufig einer gewissen Erfindungskraft beim Zurechtlegen solcher Makroabläufe bedarf, so mag es gerechtfertigt sein, diese Programmierungsart als „Trickologie" zu bezeichnen.

Résumé. L'accroissement des langages de programmation automatique pour les grandes calculatrices électroniques pose certains problèmes quant à la réalisation logique et à la programmation en code-machine. La plupart des calculatrices classiques n'est pas très bien équipée pour les actions composées, telles que le traitement des listes, le transfert en bloc des mots, le tri des mots etc. A l'heure actuelle, il y a dans le domaine des calculatrices une forte tendance à assurer ces macro-actions au moyen de dispositifs incorporés dans la machine. Le but du présent article est d'indiquer les moyens pour

réaliser ces macro-instructions à partir d'un système de code dans lequel le programmeur a un accès direct à la micro-programmation de la machine. Malheureusement ce sujet ne peut être traité sans se baser sur un code-machine particulier. A cet effet, a été choisi le code de la machine ZEBRA.

Après une brève introduction expliquant les caractéristiques de la ZEBRA, l'auteur donne un aperçu de toutes sortes de macro-actions compliquées en précisant comment elles peuvent être exprimées dans ce micro-code extrêmement flexible. Un des points d'appui du système est la possibilité de répéter une instruction permettant de faire d'une seule instruction un usage multiple. Une autre caractéristique est la création des fragments de code dans des registres rapides, fragments qui sont exécutés ensuite et qui ne sont pas écrits en toutes lettres au préalable. Cette technique est appelée celle de la «programmation submergée». Souvent, la composition des macro-instructions demande une grande ingéniosité, ce qui a donné lieu à la création du mot «Trucologie», par lequel on désigne l'art de la programmation.

1. Introduction

There is a very marked tendency today to do away with all machine languages. At the highest level, problem oriented languages are the main goal. At most a machine oriented language can serve as intermediate step in describing a translator or compiler. Nevertheless somewhere some people must descend to the machine languages themselves to be able to make the programs for the transition between machine language and machine oriented but essentially machine-free languages. It shall not be the subject of this article to go into the problems of machine-free languages at any level as they have been dealt with in the contribution by F. L. BAUER and K. SAMELSON, in this volume pp. 227—268.

It is clear that the structure of automatic programming languages will have a repercussion on the logical structure of machines. Perhaps the most important facility of automatic programming languages is the automatic allocation of names in the store. As this allocation process is essentially a dynamic process (e. g. in recursive procedures [1, 2]) the store must be dynamically addressable, i. e. reference to locations must be possible relative to the last stored quantity. Such a store is called *stack, LIFO (last-in-first-out) memory, push-down store,* or *nesting store*[1]). Of course it is possible to build the stacking property into the hardware but it is also possible to programme the facility by keeping track of the position of the top of the stack (cf. KDF 9 of English Electric [3] and B 5000 of Burroughs [4]). This brings us to the desirability of index registers as they give just the possibility to add something to the address of an order to be executed, e. g. the top address of the stack. Going to a subroutine requires the storing of the top address of the stack for later reference when returning from that subroutine. A whole hierarchy of such top-addresses forms a list and it is clear that especially list searching for reference to variables of other levels of the hierarchy can be a frequent operation.

The necessity of having index registers is often interpreted by machine builders as a necessity to add the contents of these index registers to the address on the same instruction. But when analised in time sequence this always requires an extra add cycle before the execution of the instruction. Therefore it seems more

---

[1]) The same is designated "Keller" by F. L. BAUER and K. SAMELSON, cf. this volume, pp. 255—257 (Editor's remark).

logical to do this addition during the previous instructions. The end of the previous instruction fetches the next instruction and modifies it at the same time. As a by-product the advantage emerges that now with the same ease the modifiers can be modified by a whole string of such instructions (cf. 2 and 4 orders in EDSAC 2 [5], *NKm* orders in ZEBRA [6] and the structure of the Bendix G 20 Computer [20]). In this way the most general addresses can be composed as e. g. $(((a) + (b) + c) + d) + e$ where (n) denotes contents of n. The limitation of the number of index-registers to only a few and the special orders to handle them is a very severe drawback for automatic programming. The conclusion is: make every location of the main store also available as index register (cf. the *George*-Computer [7] and the Bendix G 20).

Of course the organization of stacks, lists, index registers etc. is greatly helped by having a big store of uniform properties. As soon as a two level store enters the picture, a transferring of blocks of information between main (high speed) store and background store becomes a problem. In this connection it is worth while to mention that it is possible to make the allocation for blocks to be stored by built in hardware and to keep track of the addresses in a label list. The allocation can be done in such a way that the first available free block is seized and reserved and is given a label which is independent of the real address that need not be known to the outside program any more. Especially when two independent programs are run on an interrupt basis side by side which must not disturb each other, this scheme can have great advantages (cf. *Atlas* Computer of Ferranti [18, 19]). Of course the same technique can be programmed as well.

Instead of having a stack, the individual locations can be organised in quite another way. When a variable must be stored the first free location can be looked up. To link the position of that location to the previous one in the stack or list a tag or label can be assiciated with it which gives reference to the previous address. This is called a threaded list. Manipulation of this list only requires manipulation with the tags, never with the information itself. Especially for system with variable length items (sorting problems, variable multi-length arithmetic, automatic allocation) this way of organization has many advantages notwithstanding the drawback of consuming extra storage space for the tags.

The reason for going in many details of machine structure in connection with automatic programming is that the present article wants to deal with some of the organization problems at the lowest machine level.

The structure of the micro-instructions is of course very important for building machines which are well suited for doing their work efficiently but that goal can be attained through a suitable structure of micro-instructions. The question is, how far must one go in decomposing the well known mathematical concepts of addition; multiplication and the organizational operation as transport, test, list searching etc. into more elementary fragments to be able to make one's own order code. The argument that micro-code is more difficult to be handled by the human programmer does not hold for automatic programming and the flexibility gained could well be a boon to speed.

It is the fate and doom of a machine code programmer that he can only describe his findings in a particular code for a particular machine. This has been done before, and most books on programming descend to the level of a particular machine (e. g. WILKES, WHEELER, GILL [8]). Nevertheless I shall go through the

cumbersome details of describing a particular machine to be able to come to the subject proper.

Some justification for doing this can perhaps be found in the reason that the structure of the machine in question (*Stantec ZEBRA*) is rather different from most classical machines so that the order code is composed of *functional bits* which each have a seperate and independent meaning. (Reference is made to [6] pp. 49—94.) We have tried to devise the logical design in such a way that the micro-programming permits the easy implementation of most macro-operations required. In fact it has appeared that a completely new technique of programming emerged (which we have called *under-water programming*) in which far more complicated macro-operations can be more easily dealt with than in most usual built-in machine codes. It also appeared that some very complicated actions involving timing problems in strobing a real time input or output device (such as a telephone dial or a teleprinter) can be solved in an incredible low number of instructions. Many of these complicated macro-instructions are connected with list searching, manipulation of treaded lists, block transfer, interpretation techniques so that the structure of micro-instructions has helped a great deal to make all sort of processes occuring in automatic programming particularly simple and speed.

In the design of ZEBRA not all ideal circumstances for making a good machine for automatic programming have been realised. For instance, the limited number of fast access index registers with special treatment and the optimum programmed store do not comply with the requirements given in the first part of this introduction.

A second reason for making the design as it stands was economic need to make the machine as simple as possible without sacrificing speed. Indeed much gain in speed has resulted from a more compact use of time and simultaneous action of the elementary particles of the operation; on the other hand input and output facilities were rather limited.

Hence I consider the purpose of this article to lie more in the line of giving limits how some sort of macro-operations can be dissected in general, but the only way to describe it is by taking two particular examples at hand. Other machines with a coding of similar scope have been built. To mention a few of them: The Z 22 Computer has a very similar functional bit coding (cf. the contribution by Zuse, in this volume, particularly p. 528); the *Mailüfterl* Computer of the Institut für Niederfrequenztechnik, Technische Hochschule Wien [9] is also based on a similar functional bit coding and has as special features operations for both binary and binary coded decimal. All have a one cycle basic operation.

In another line of thought the microprogramming in EDSAC 2 [10] has been applied. Here a class of micro-operations are provided in the machine but they are not accessible for the outside programmer. Instead they are used as constituents in time series for composing the more complicated instructions on a wired-in basis. All wiring is done in matrix form so that it is not too difficult to devise new orders and to build them in. In the same way the computers G 3 of the Max-Planck-Institut für Physik und Astrophysik, München [11, 12], and TR 4 of Telefunken [13] are logically designed.

Again a slightly different form of micro-coding is used in the TX—0 Computer built at the Lincoln Laboratory, Massachusetts Institute of Technology [14, 15].

The structure of the computer had to be made simple as it was only meant as test machine. Here there was adopted a decoded operation part of three orders with an address for fetching, storing and jumping. The fourth operation was addressless and the address bits were used to do all other operations (including input and output) in a functional bit way [15]. Both in EDSAC 2 and TX–0 the concept of having different groups of digits controling operations in time sequence was incorporated. In ZEBRA this is done only to the extent as comes naturally.

The concept of micro-programming and the practice of devising tricks to do the more complicated composite actions is so interwoven in ZEBRA that the volume of knowledge of these tricks has been given a special name: *trickology*. Without this knowledge of trickology and the standard programs based on it, ZEBRA would be a useless machine. In general this applies to all computer systems. The computer in itself will be of little value when given to a man only in possession of the manual of basic machine properties. The library of programs and the philosophy of program organization will make this computer into a useful tool. It is not unusual that this body of paper knowledge is more costly and more difficult to obtain than the machine itself. Especially when exchange of programming between machines takes place the program organization or languages used must be rigorously the same.

2. Description of ZEBRA Computer

ZEBRA is a binary magnetic drum calculator with a storage capacity of 8192 words of 33 bits. The drum is divided in 256 tracks of 32 words each. The words are consecutive on the drum. Words are transferred in series through the machine. Revolution time is 10 ms. The arithmetic unit comprises two accumulators A and B, A having 33 bits and an overflow position, B having 33 bits and a special carry trap for a carry-over. The control unit comprises a control register C which holds the next instruction, a control counter D both of 33 bits word length, and an execution register E in which the instruction to be executed is set up from C. A fast store comprising 12 short registers of 33 bits plus a few odd registers containing constants or performing special functions completes the picture.

Input is via 5 hole punched paper tape. Max. speed is 200 symbols/s.

Output is via 5 hole punched paper tape. Max. speed is 60 symbols/s.

Further output is via ordinary teleprinter (7 symbols/s).

The bits of the contents of a word are denoted by small letters derived from the name of the register or location with the left most digit starting in 0.

Thus

$$(B) = \overline{b_0\, b_1\, b_2 \ldots\ldots\ldots b_{32}}$$

It is a matter of interpretation to use the digits in a word to represent a fraction in the following way:

$$\overline{p_0\, p_1 \ldots\ldots\ldots\ldots p_{32}} = -\,p_0 + \sum_{j=1}^{32} p_j\, 2^{-j}, \quad \text{where } p_0 \text{ acts as sign digit.}$$

In the same way a number can be regarded as a signless integer:

$$\overline{p_0\, p_1 \ldots\ldots\ldots\ldots p_{32}} = \sum_{j=0}^{32} p_j\, 2^{32-j}$$

For the machine addition this makes no difference as all digits are treated in exactly the same way.

The structure of an instruction is as follows:

$\underbrace{c_0\, c_1\, c_2\, c_3\, c_4\, c_5\, c_6\, c_7\, c_8\, c_9\, c_{10}\, c_{11}\, c_{12}\, c_{13}\, c_{14}}_{\text{operation part consisting of 15 bits bearing the names}}\ \underbrace{c_{15}\, c_{16}\, c_{17}\, c_{18}\, c_{19}}_{\text{fast store address}}\ \underbrace{c_{20}\, c_{21}\, c_{22}\, c_{23}\, c_{21}\, c_{25}\, c_{26}\, c_{27}\, c_{28}\, c_{29}\, c_{30}\, c_{31}\, c_{32}}_{\text{13 bits for drums address}}$

$A\,K\,Q\,L\ R\ I\ B\,C\,D\,E\ V\ V_4\,V_2\,V_1\,W \qquad m \qquad n$

There are two addresses, one for selecting a fast register, the other for selecting a drum location. The 15 operation bits all have a separate meaning.

The *fast addresses* have the following properties and contents:

0 contains a fixed constant 0. It cannot be written into.

1 contains a fixed constant ε ($p_{32} = 1$). It cannot be written into.

2 is identical with accumulator A. It cannot be written into.

3 is identical with accumulator B. It cannot be written into.

4 . . . 15 } normal fast registers. They can be read off and written into.

16 . . . 21 } not provided in the machine.

22 contains (A). It cannot be written into. When selected, a_0 is transferred to the flip-flop for generating the signal for operating printer 2.

23 contains $p_0 = 1$; p_1 to $p_{32} = 0$. Constant. It cannot be written into.

24 contains the logical product of (A) and (B) taken bit by bit when read off. When written into it has no effect as such but causes the logical product of (5) and the contents of the selected drum address to be read from the drum instead of the original contents.

25 same as 22 except that it operates teleprinter 1.

26 contains contents of 5th hole of input tape. All zero's for 0, all ones for 1. When written into, a_0 is transferred to 5th hole of output punch.

27 same for 4th hole.

28 same for 3rd hole.

29 same for 2nd hole.

30 same for 1st hole.

31 when read off contents is 0 and input tape is stepped. When written into this has no effect except that it causes the symbol set up in the punch to be punched and the tape advanced.

2.1 Something about the Notation of Instructions

The instructions when written down on paper differ from the form in which they are present in the machine. This is purely a matter of input program and does not concern any of the principal points of the article. But as this notation has grown and is used we shall adhere to its conventions.

In general, functional bits $A, K, \ldots\ldots$ are written when present and are omitted when absent. The letter A serves as opening symbol and must stand in front. The letter X serves as opening symbol when A is absent; thus $\bar{A} = X$.

Other functional letters can be written in an arbitrary order. They serve to separate the addresses. The V-digits are treated separately and are always written at the end.

Of the two addresses none, one or both can be present. A drum address is written with at least 3 digits or must be $\geqq 32$. A fast address is smaller than 32. Non-significant zero's can be suppressed even if the address is zero.

The drum address is written before the fast address when the W-digit is absent, thus:

A200BCE5 Functional digits A, B, C and E present. Drum address = 200, fast address = 5.

X200R Functional bits R only. Drum address = 200, fast address = 0.

The fast address is written first when the W-digit is present. The W-digit is automatically inserted by the input program and is never written by the programmer. When the fast address is written first, another address p can be written. This will cause an inactive drum address $8192-2p$ to be input. Thus:

XK5 Functional digits K and W present, drum address 000, fast address 5.

X5K7 Functional digits K and W present, drum address 8178, fast address 5.

A point "$\cdot$" is written when no functional digits are available for separation of two addresses.

As an X jumping to the immediately following register is very frequent, an abbreviation will be introduced: $(p) = Xp + 1$ is denoted by N. In the same way NKK denotes $(p) = Ap + 1$.

2.2 The Function of the Operation Digits

The A-digit determines the character of the operation. If $c_0 = 0$ the operation is called X, and if $c_0 = 1$ the operation is called A. An X-operation has as main element the extraction of a new instruction, and the A-operation has as main element the execution of an instruction. However, the distinction between these kinds is not sharp.

The K-digit determines for which unit the fast registers are used, i. e. for the arithmetic unit or for the control. Together with the A-digit, the K-digit determines the way of coupling between the four parts: arithmetic unit, control, fast registers, and drum store. This will be clear from the functional interconnection scheme depicted in Fig. 1. (The term *fast registers* is now preferred to and replacing the terms *short registers* or *short store* which have frequently been used previously.)

The function of the Q-digit is the addition of $\pm\ \varepsilon$ to the B-accumulator, independent of the store.

The digits L and R effect the shifting of the contents of the double-length accumulator to the left or to the right, respectively.

The I-digit controls the additive or subtractive action of an instruction. This only applies to the accumulators, not to the control, and then only for the transfer to A or B.

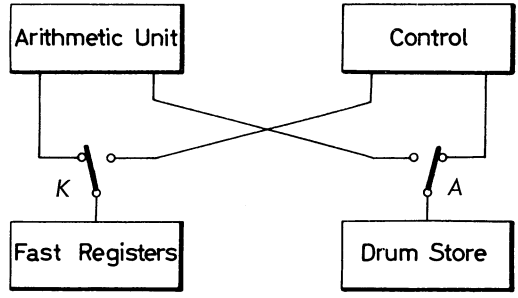


Fig. 1. Scheme of the functional interconnection between the four main computer units

The B-digit determines whether an operation refers to the A or to the B-accumulator. This only applies to adding, not to shifting.

The C-digit determines whether or not the accumulator engaged in the operation must be cleared.

The digits D and E determine whether reading or writing takes place from/to the drum and the fast registers, respectively.

The digits V, V_4, V_2, V_1 are called the test digits. With a testing operation the operation is either or not executed, dependent on the criterion described by the digits V, V_4, V_2, V_1. If the instruction is not executed, an instruction $A\,0$ is executed instead.

The digit W is related to the time selection on the drum. If $c_{14} = 0$, the execution of an operation is delayed till the selected storage location on the drum is present. If $c_{14} = 1$, the operation is executed immediately without the drum being waited for. The drum is completely disregarded. Zero is always read and nothing can be written on the drum.

The remainder of the digits forms the addresses: c_{15} to c_{19} constitute the fast address and c_{20} to c_{32} constitute the drum address; c_{20} to c_{27} serve the track selection and c_{28} to c_{32} serve the time selection within the selected track. For the sake of shortness the contents of the drum address will always be denoted by (n) and the contents of the selected fast address will always be written as (m). There is $(n) = 0$ if the W-digit is 1. If (n) is destined for A, this number is denoted by $(n)_A$. Then $(n)_B$ and $(n)_C$ are 0.

In the same way by $(m)_A$, $(m)_B$ and $(m)_C$ is denoted the contents of (m) as far as they are destined for A, B, or C. Both other entrances receive a 0.

2.3 The Action of the Instructions — The Functional Digits

The A-digit.

In the control the A-digit has the following action:

Operation X: $(C) + 2\varepsilon \rightarrow D$ $\quad (n)_C + (m)_C \rightarrow C$

Operation A: $(m)_C + (D) \rightarrow C$ $\quad (4) \rightarrow D$

Both operations do not differ in so far as the arithmetic unit is concerned. In any case adding or storing takes place according to:

$$(A) \pm \{(n)_A + (m)_A\} \rightarrow A \qquad (B) \pm \{(n)_B + (m)_B\} \rightarrow B$$

These standard operations can be modified by the other operation digits.

Register 4 has a special function and is related to the A-operations. All instructions are either X-instructions or A-instructions.

The K-digit.

If the K-digit is absent: the fast registers are used for the arithmetic unit.

If K is present: the fast registers are used for the control.

On a reading operation: $(m) \rightarrow C$

On a writing operation: $(D) \rightarrow m$

The *Q-digit.*

If the Q-digit is absent: normal.

If Q is present: ε is added to (B) (or is subtracted dependent on I). The ε is introduced in the carry entrance of the pre-adder of B as if it were a carry from "b_{33}". The adding of ε under control of Q is also taking place on a storing operation.

The L-digit.

If L is absent: normal.

If L is present: (A) and (B) are shifted one place to the left. If A and B are not cleared, the leftmost digit of B shifts to the rightmost digit of A, and B is completed on the right-hand side with a zero. The leftmost digit of A is lost. If A or B are cleared, zero is always transported from B to A. All other operations are performed in the normal way.

The R-digit.

If R is absent: normal.

If R is present: A and B are shifted one place to the right. When A and B are not cleared, the rightmost digit of A shifts to the leftmost digit of B. The rightmost digit of B is lost. A is supplemented on the left-hand side with a digit from a place which will be called a_{-1}. This place is situated on the left side of a_0, and completes the A-accumulator to an adder of 34 places instead of 33 places. For this extra place the following rules hold:

If A is cleared, a_{-1} is also cleared. All numbers to be added are first added together in what is called the pre-adder; then the resulting number is completed with a copy of its sign digit, after which the number of 34 digits is added into A with the main adder. This digit is serving effectively to store an overflow. The only method to recover this digit is to shift it to the right by an R-operation. The shifting to the right prevails over shifting to the left; thus a combination of R and L shifts to the right only.

For the sake of doing multiplications the following facility has been added to LR: if LR is present, add b_{32}. (15) to A instead of $(m)_A$.

The I-digit.

If the I-digit is absent: normal.

If the I-digit is present: take the complement of the numbers of drum and fast register, in so far as they are destined for the arithmetic unit. The contents of

15 on an *XD*- and an *LR*-operation and the ε on a *Q*-operation are also complemented when *I* is present. The *I*-digit does not refer to numbers to be stored, or to the control.

The *B*-digit.

If the *B*-digit is absent: the operation refers to *A*.

If the *B*-digit is present: the operation refers to *B*.

The addition normally takes place in *A* just as the storing normally takes place from *A*. However, if the *B*-digit is present, the addition takes place in *B* and the storing also takes place from *B*. The *B*-digit has no influence on the addition of (15) to *A* on an *LR*-operation. This addition always relates to *A*. The addition or subtraction of ε on a *Q*-digit also always takes place in *B*. The *B*-digit has no relation to the control.

The *C*-digit.

If the *C*-digit is absent: do not clear *A* and *B*.

If the *C*-digit is present: clear the accumulator as prescribed by the *B*-digit, before an addition or a shift takes place. The *C*-digit does not relate to the control.

The *D*-digit.

If the *D*-digit is absent, and if the execution is waiting for the drum: read the number from the selected drum storage location and perform on it an operation according to the other digits.

If the *D*-digit is present, and the execution is waiting for the drum: write in the selected drum storage location the number from *A* or *B* according to the following rules:

With an operation without *B*:	$c_8 = 0$:	(n) destined for *A*.
	$c_8 = 1$:	Transfer (A) to n.
With a *B*-operation:	$c_8 = 0$:	(n) destined for *B*.
	$c_8 = 1$:	Transfer (B) to n.

On the combination of *X* and *D* an extra addition takes place: Add (15) instead of $(m)_A$ or $(m)_B$ to *A* or *B* according to the *B*-digit.

The *E-digit*.

If the *E*-digit is absent: read the relevant fast register and use it for *A*, *B* or *C* according to the *K* and *B*-digit in the operations.

If the *E*-digit is present: read the number as determined by *K* and *B* in the selected register.

If *K* and *B* are both absent:	$c_9 = 0$:	(m) destined for *A*.
	$c_9 = 1$:	(A) m
If *K* is absent, *B* is present:	$c_9 = 0$:	(m) destined for *B*.
	$c_9 = 1$:	(B) m
If *K* is present:	$c_9 = 0$:	(m) destined for *C*.
	$c_9 = 1$:	(D) m

2.4 The Test Digits

If the *V*-digit is not present, the digits V_4, V_2 and V_1 together determine a number, having the value 0 to 7. These combinations are denoted by *U*0 to *U*7, added behind an instruction. If the instruction contains *Uk*, this operation is executed if

a testable switch k has been thrown. If not, the operation $A0$ is executed. The sense switches will be also denoted by $U1$ to $U7$. $U0$ is considered to be always thrown. An instruction with $c_{10}, c_{11}, c_{12}, c_{13} = 0$ will be executed in the normal way. $U7$ is materialized as a key having a normally closed contact; hence in contrast to the other six switches the test $U7$ succeeds when switch $U7$ is not thrown. This key serves as a start key.

If the V-digit is present, a V is added to the instruction.

$\overline{c_{10}, c_{11}, c_{12}, c_{13}} = 1000$ is denoted by V: See next paragraph.

$\overline{c_{10}, c_{11}, c_{12}, c_{13}} = 1001$ is denoted by $V1$: Execute the instruction if $a_0 = 1$, else execute A.

$\overline{c_{10}, c_{11}, c_{12}, c_{13}} = 1010$ is denoted by $V2$: Execute the instruction if $b_0 = 1$, else execute A.

$\overline{c_{10}, c_{11}, c_{12}, c_{13}} = 1011$ is denoted by $V3$: Execute the instruction if $(A) = 0$, else execute A.

$\overline{c_{10}, c_{11}, c_{12}, c_{13}} = 1100$ is denoted by $V4$: Execute the instruction if $b_{32} = 1$, else execute A.

The combinations $V5$, $V6$ and $V7$ are free for special applications. A test can be performed with the aid of these functional digits.

2.5 Double-length Facilities

To be able to perform double-length arithmetic very easily a device to take the carry-over from B to A is provided. As this carry-over is only produced on the last impulse time in a word, it is not possible to add it to A in the same cycle. This is always done in a later cycle (not necessarily the next).

The normal rule for double-length arithmetic is as follows: On every B- or Q-operation the carry-over is stored in an intermediate storage of one digit, named *carry-trap*. This carry is added to A on the first instruction having a $V0$, which can be written simply as V. The B-instruction and the related V-instruction must have an equal I-digit. The carry-trap retains the carry which has been put into it on the last B- or Q-operation. The carry from the carry-trap is introduced on the carry entrance of the pre-adder of A as if it were a carry from "a_{33}". On a left shifting instruction with V it is introduced one digit time late as if it were a carry from a_{32}. This implies that an instruction of the form $A200L5V$ can give wrong results, because the addition of (200) and (5) in the pre-adder can give rise already to a carry from a_{33} to a_{32} so that no other carry can be added at the same time. For a better understanding a short account will be given of the precise action of the carry-trap. A subtraction in B is performed by adding the inverse of the number together with introducing an extra complementary one on the carry entrance of the main adder of B as if it were a carry from "b_{33}". When a number is added, the resulting carry is just the opposite of what it would be, when the same number would be subtracted. For example, subtracting 0 gives a carry 1. In general this can be formulated as follows: The borrow produced on a subtraction is the opposite of the carry produced by adding the complement. However, on the next V-instruction the fact that a borrow has been stored in the carry-trap in opposite form must be taken into consideration by reversing its significance as an I-operation. The negative value of a borrow is automatically accounted for

by the introduction into the pre-adder. The result of this pre-addition (now including the borrow) is subtracted from A on a subtraction.

These seemingly awkward rules are necessary to be able to round-off on multiplication with a special trick, and to use the V as a sort of "Q-digit" for the A-accumulator.

Examples.

Round-off on multiplication:

$N \ldots IB23$	Last instruction of multiplication contains I. $B23$ subtracts $\frac{1}{2}$ from tail giving carry-over when tail $\geq \frac{1}{2}$.
$N \ldots V$	Round-off is added to head on next operation. B-instruction and corresponding V-instruction do not have the same I-digit!

The use of V as "Q-digit":

$N \ldots BI$	Subtract 0 from B thus making carry $= 1$.
$N \ldots V$	Add extra 1 to head from carry-trap; etc.

2.6 The Order of Preference

The functional digits of the operation are written in a certain order. This order is: $AKQLRIBCDEVV_4V_2V_1$. By reading it from the right to the left the order of preference of the functional digits is given. One can imagine the action to be thus that all functions take place subsequently. First from the test digits it is tested whether the operation is taking place or not. Then if storing has to take place, first storing is effected. Then if clearing has to take place, the clearing is performed. The relevant register is indicated by the B-digit. The position of the inversion digit is of no importance. The order of LR indicates that R has preference over L. When R and L are used together, only a shift to the right is effective. As last action the additions with Q and A take place. The position of the K-digit is unimportant.

3. The Repetition Instruction

The possibility to repeat an instruction has given this type of coding its greatest power. In this chapter we shall give a number of applications which encompass the most frequent types of serial operations. They comprise multiplication, division, normalisation (single and double length), block transport, zero-searching, searching in a non-ordered list for a specified part of a word, generating random numbers with FIBONACCI series, etc.

The basic idea of repeating an instruction stems from the fact that when a register is serving as the next instruction source, the drum address is not used as such when the W-bit is present. Nevertheless the address counter is augmented by 2 every cycle. This does not influence the fast address until the drum address overflows into the fast address. Thus when the drum address is equal to $8192-2p$, it will overflow after getting added p times 2 to it. Hence the notation $X5K7$ for: repeat instruction in 5 seven times.

Program:

100	*NKE6*
101	*X5Kp→*
→102	etc.

5	instruction to be repeated: *A* . .
6	return instruction

Action:

C	*D*	
X100		
X102KE6	*X102*	
X5Kp		*X102→6*
A . . .	*X5Kp-1*	$X5Kp = X5K + 8192 - 2p$
X5Kp-1		$X5Kp\text{-}1 = X5K + 8192 - 2p + 2$
A . . .	*X5Kp-2*	
.	.	
.	.	
.	.	
X5K1		
A . . .	*X6K*	*X5K1=X5K+8192−2* hence *X5K1+002=X6K*
X6K		
X102		Return to drum.

In this example we see the alternation between an *X*- and an *A*-instruction. The *X* is called the repeating instruction, the *A* is called the repeated instruction.

Of course both *X*- and *A*-instructions can do useful things. Observe that this count requires no extra apparatus but uses the normal address counter.

3.1 Multiplication

The most important application for a repetition instruction is multiplication. This can be done by the classical VON NEUMANN system. The *A* and *B* are forming a double-length accumulator, the multiplicand is placed in 15, the multiplier is placed in *B*, and *A* is cleared initially. At every cycle the last digit of *B* is tested and only when it is 1, the contents of 15 is added to *A*. Then *A* and *B* are both shifted to the right, thereby dropping the right hand digit of the multiplier and shifting a bit from the product from *A* to *B*. This digit does not change any more. The repetition of a multiplication runs as follows.

Program:

100	*NKE6LRC*	Place return instruction in 6. Clear *A* and add (15) conditionally. Shift right.
101	*X5K15LR*	Repeat *ALR* fifteen times (*X5K15LR* itself is done sixteen times).
102	*NLRI*	The last cycle is done negatively because of the sign convention.

5	*ALR*
6	return instruction

With respect to timing the 31 repeated and repeating instructions just fit into one revolution time of the drum.

Of course for an isolated multiplication the instruction *ALR* in 5 must be prepared. To give an insight how this can be done, an example shall be given of a complete open subroutine for multiplication of (*A*) and (*B*) without pre-supposing any contents of the registers.

Program:

100	*NE5*	Pre-instruction activity only starts after next instruction.
101	*NKKCE15*	Store multiplicand → 15. *ALR* → *A*.
102	*ALR*	Constant. After-action of *NE5* stores *ALR* → 5.
103	*NKE6LRC*	
104	*X5K15LR*	Multiplication as described above.
105	*NLRI*	After-action of *E5* destroys *ALR* but that does not matter.

3.2 Division

A division is more complicated but here also the classical VON NEUMANN scheme fits into the two available operations.

A step of a division can be subdivided in:

a) shift to the left, subtract divisor and note down a quotient digit 1, right or wrong;

b) test whether result has become negative. If so, subtraction must be undone. Add divisor again and remove quotient digit.

The repetition starts with the following initial contents: *A* and *B* contain the double-length dividend (positive), (15) = negative divisor.

Then the *program* runs as follows:

100	*NKE7*	Place return instruction in 7.
101	*X6K31QLD*	Repeating instruction: shift left, *XD* subtracts divisor, *Q* adds 1 to quotient. Repeated instruction: test sign of result. If negative undo action of *XDQ*. Only 63 word times fit into 2 revolutions.
102	*AQI15V1*	So last restoration is done separately.
— —	— — — — —	
103	unused	
104	etc.	

6	*AQI15V1*
7	return instruction

Of course in a practical application this core has to be supplemented by some preparatory programming for dealing with all combinations of signs. In practice a closed subroutine will be made for division once and for all. An example can be found in [6].

3.3 Normalisation

Normalisation is shifting a number *a* to the left until $(a) > \frac{1}{2}$ and counting the number of necessary steps. The example of single length normalisation (shifting in *A* and counting in *B*) has been given already in [6]. So we shall deal

with the more difficult case of double-length normalisation. *A* and *B* are supposed to be filled with a positive double-length number which has to be normalised with a repetition instruction. The difficulty is that both accumulators are occupied for shifting and cannot be used for counting. The solution can be found by using the repeating instruction itself as an indication for the number of steps.

Program:

100	*NR*23	Shift double-length number temporarily to the right and make sign-digit 1 for making next test succeed for the first time.
101	*NKE6V*1	Pre-instruction succeeds. *KE*6 has no meaning yet.
102	*NKE*6	Place return instruction *X*104*KE6V*1 in 6. Thus the repeated instruction is a test.
103	*XK6L*	Repeat and shift left *X*104*KE6V*1. As long as number to be normalised is still positive, test fails and repetition goes on. As soon as test succeeds, return to 104 and store the present repeating instruction in 6. When having shifted over n places, contents of 6 is in the end $XK6L+2n$. $2n$ can be separated from (6) later.

The technique of first preparing a few instructions in the registers which afterwards are executed many times and meanwhile alter themselves is called *under-water programming* because the active instructions do not appear as such in the object program. We shall see many examples of under-water programming later on where often the instructions executed far outnumber the instructions written down.

3.4 Block Transport from Drum to Registers

It is clear that for the preparation of under-water programs often a block of words has to be transferred to the registers. This can be done by a repetition instruction in the following way. As the instruction to be repeated must be modified during the repetition, the obvious place to put it is the *B*-accumulator. With the *XBD* combination the repeating instruction can modify the repeated instruction on every cycle.

Suppose we want to transfer (m), $(m+2)\ldots(m+8)$ to registers 6, 7, 8, 9, 10; then the *program* runs as follows:

100	*NC*5	(5) to *A*. Necessary for starting.
101	*NKKBC*	Take modifier *X*002·1 in *A*.
102	*X*002·1	Modifier for augmenting drum address with 2 and register address with 1.
103	*NKKBCE*15	Put modifier *X*002·1 → 15
104	*AmCE*5	and take instruction to be repeated in *B*.
105	*NKE*4	Store return instruction in 4.
106	*X*3*K*6*BD*	Repeat *AmCE*5 six times and modify it during repetition in *B*. It becomes successively *AmCE*5, *Am* + 2*CE*6, *Am* + 4*CE*7, Am + 6*CE*8, *Am* + 8*CE*9, *Am* + 10*CE*10 so that it has just transferred $(m) \rightarrow 6$, $(m+2) \rightarrow 7$ etc. Remark that there are no waiting times except for the first one.
107	etc.	

The same type of procedure can be applied for transport of numbers in the other direction.

3.5 Zero Searching

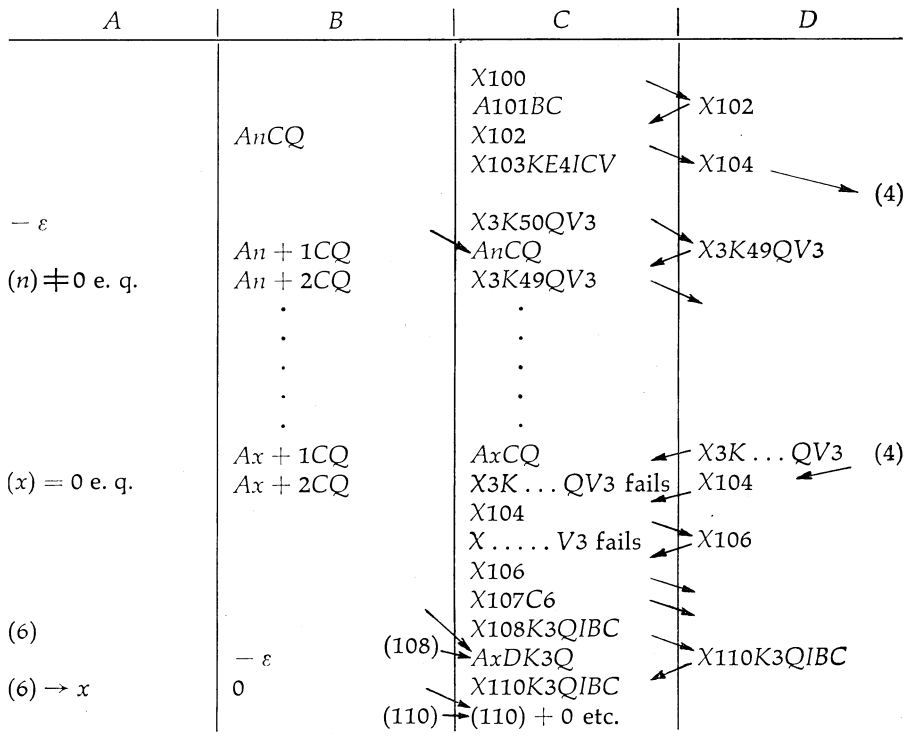
In list processing it often occurs that the first free location of a list must be looked up. This can again be done with a repetition instruction.

Suppose that the list is 50 places long and that these places are alternately spaced on n, $n+2$, etc. The *program* now runs as follows:

100	*NKKBC*	
101	*AnCQ*	Take *AnCQ* in *B* to be repeated.
102	*NKE4ICV*	Store return instruction *X*104 in 4. Fill *A* with a number $\neq 0$ to insure that process will start.
103	*X3K50QV3*	Repeat (*B*) = *AnCQ* 50 times. The *Q* on the repeating as well as on the repeated instruction step the address *n* by two every cycle so that all alternate locations are fetched in *A*. The test *V*3 looks for zero.
104	*X . . . V3*	When somewhere during the repetition the test fails, (4) comes into *C* and the program returns to 104 (see below). A *V*3 test on 104 can see whether actually a zero has been found and then goes on to 106. If nowhere a zero can be found, the repetition comes to a normal end on 4 after having repeated (3) for 50 times. But now the test on 104 succeeds because (*A*) $\neq 0$. The place x where an eventual zero has been found can be reconstructed from (*B*) = *Ax*+2*CQ*. For example when (6) must be stored in x there can follow:
105	not used	
106	*NC6*	Take (6) in *A*.
107	*NK3QIBC*	Take as next instruction (108) + (*B*) = *ADxK3Q*. Put $-\varepsilon$ in *B* afterwards.
108	*AD*000*K3Q*– *A*002*CQ*	Instruction executed is *ADxK3Q* storing (6) $\rightarrow x$.
109 110	*N* etc.	*K*3 modifies after-action not to *X*110*K3QIBC* but to *X*109*K3QIBC* and the *Q* on 108 clears *B* again so that the instruction on 109 is extracted unmodified by the *X*109*K3QIBC*.

In this example there are a few difficult actions to visualize. Therefore an action diagram shall be added. Each successive line gives an instruction.

Action:



3.6 Searching in a List

A more complicated action is searching an item in a non-ordered list. Suppose e. g. that a list contains in the usual even numbered places an identifier in the right-hand 15 digits. The left-hand 18 digits and the next word contain information to be extracted. So in this case a search must be made for a part of the word to be equal to a prescribed word. A mask defines the part of the word.

The mask is put in 5 and the prescribed word is put negatively in 15 at the start. $(5) = 2^{15} - 1$; $(15) = -a$.

100	*NKKBC*	
101	*AnQE24*	Put *AnQE24* in *B* as instruction to be repeated.
102	*NKE4ICV*	Return instruction to 4. $-\varepsilon$ to *A* to let the first repetition succeed.
103	*X3K50QCDV3*	The *XCD* combination puts $(15) = -a$ in *A*. Then *AnQE24* is repeated. *E24* fetches (n) masked by (5) and adds this to *A*. The next *X . . . . V3* tests for $-a + (n)_{\text{masked}} = 0$? The *Q* on *X . . . QV3* and *AnQE24* steps up the instruction *AnQE24* over 2.
104	*X . . . V3*	Test whether zero has been found or whether repetition has ended list.
	etc.	

Needless to say that this type of repetition is a keystone to all sorts of automatic programming language translation programs to search in identifier lists. Even in this computer with its waiting type store, the action is comparatively fast; only two word times per item. When the first item has been looked up, all others follow without further waiting times.

3.7 Generating Random Numbers by the Series of Fibonacci

A curious example of the application of a repetition instruction is the execution of a number of steps of the process $u_{n+1} = u_n + u_{n-1}$, the series of FIBONACCI. This is sometimes used as a generator of random numbers. The overflow of the addition is lost. The process described here, is a simplified form which would not be very good as random generator as the numbers are cyclically even, odd, odd. Suppose we want to progress p terms in the series. $(A) = u_{n-1}$; $(15) = u_n$.

Program:

100	*NBE5*	Pre-instruction.
101	*NKKBC*	Take *ALRCE15*.
102	*ALRCE15*	After-action *BE5* puts this in 5 as instruction to be repeated.
103	*NKE6BC*	Place return instruction in 6 and clear *B*.
104	*X5KpDQ*	Repeating instruction forms u_{n+1} in *A* by the *XD*
105	etc.	facility and puts ε in *B*.
		The repeated instruction *ALCRE15* then interchanges (A) and (15)!! For *CE15* places u_{n+1} in 15 and the *LR* facility puts $(15) = u_n$ at the same time in *A*. The ε in *B* made *LR* succeed. The right shift of *LR* cleared *B* again. The same process is repeated *p* times.

3.8 The Repetition of a Subroutine

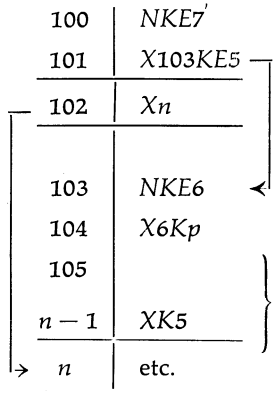
Although highly important for the most frequent processes a single repeated instruction cannot do more complicated repetitive processes. But fortunately it is possible to use as repeated instruction the call-in combination of a subroutine so that in fact the whole subroutine is repeated. This mode of working has as a drawback a loss of time because the program repeated is not any more in the fast registers but on the drum.

In principle this repetition works as follows.

Program:

100	*NKE5*	Store return instruction. The subroutine 200 reads like	*N* etc.
101	*X4Kn*	where e. g. (4) = *X200KE6*	
102		The first time *X4Kn*−1 is stored in 6 etc. The last time this instruction has become *X5K* and returns to 102.	*XK6* return

A practical and elegant way to implement this idea is the following *program:*



4. Fast Repetitions

Until so far all repetitions were of the type: two instructions, a repeating and a repeated instruction alternating each other. They work most effectively on alternate places of the drum. As soon as they have to work on every consecutive word they become very slow. There is another type of repetition which is termed a fast repetition. Here only one instruction is doing the work and is repeating itself.

4.1 Drum Clearing

As a first example a drum clearing routine is given. It will be programmed on the drum and consequently destroys itself during its action.

Program:

100	*NCE15*	Pre-instruction clears A and clears 15 as second action.
101	*A103BCKE4*	*X000K3QCD* → B. Place return instruction to 103 which will thence be cleared.
102	unused	A is cleared.
103	*X000K3QCD*	Start execution of instruction in B. Store 0 → 000; next instruction is *X001K3QCD*. *XCD* takes (15) = 0 → A. Q augments instruction in B and *K3* takes new instruction. Process stops when at last instruction has become *X000K4QCD*.

With another filling of 15 and making (103) = *X000K3QD* it is possible with the same trick to fill the store with any arithmetic progression.

4.2 Fast Sorting in Classes

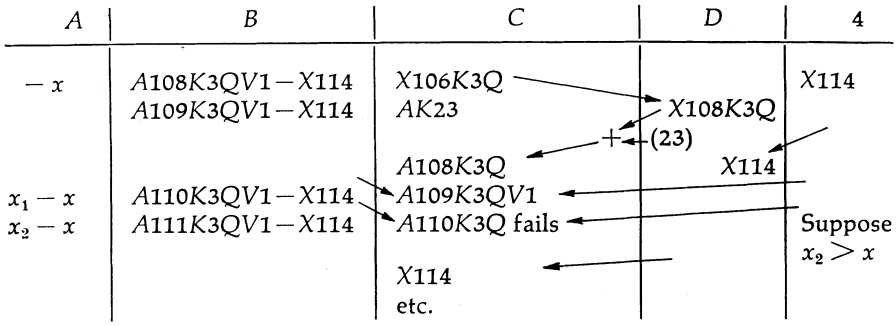
A frequent problem is the determination between which boundaries x_1, x_2, x_3 etc. ($x_1 < x_2 < x_3 < x_4$ etc.) a number x is lying. According to the class found another number can be extracted.

Program:

100	N	
101	$NKKBC$	Take return instruction in B.
102	$X114$	
103	$NKKBCE4$	$X114 \rightarrow 4$.
104	$A108K3QV1-X114$	Take instruction to be executed $-(4)$ in B.
105	$NK3Q$	Modify next instruction into $AK23$.
106	$AK23-A108K3QV1+X114$	$AK23$ makes second action of $X108K3Q$ into $A108K3Q$.
107	not used	Thus form $x_1 - x$. Next instruction comes from $(3) + (4) = A109K3QV1$.
108	x_1	Test $x_1 - x$. If negative: go on. If positive: return to 114.
109	$x_2 - x_1$	Form $x_2 - x$; etc.
110	$x_3 - x_2$	Form $x_3 - x$.
111	$x_4 - x_3$	
112	$A000 - x_4$	At last form something which is certainly positive.
113	not used	
114	etc.	

Of the critical part of the program an action diagram will be given which clarifies the action in the different registers.

Action:



4.3 Summing the Store

For checking purpose it can be very convenient to form a sum of the store from a pre-determined beginning to the end. When a single spare location is filled with the negative sum of all the others then this checks sum must result in 0, which can be easily tested.

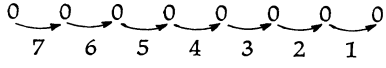
Suppose we want to sum the store from address x to the end. Then the *program* could run as follows:

100	*A*000*IC*	Take − (000) in *A*.
101		
102	*NKKBC*	
103	*X*54	$= \frac{1}{2} \times$ (return instruction).
104	*NKKBCE*4	Place $\frac{1}{2}$ (return instruction) in 4.
105	*AxK*3*Q* − *X*54	$= AxK3Q - (4) \rightarrow B$.
106	*NK*3*Q*	Execute *A*109*K*23*QIC* which subtracts (109) from *A* and 1 from *B*.
107	*A*109*K*23*QIC* − (105)	Next instruction becomes *A*109*K*3*Q* instead of *X*109*K*3*Q*. This adds again (109) to *A* and 1 to *B*. Next instruction becomes *AxK*3*Q* which adds (*x*) to A. Next instruction is *Ax*+1*K*3*Q* etc. Last instruction becomes *A*000*K*4*Q*. This adds (000) which had been subtracted right at the beginning. Furthermore the next instruction becomes (4) + (4) = *X*108 and the program returns to 108 with
108	etc.	$\sum_{k=x}^{8191} (k)$ in *A*.

The drawback of all three fast repetitions is that an end can only be forced by reaching the physical end of the store or by a test failing during the process. This limits the scope of the fast repetition. But for forming hash totals for checking purposes after having filled the store with a previously dumped contents it is very fast. In fact it is the fastest process which can ever be devised even in an immediate access store.

4.4 Displacing

A very neat application of a peculiar type of fast repetition appeared in a sorting routine. In that particular routine a set of items standing in alternate locations on the drum had to be moved up over two locations. Of course this can be done when starting at the last item.



But in this way it is a very slow process. When (*x*) has been picked up it can be dropped into $x + 2$ but then almost a revolution is lost in reaching $x - 2$.

The following trick solves the difficulty. Suppose the intermediate odd places can be used temporarily.

Now starting at the first item in *a*, (*a*) can be picked up and dropped in $a + 1$, in time $(a + 2)$ can be picked up and dropped in $a + 3$ etc. By repeating this procedure a second time the displacement has been performed.

In the program example the address *a* of the first item to be displaced will be supposed in *B* and a flag consisting of a zero will be considered to be present as last element. Only one of the two steps necessary will be described.

Program:

100	*NKKC3*	Form in *A:*
101	*XD001K3QV3−X109K3Q*	*XDa+1K3QV3−X109K3Q* as modifier.
102	*NKKB*	Form in *B.*
103	*AC000Q*	*AaCQ.*
104	*NKE4*	Store return instruction to 106 in 4.
105	*X107K3Q*	Form variable instruction *ACaQK2.*
106	etc.	Program returns here.
107	*X000K2*	Is executed as *ACaQK2.* Take (*a*) in *A.* Modify second action *X109K3Q* into *XDa+1K3QV3.* Test if (*A*) = 0.
		If instruction: store (*a*) → *a* + 1. Take next instruction from *B.* This has become *ACa* + 2*Q* in the meantime because of the *Q.* Repetition ends with a failing test when 0 is found and program returns via 4 to 106.

4.5 Fast Division

As a last application of fast repetitions a division will be treated. Often it is known in advance that the quotient will be a small integer only. In a conversion process from binary to decimal a binary number < 1000 can be divided by 100. The quotient never exceeds 9. The fastest way to program such a division is a stretched division consisting of repeated subtractions only.

Suppose for the example that the divisor has been put in 15. The dividend is but negatively in *A* and *B* is cleared for the quotient.

Program:

100	*NBE5*	Pre-instruction.
101	*NKKBC*	Take *XK5QDV1* in *B.*
102	*XK5QDV1*	and put it in 5 by second action of *BE5.*
103	*NKE4BC*	Place return instruction in 4 and clear *B.*
104	*XK5QD*	Start division. *Q* notes down units of the quotient. The *XD* facility adds the divisor to the negative dividend. From now on (5) = *XK5QDV1* is continuously repeated testing the dividend. As long as subtraction succeeds quotient bits are registered until *V1* fails. Then the second action also fails and (4) comes into *C* returning to 105 with the remainder in *A* (positive because the subtraction has been performed one step to far) and the quotient + 1 in *B.*
105	*NI15Q*	This instruction restores the correct remainder and quotient.

5. Miscellaneous Tricks

A lot of useful tricks do not fall under the heading repetition instruction. But all tricks treated below fulfil the requirement that they are minimum programs in respect of time as well as of number of instructions or both. Some of them indicate ways of doing things which are not possible in another way, e. g. the extraction of four consecutive words from the store in four consecutive word times. In this respect it is rather irrelevant that the store of the machine in question is a waiting type store although many of the tricks have been produced under the necessity of doing it optimally or alternatively wasting prohibitive waiting times. The result however can be applied to other machines with non-waiting types of store. The gain in speed will then not be of the order of 32 but of the order of 2 to 4.

5.1 Transferring a Number without Making Use of the Accumulators

By accident the following trick was discovered in a situation where the accumulators could not be destroyed and all registers except a particular one (say m) were occupied. In that situation (4) had to be transferred to m. The following few instructions do this transportation via the D register, instead of via A or B.

Instructions:

NKEm	Pre-instruction. Store "return-instruction" from D to m.
A....	Any A-instruction e. g. *AE*4. On every A-instruction (4) $\rightarrow D$.
etc.	Second action of *KEm* stores $(D) \rightarrow m$.

The pictured case of passing over a number *behind your back* shows once more how arithmetic unit and control unit must be regarded as one integral organising unit as has been shown before in repetition instructions where B often served as an extension of the control. Especially in the next few tricks the boundaries between arithmetic and control become very vague. In a certain way this is true for every machine as soon as it starts calculation on instructions. However in many machines calculation with instructions only means calculation with addresses. In all examples shown until here it is very clear that the aspect of altering the operation part as well is at least as important. This is the main reason that all registers contain full words, even the D-register and the short registers when used as modifier with *NKm*.

5.2 Extraction of Three and Four Consecutive Words

The problem of extraction from the main store two or more words (or numbers) from consecutive lines is more a problem of timing. Of course a program can always be written for it but then more than one word time is lost for extraction of one word. This is perhaps not so serious in an immediate access store but in a waiting type store like a drum this wastes a whole revolution. In any case doing it in one word time per word is quicker. The difficulty in getting access to consecutive words with a two address instruction is amounting to two main points:

a) All instructions must be dependent on the same initial and variable address. For fixed addresses there is not the problem of index-modifying the instructions.

b) The first address is used for the extraction, the register address can serve as next instruction source (via modification) but then the extracted number cannot be stored away. Or the accumulator is freed from the extracted numbers by *Em* but then the register address is not available any more for fetching a new instruction.

For two numbers it is not difficult to devise a solution as there the difficulty mentioned under point b) is not yet present; both numbers can be left in the accumulators. Therefore our attention will only be directed to the extraction of three and four numbers. Both examples given are the result of laborious trying. Although thought to be possible it was not known for a long time how to do the four number extraction until VAN LEYDEN found the solution. A proof can be given that no five consecutive numbers can be extracted.

The three word extraction reads as follows: $(B) = n$; then $(n) \rightarrow 4$, $(n+1) \rightarrow B$ and $(n+2) \rightarrow A$ at the end of the program.

Program:

100	*NQIBE6*
101	*NKKIB*
102	*NKKBDE61V7*
103	*NQIE1*
104	*NKQIE4*
105	*NKIB3*
106	*NKI*
107	etc.

An explanation of the above program is given in detail as follows.

Action:

A	*B*	*C*	*D*	
	n	*X101QIBE6*		Only *QI* is important, *BE6* is pre-instruction.
	$n-1$	*A102IB*	*X103QIBE6*	Subtract constant from (102) to (*B*).
	$n-1-A103BDE61V7$			Accidentally this constant could be written as an *NKK*.
		X103QIBE6		$n-A104BDE61V7 \rightarrow 6$.
	$n-A105BDE61V7$	*X104QIE1*		Only *QI* is important. The *E1* is harmless and required on the return instruction in 4 for modification.
	$n-A106BDE61V7$	*X105QIKE4*	*X106QIE1*	Store return instruction.
			4	*X106QIE1* → 4. *QI* goes on subtracting 1 from *B*.

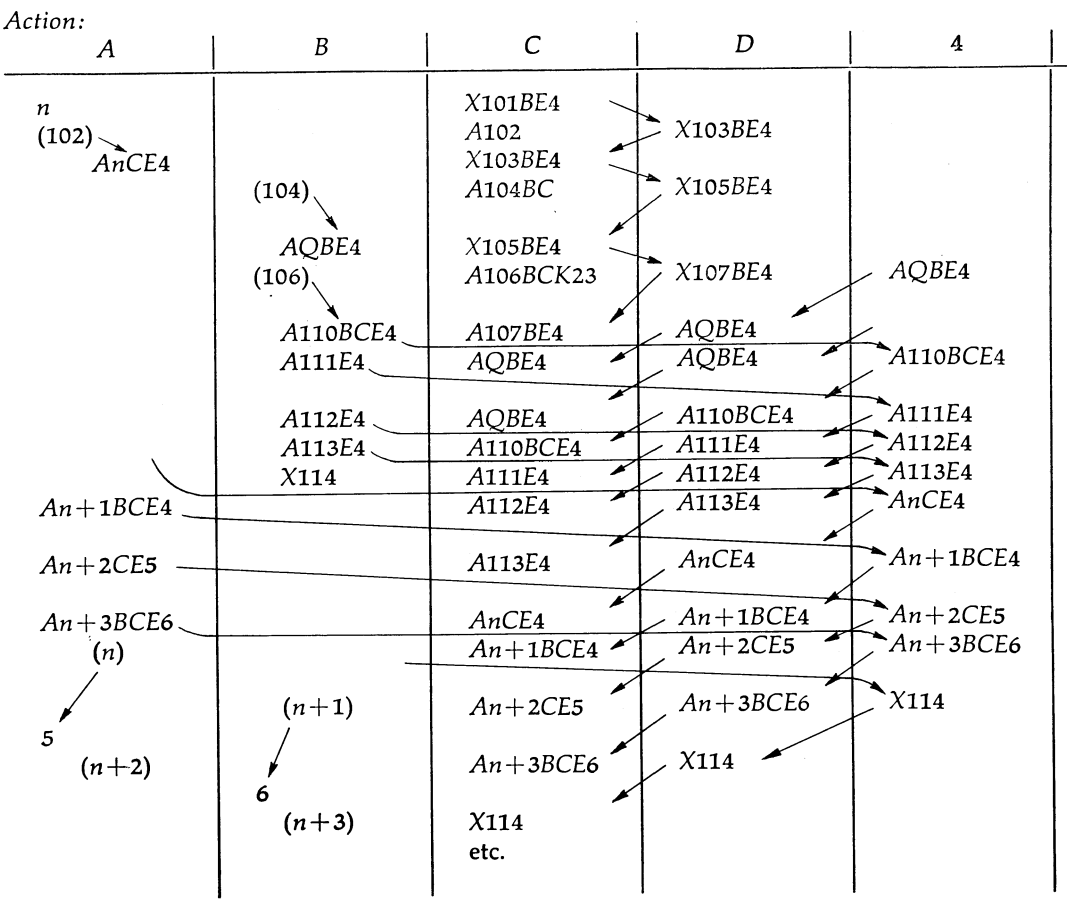
(cont'd)

A	*B*	*C*	*D*	
	$n-A107BDEG1V7$	$X106IBK3$		*IB* is harmless but required for modification. Take as next instruction $(106)+(B)=X107KI+n-A107BDE61V7=AnCK3$.
		$+(106)$		
		$AnCK3$	$X108IBK3$	Extract first number from *n*. Modify $X108IBK3$ with (*B*).
(n)				
			(4)	$X108IBK3+n-A107BDE61V7=An+1KBC6$.
		$An+1BCK6$	$X106QIE1$	Extract second number from $n+1$. Next instruction is obtained as: $(4)+(6)=X106QIE1+n-A104BDE61V7=An+2QCE4$.
	$(n+1)$			
		$+(6)$		
		$An+2QCE4$		Store first number in 4. Extract third number in *A*. *Q* serves for counteracting *QI* on return instruction.
$\rightarrow 4$				
$(n+2)$	$(n+1)+1$	$X106QIE1$		Restore (*B*). *E*1 is harmless.
	$(n+1)$	$X107KI$		*KI* is harmless.
		etc.		

Observe that the program as written down in the form of *N* and *NKK* instructions only, is impervious against displacement, i. e. it would work equally well in any place of the drum when input with the *N*, *NKK* notation. Of course for all programming on paper a symbolic or relative addressing system is used, but as this is a question of the construction of an appropriate input program, it does not belong to the realm of machine-bound micro-programming and hence will be explicitly omitted from this article.

The four word extraction is based on the idea that the only channel from where a string of four consecutive instructions can come is from *A* or *B* by storing $\rightarrow 4 \rightarrow D \rightarrow C$. First by successive *A* . . . *E*4 orders a chain of appropriate orders is built up in *B*, 4, *D*, *C*. The best way to explain is the action diagram (cf. p. 295).

The program (cf. p. 296) is as follows: $(A) = n$ at the beginning, the program returns with (n) in 5, $(n+1)$ in 6, $(n+2)$ in *A*, $(n+3)$ in *B*.


Action:
A
B
C
D
4
n
(102)
AnCE4
(104)
AQBE4
(106)
A110BCE4
A111E4
A112E4
A113E4
X114
An+1BCE4
An+2CE5
An+3BCE6
(n)
5
(n+2)
(n+1)
6
(n+3)
X101BE4
A102
X103BE4
A104BC
X105BE4
A106BCK23
A107BE4
AQBE4
AQBE4
A110BCE4
A111E4
A112E4
A113E4
AnCE4
An+1BCE4
An+2CE5
An+3BCE6
X114
etc.
X103BE4
X105BE4
X107BE4
AQBE4
AQBE4
A110BCE4
A111E4
A112E4
A113E4
AnCE4
An+1BCE4
An+2CE5
An+3BCE6
X114
AQBE4
A110BCE4
A111E4
A112E4
A113E4
AnCE4
An+1BCE4
An+2CE5
An+3BCE6
X114
Pre-instruction.
Add A000CE4 to n.
Take chain filling instruction in B.
Store AQBE4 in 4.
Take constant B and modify X107BE4 into A107BE4.
Change (B), store to 4.
The chain has started: A → 4 → D → C.
Q augments B.
Return instruction to B.
AnCE4+X001B=An+1BCE4.
An+1BCE4−X000B+X001·1=An+2CE5.
An+2CE5+X001B1=An+3BCE6.
At last (n) → A.
(n+1) → B
return instruction → 4
(n) → 5.
(n+1) → 6.

Program:

100	*NBE4*
101	*NKK*
102	*A000CE4*
103	*NKKBC*
104	*AQBE4*
105	*NKKBCK23*
106	*A110BCE4*
107	$+1-X000BC$
108	} free places
109	
110	*X114*
111	*X001B*
112	$X001 \cdot 1 - X000B$
113	*X001B1*
114	etc.

Perhaps the idea of having a series of instructions available, which are put up beforehand, can be useful generally for inner cycles of procedures where the utmost of speed is required. In this machine the setting up could be done only in a clumsy way, in most machines it cannot be done at all, but a control could be built with a stack of fast-access registers pre-filled with the required instructions and executed without an extra instruction fetch cycle.

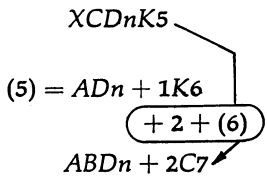
5.3 Storing Four Numbers in Consecutive Locations

The storing of four numbers is much easier. This is caused by the irregular action of the *D*-digit which always stores from the accumulators. Only a sketch of the program shall be given.

The numbers to be stored in n, $n+1$, $n+2$ and $n+3$ shall be denoted by a, b, c and d. Then at the outset the following contents of the registers must be set up:

$A=2$	a
$B=3$	c
4	$XBDn+3K8$
5	$ADn+1K6$
6	$ABD000C - XCD000K5$
7	d
8	return instruction to drum
15	b

The process is started by:



Store $a \rightarrow n$ next instruction from 5.
XCD takes (15) $\rightarrow A$.
Store $b \rightarrow n + 1$. *K6* modifies *XCNnK5* to $ABDn + 2C$.
Store $c \rightarrow n + 2$; $n \rightarrow B$. Two successive *A*-instructions hence next instruction from 4.

(4) = *XBDn + 3K8* Store $d \rightarrow n + 3$. Next instruction from 8. Return.

5.4 Modifying a Modifier during a Repetition

Once the problem arose of storing two numbers in n and $n + 2$ and extracting three numbers from $n + 6$, $n + 8$ and $n + 10$[2]). Of course the alternate spacing of locations lends itself better for treatment with a normal repetition instruction. There is no time to do it with index-modified drum instructions as n is variable. The only way to make it quick is by *under-water programming.*

Action:

A	*B*	*C*	
a	*AnCD11V*	*XK3BD6*	Modify (*B*) with (15).
	An+2CD13	*AnCD11V*	Store $a \rightarrow n$. Fetch (11) + 1 = *b*.
b		*XK3BD5*	Modify *An+2CD13+X002.2–X000V = An+4CE15*.
	An+4CE15V	*An+2CD13*	Store $b \rightarrow n+2$. Fetch 2nd modifier in *A*.
X002-X000·6		*XK3BD4*	
	An+6CE17	*An+4CE15*	Store new modifier in 15. (15) = *X002 − X000 · 6*. Extraction of $(n+4)$ is not used.
$(n+4)+1$		*XK3BD3*	Go on modifying *An+6CE17+X002−X000·6 = An+8CE11*.
	An+8CE11	*An+6CE17*	$(n+6) \rightarrow A$. *E17* is harmless.
$(n+6)$		*XK3BD2*	
	An+10CE5	*An+8CE11*	$(n+6) \rightarrow 11$; $(n+8) \rightarrow A$.
$(n+8)$		*XK3BD1*	Last modification is not important.
		An+10CE5	$(n+8) \rightarrow 5$; $(n+10) \rightarrow A$
		Xk4BD	(4)
		return instruction	Return to drum routine.

[2]) The problem came from a program for solution of simultaneous differential equations with the method of RUNGE-KUTTA-GILL [16] where *y* and *q* of the previous equation must be stored and *y*, *q*, *k* of the next equation must be fetched.

The preparation shall not be given. At the outset we suppose:

$(A) =$	a, first number to be stored
$(11) =$	b—1, b is second number to be stored
$(15) =$	$X002{\cdot}2 - X000V$ 1st modifier
$(13) =$	$X002 - X000{\cdot}6$ 2nd modifier
$(B) =$	$AnCD11V$

The *program* starts with

$NKE4$	Place return instruction in 4.
$XK3BD6$	Repeat (B) six times.

The explanation follows from the action diagram (cf. p. 297).

5.5 Multiplication with Small Factors

For multiplication with small constant factors often shorter programs can be devised than would appear possible at first sight. Only a few examples will be given.

Multiplication of (B) with 10.

Program:

$NLC3$	Form 2-fold of B but take 1-fold in A.
$AL2$	Form 4-fold in B and add 1-fold from A giving 5-fold.
	After-action $LC3$ forms 10-fold.

Multiplication with 32. It is obvious how it can be done with five shifts. It can, however, be done in four instructions. Suppose $(B) = b$.

Program:

$NLC3$	Form $2b$ in B but take b in A.
$NLB3$	Form $2b$ in A and $4b+2b=6b$ in B.
$NLB3$	Again double A and triple B giving $(A) = 4b$; $(B) = 18b$.
$NLIB2$	Form in B $36b-4b=32b$.

Multiplication with 100. It is obvious how to do it in six word times (twice the program for forming 10-fold). It can be done in five instructions.

Program:

$NLC3$	$(A) = b$	$(B) = 2b$
$NLB3$	$(A) = 2b$	$(B) = 6b$
$NLB3$	$(A) = 4b$	$(B) = 18b$
$NLB3$	$(A) = 8b$	$(B) = 54b$
$NLIB2$		$(B) = 108b - 8b = 100b$

Dependent on the required constant remarkable short solutions can be found. Until so far no systematic tabulation of the shortest programs for factors has been undertaken but for all factors under 100 the solution is known by hand methods.

6. Miniaturization

Until so far problems of micro-programming have been treated, doing compound actions with repetition instructions. A second field of applications is miniaturization in the sense of compressing programs in a space as small as possible. Especially one kind deserves attention, viz. the so-called *tape programs*. A tape program is a program which does not use anything on the drum (except track zero, see below) but reads its instructions during action. The registers may be used freely. Therefore, these programs could also be named *register programs*. As the number of registers is very limited much ingenuity has gone into these tape programs. To be read-in they make use of an input program for input in binary form. This binary input program is almost permanently contained on the drum in track zero and is kept locked (i. e. track zero can be read but not be written into unless specifically unlocked by a carefully guarded switch). Many of the register programs borrow instructions from track zero. The advantage of tape programs lies in the fact that they can be run and used without being anything on the drum and without destroying anything on the drum. So they are inherently suitable for service programs, testing programs etc. In the sequel a few examples will be treated, namely:

1) a tape copying program to copy tape from input reader to output punch,

2) a program to input decimal number by telephone dial,

3) a program for punching out the contents of the store (from a predetermined address to another address) in binary form to be read in subsequently,

4) a program for reading tape and printing the symbols immediately on the teleprinter. This serves for making tapes print their own title on the output printer without even the standard printing routines being present in the machine.

For a good understanding of the register programs it is not strictly necessary but very desirable to know how they can be read into the machine by track zero. Since a few instructions are borrowed from the binary input program, the description has been added in an appendix (cf. Section 7, p. 308 ff.).

Another kind of miniaturization was required in finding a pre-input program; i. e. a program consisting of as few instructions as possible which enables the machine to read in a more complete input program. Some machines have a built-in facility to read words into the store starting with an empty machine; this machine has not. Therefore the pre-input program must be put into the machine manually which is a rather tedious procedure. Fortunately this need never be done under normal operating conditions as track zero cannot normally be destroyed. Only in case of a breakdown of track zero one must revert to the pre-input program. In fact the pre-input program does not build up track zero in one step but in three steps. This *bootstrapping* technique is well known.

6.1 The Pre-input Program

After an intensive search for miniaturization at last a program of only two instructions could be devised. It is rather unsatisfactory that in general no theory exists which can prove that a particular solution is the minimum solution although for this case an ad hoc proof can be given that two instructions form the minimum pre-input program.

Instructions:

000	*X8190IB30*	Read 1st hole from tape into *B*.
8190	*AD8191LK29*	Store word from *A* into 8191. Shift *A* and *B* left. If 2nd hole = 0 after-action of *X8190IB30* becomes *X000IB31*: step tape and go again to 000. If 2nd hole = 1 after-action becomes *X8191IB30*: go to instruction in 8191.

Reading from the first hole by register 30 causes a string of zeros to be read in case of hole zero and a string of ones (= − 1) in case of hole one. Hence *IB30* reads 0 or 1 into the right-hand side of *B*. The *L* on 8190 shifts *A* and *B* to the left, the after-action of *IB30* becomes *IB31* because 8190 + 2 overflows into the register address. This does the stepping. In this way *A* and *B* can arbitrarily be filled. Every cycle (*A*) is stored in 8191 overwriting the previous number in 8191. At last a suitable storing instruction (e. g. *XnBD31*) which stores the word built up in *B* in location *n*. The end mark of a word is given by the presence of a second hole whereupon the *K29* modifies *X8192IB30* by −1 into *X8191IB30*. One can see that *B* always ends in 00 or in 11 which puts a severe limitation to the words which can be input. For a detailed description of the coding on the tape we must refer to the programming manual of the machine [17].

6.2 Tape Copying Program

The requirement of this program was that it could copy tape continuously as well as step by step (this for correction purposes). It is admitted that this copying of tapes with a high speed computer is an abuse of the machine. Parts of the program however have served for copying titles, etc.

Program:

4	*X6K1*
5	*XK14BCU7*
6	*X5K1V4*
7	*XK15RIC*
8	*ALR*
9	*XK10L*
10	*X011LE26*
11	*XK11QBCU7*
12	*X013LE27*
13	*XK11QBCU1*
14	*X001C7*
15	*X8K5*

The progam is started in 11 and stops. Copying is started when "start" is pressed and stopped when *U1* is pressed. When *U1* is locked in the "1" position, starting with *U7* only does a single step. The program is explained in the following action table giving the successive contents of *C*.

Action:

11 =	*XK11QBCU7*	Program makes a loop stop until *U7* is pressed. Then also the after-action fails and control goes to 4. 1 has been put into *B*.
4 =	*X6K1*	Go to 6 and execute once (unless a jump).
6	*X5K1V4*	*V4* succeeds. Go to 5 and execute once.
5	*XK14BCU7*	As long as *U7* is pressed *U7* fails and control comes to 6 again with $(B) = 1$. As soon as *U7* is released, go to 14 and clear *B*.
14	*X001C7*	Take *XK15RIC* in *A* as constant. Of this constant only the *I*-bit is of importance. $(A) = 000011$ etc.
001	*X003LIB26*	Read 5th hole of symbol to be copied in *B*. $(A) = 00011\ldots$
003	*NLIB27*	Read 4th hole of symbol to be copied in *B*. $(A) = 0011\ldots$
004	*NLIB28*	Read 3rd hole of symbol to be copied in *B*. $(A) = 011\ldots$
005	*NLIB29*	Read 2nd hole of symbol to be copied in *B*. $(A) = 11\ldots$
006	*NLIB30*	Read 1st hole. Symbol *S* is now complete. $(A) = 1\ldots$
007	*NL31V1*	Test succeeds. $2S \rightarrow B$, step tape $(A) = 0\ldots$
008	*X001RV1* *L31V1*	*V1* fails. After-action also fails. Hence go to 4.
4	*X6K1*	
6	*X5K1V4*	*V4* now fails because $2S$ is even. Hence go to 7.
7	*XK15RIC*	Jump to 15; clear *A*, $S \rightarrow B$.
15	*X8K5*	Start multiplication of 5 steps.
8	*ALR*	The multiplication constant is (15) = *X8K5* itself!!
	X8K4	Only the *K*-bit is important. The multiplication
	etc.	brings symbol *S* from the right most places of *B* to one place but left in *A*. $(A) = 0xxxxx$. Repetition ends in 9.
9	*XK10L*	Shift symbol in *A* left.
10	*X011LE26*	Put 5th hole in punch buffer. Shift bit off.
011	*X12K5*	Go to 12. This instruction is borrowed from track zero, otherwise instruction in 10 could not make use of a register address for setting up the punch.

(cont'd)

12	*X013LE27*	Set up 5th hole. Timing is just right.
013	*NQLE28*	Set up 3rd hole. *Q* is of no importance.
014	*NLE29*	Set up 2nd hole.
015	*NLE30*	Set up 1st hole.
016	*NE31*	Punch symbol.
017	*X13K1*	Go back to register 13.
13	*XK11QBCU1*	Test *U1*. If $U1 = 0$ go on to 14 and copy another symbol. If $U1 = 1$ go to stop cycle on 11 with $(B) = 1$ again.

6.3 Decimal Input by the Telephone Dial

More difficult, especially in timing, is the dial input program. The telephone dial is coupled in series with *U7*. In quiescent state it means that *U7* normally succeeds. When dialling the dial interrupts *U7* for 60 ms for every impulse, the time between the impulse being 40 ms (both with 10 % tolerance). A zero is dialled as 10 impulses. The convention has been made that when a next decimal digit follows within 1.5 s it must be accepted; 1.5 s after the last decimal digit the program goes on with the dialled number converted to binary in *A*.

The filling of the registers is as follows.

Program:

4	*XK14QCD*
5	*A*
6	*X011K3*
7	*AD000*
8	*X5K96U7*
9	*AE15*
10	return instruction
11	*XK11BCU7*
12	*X017.3*
13	*X9K2400BCU7*
14	*X7K8D*
15	0 initially. Later: partially converted number.

The explanation of the program is given in an action table (cf. pp. 303–304).

The principle of strobing the timing of the dial is done according to the timing diagram shown in Fig. 2.

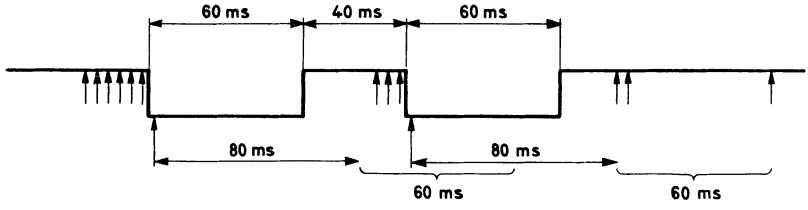


Fig. 2. Principle of strobing the timing of the dial

Strobing is done continuously until the start of the first pulse is seen. From that moment onward the program goes into a wait cycle for 80 ms (nothing interesting in the meantime). Then the program looks again for the start of the next pulse during 60 ms. When it does not arrive within 60 ms, the decimal digit is finished and the digit can be added to ten times the previous result. Then it enters into a wait cycle for 1500 ms. If a next pulse arrives within these 1500 ms, the program starts building up the next decimal digit; if no pulse arrives, the program must return with the complete dialled number in *A*.

Action:

11	*XK11BCU7*	Loop stop on 11. Clear *B*. Test *U7*. As soon as the beginning of the first pulse comes in, *U7* fails. Then also the contents of *D* fails and program goes to 4.
4	*XK14QCD*	The Q-bit registers a one in *B* for the pulse seen. *XCD* adds (15) into a cleared *A*. We shall suppose that in 15 an already partially built-up number *a* is present.
14	*X7K8D*	Start waiting 80 ms by repeating *AD*000 eight times. The instruction *AD*000 tries to write on 000 but track zero is locked. Hence this has no effect. But it must wait for 000 and thus loses 10 ms. In the meantime the repeating instruction has added nine times (15) = *a* to the accumulator, thus forming 10*a* in *A*. The repetition ends in 8.
7	*AD*000	
8	*X5K96U7*	Start repeating (5) = *A* for 96 times. This takes 60 ms. *A* is a harmless non-waiting instruction.
5	*A*	The repeating instruction tests *U7*. When a next pulse arrives within that time, the repeating as well as the repeated instructions are *A*-instructions and the program goes back to 4 where a next one is noted down in *B*. The forming of 10*a* is done again. When no pulse arrives within 60 ms, the decimal digit *S* is complete in *B*, zero being represented by 10 (*A*) = 10*a*. The repetition ends in 6.
6	*X011K3*	Borrow an instruction from track zero and modify it with the digit *S*. (011) = *X12K5* = *X12K* + 8192 − 10 hence (011) + *S* = *X12K* + 8192 − 10 + *S*.
	X12K5 + 8192—10 + *S*	For all digits *S* < 10 this is a jump to 12. But in case of *S* = 10 the instruction just becomes *X13K*.
12	*X017·3*	Add the digit *S* to 10*a* thus having performed the conversion.
017	*X13K1*	Via a borrowed instruction on 017 it comes to 13. In case of a digit *S* = 10 the instruction on 12 is skipped and nothing is added.
13	*X9K2400BCU7*	Go into the 1500 ms wait cycle by repeating (9) 2400 times.

(cont'd)

9	*AE*15	Store $10a+S$ into 15. This action is done repeatedly: clear *B*. In the meantime *U*7 tests the dial again. When a pulse arrives within 1500 ms, the after-action of *U*7 fails and the program goes to 4 again. When ready, repetition ends in 10.
10	return instruction	

6.4 Punching the Contents of the Store in Binary Form

We shall suppose that t words in the store from address n onward have to be dumped in binary form on the tape. The format shall be the same as for binary input, i. e. the 33 bits of the word will be punched as 7 characters of 5 bits each and of which the 5th bit of the most significant symbol and the first bit of the least significant symbol will be 0 (cf. Appendix Section 7).

Program:

4	return instruction
5	*A*
6	*X*011*LE*26
7	*AC*11*V*
8	*X*012*CE*11
9	*ABE*15
10	*X*7*K*16*IBC*
→ 11	*ACnE*26
12	*X*013*LE*27
13	*X*9*K*1*QBCDV*2
14	*X*5*K*16
15	$-t$

The program is entered at 11 by *X*11*K*1 with $(A) = 0$, $(B) = 0$.

Action:

11	*ACnE*26	Set up 5th hole of first symbol = 0. Extract (n) = word to be punched.
12	*X*013*LE*27	Set up 4th hole on punch and shift next bit to a_0. A small piece of punching program is borrowed from track zero. *B* was clear initially.
013	*NLE*28*Q*	Set up 3rd hole on punch. Shift a one into the least significant side of *B*. This one travels left during the punching of seven symbols and comes to b_0 right at the end of punching the 7th symbol. All other ones shifted into *B* by other than the first symbol, have no meaning.
014	*NLE*29	Set up 2nd hole.

015	*NE30*	Set up 1st hole.
016	*NE31*	Punch symbol. Note that even setting up the holes and punching are microprogrammed.
017	*X13K1*	Go to 13.
13	*X9K1QBCDV2*	Test shifting count in *B*. When not all seven symbols have been output it fails and program comes to 14.
14	*X5K16*	Introduce a time delay of an extra revolution by repeating a harmless instruction in 5. Only once every 20 ms a symbol can be punched owing to the speed of the punch. For a faster punch this delay could be changed.
5	*A*	
6	*X011LE26*	Set up 5th hole of next symbol and shift.
011	*X12K5*	Borrow (011). Repeat from 12 until all 7 symbols have been punched. Then:
	.	
	.	
	.	
	.	
	.	
13	*X9K1QBCDV2*	Test shifting count in *B*. All symbols have been punched and test succeeds. *XQBCD* adds (15)+1=−*t*+1.
9	*ABE15*	Store augmented count again in 15.
	X10KQBCDV2	The after-action again tests with *V2* but now (*B*) = count!! As long as count is negative, output must go on.
10	*X7K16IBC*	Introduce a time delay for the punch. *IBC* prepares a carry = 1 and clears *B*.
7	*AC11V*	Although repeated 16 times, augment extraction instruction with 1.
8	*X012CE11*	Store augmented extraction instruction *An*+1*CE26* in 11. Clear *A*.
012	*X11K1IV*	*IV* is not active. Start again in 11. When at last all words have been punched:
9	*ABE15*	
	X10KQBCDV2	Test of after-action: on *X10K . . . V2* fails and program goes to 4.
4	return instruction	

6.5 Read and Print Text

Although normally all printing is done via the standard output program, this very short register program is just meant for printing titles on the supervisory typewriter even when no standard output program is present in the machine.

The standard output program is necessary for all normal printing and can arrange for all types of digit lay-out. It has been made because operating the printer by micro-programming it is no easy matter. This will become clear when it is realised that the only means to influence the output teleprinter is by transferring the sign bit of A into a special flip-flop through a gate operated by selecting register 25. A teleprinter requires a signal of a structure as depicted in Fig. 3 (signal to teleprinter is 1 in quiescent state).

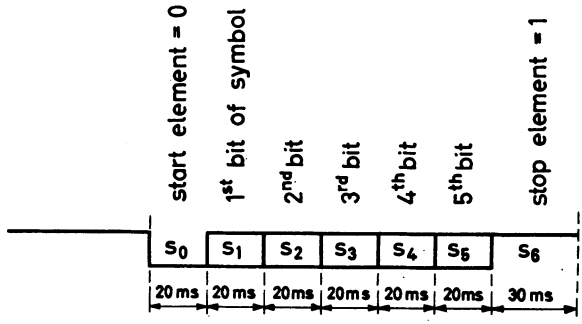


Fig. 3. Structure of the signal required by teleprinter

The teleprinter reverts to rest if signal (= contents of output flip-flop) remains 1. Or a next start can follow. The separate bits will be designated with $s_0 - s_6$.

The correct timing has to be generated by micro-programming. Fortunately the symbols can be read from tape in the same form as they are to be printed. The program has been designed in such a way that it stops printing as soon as a blank symbol is read (blank is a non-existant symbol in teleprinter code). The filling of the registers is as follows.

Program:

4	*X5K1*
5	*AC3*
6	*X7K1*
7	*X12K1BDV3*
8	return instruction
9	*X017CV3*
→ 10	*X001C12*
11	*X011C3*
12	*X9K1I*
13	*AC23V4*
14	*X012RC25*
15	+64

The program is entered at 10 with a cleared B-accumulator.

Action:

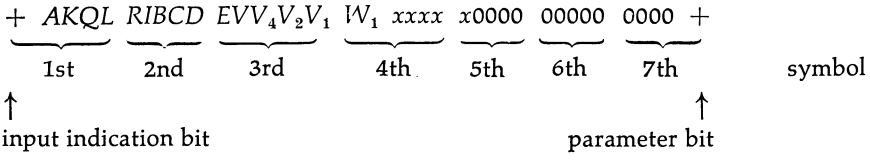
10	*X*001*C*12	Put *X*9*K*1*I* in *A* as shifting count. Only *I*-bit is important. Go to 001 for reading a symbol.
001	*X*003*LIB*26	Read symbol from tape. (For explanation reference is made to the example of the tape copying program.)
003	*NLIB*27	
004	*NLIB*28	
005	*NLIB*29	
006	*NLIB*30	
007	*NL*31*V*1	$(B) = 0$————0 $s_5\ s_4\ s_3\ s_2\ s_1$ $(s_6 = 0)$.
008	*X*001*RV*1	*I*-bit in *A* shifted out. Instruction fails.
	*L*31*V*1	After-action fails. Instructions executed at level 009 of drum.
4	*X*5*K*1	(level 010)
5	*AC*3	$(A) = 0$————0 $s_5\ s_4\ s_3\ s_2\ s_1\ s_0$ (level 011)
	*XK*6	After-action.
6	*X*7*K*1	
7	*X*12*K*1*BDV*3	Test symbol in *A*. If $s = 0$, (7) fails and program returns on 8. If no blank: add stop bit $s_6 = 1$ with *XBD* facility. (level 014)
12	*X*9*K*1	$(B) = 0$————0 $s_6\ s_5\ s_4\ s_3\ s_2\ s_1\ s_0$ (level 015)
9	*X*017*CV*3	Test if all bits have been put on printer.
		Clear *A*. (level 016)
017	*X*13*K*1	Level 017 was just reached in time.
13	*AC*23*V*4	Transfer right-most bit of *B* to left-most bit of *A*. (level 018)
	*X*14*K*	After-action. (level 019)
14	*X*012*RC*25	Set up this bit on printer flip-flop and shift bit off in *B*. From now on next bit has to wait 20 ms. (level 020)
012	*X*11*K*1*IV*	Level 012 is reached after 23 word times waiting.
11	*X*011*C*3	$(B) \rightarrow A$.
011	*X*12*K*5	Level 011 is reached after almost another revolution (= 10 ms).
12	*X*9*K*1*I*	Go back to 9 for next digit.
9	*X*017*CV*3	If all digits including stop bit have been set up test fails.
	*X*10*K*	After-action $(B) = 0$.
10	*X*001*C*12	Start reading next symbol. Reading a symbol. just wastes an extra 10 ms making up for 30 ms of stop bit.

APPENDIX

7. The Binary Input Program on Track Zero

For the understanding of the binary input program which can be supposed to be permanently stored in track zero it is necessary to know the composition of a binary word on tape. A binary word is represented on tape by 7 symbols of 5 bits each. Of the available 35 bits only 33 are necessary for the word, hence 2 are available for other purposes. One has been given the significance that the word must not be placed in the store by the so called store instruction but that the store instruction itself has to be replaced by that word. In that way input can be started at arbitrary location by giving the appropriate *input indication*. The other spare bit is used for relative addressing by adding (9) to the word when this bit is present. With the help of it programs can be made relocatable.

The composition of the word is as follows:



Track zero has to fulfil the following requirements:

It must provide a stop at 000.

Blank tape at start must be skipped.

All symbols 2–31 must leave track zero and are treated elsewhere.

The opening symbol 1 indicates that binary tape follows. The first word read must replace the store instruction, following words are to be stored until another input indication follows.

A rudimentary punch routine is included in track zero.

The coding and explanation of track zero is as follows:

000	*X000KE4U7*	Loop stop on 000. Loop until *U7* is pressed and released. Go on to 002.
001	*X003LIB26*	Read symbol. In the meantime, shift shifting count in *A* left.
002	*X020E4*	Go on to 020. *E4* is of no significance for the present use.
003	*NLIB27*	
004	*NLIB28*	Read symbol and shift it into *B*.
005	*NLIB29*	Shift shifting count in *A* at the same time.
006	*NLIB30*	
007	*NL31V1*	Step tape. Test shifting count. If $(A) \geqq 0$ test fails and word is ready.
008	*X001RV1*	If $(A) < 0$, undo *L* of previous order. Read next symbol. If *V1* fails: go to special outlet on 4 (cf. some of the tape programs).

009	*NRB*31	If 007 failed, do step here and undo after-action of *LIB*30. Hence of the 35 bits, two are now in *A*, 33 in *B*.
010	*X*012*RB*9*V*4	If parameter bit is present: add (9) and shift off parameter bit. Otherwise after-action of 009 does right shift. In both cases no carry hence a borrow has been put in the carry trap.
011	*X*12*K*5	Instruction only used in dial program.
012	*X*11*K*1*IV*	Go to 11 and repeat it once. *IV* subtracts 1 from *A*. *A* just contained input indication bit. Hence no input indication on (*A*) becomes −1 (all ones). If input indication: (*A*) = 0. Normally (11) = *ADnQBC*11*V*1: store word built up in *B* into *n*. *QBC*11 augments instruction itself with 1 thus forming *ADn*+1*QBC*11*V*1. All this only when (*A*) < 0. Then go on to 12. (12) = *X*001*BCE*11*V*1: put augmented store instruction again in 11. Or in case (11) failed: replace store instruction by another. *V*1 now succeeds in all cases as after-action *X*12*KIV* has subtracted 1 from *A*. Return to 001 with cleared *B* and read next word.
013	*NLE*28*Q*	Rudimentary punching cycle for use by tape programs.
014	*NLE*29	
015	*NLE*30	
016	*NE*31	
017	*X*13*K*1	
018	*NKKBCK*3	If blank tape has been read (*B*) = 2. Go on to 022 and read next symbol.
019	*X*001*BCE*11*V*1	If 1 has been read (*B*) = 1: prepare for binary reading *X*001*BCE*11*V*1 → *B*. Go to 21!
002 → 020	*X*029*U*6	Test *U*6. If *U*6 = 1 go to 029. If *U*6 = 0 go to 022.
018 → 021	*X*019*BE*12	Put *X*001*BCE*11*V*1 in 12 and go to 019 (now executed as instruction). This also put *X*001*BCE*11*V*1 in 11. Hence first word read will always replace 11 by suicide.
018 → 022	*NBC*26	
023	*NLB*27	
024	*NLB*28	Read opening symbol negatively in cleared *B*.
025	*NLB*29	Only case that *S* = 0 and *S* = 1 have to be
026	*NLB*30	considered here.

(cont'd)

027	*N*31*Q*	Step tape and add 1 to $-S$. Hence $(B)=1$ for $S=0$ $(B)=0$ for $S=1$ $(B)<0$ for $S\geqq 2$
028	*X*32*K*3*QIBCV*2	Test if (B) is positive. If negative: go to outlet for other symbols $s\geqq 2$. For $S=0$ and 1 test fails.
020 → 029	*X*34*U*7	If $U7=1$ go to 34. The contents of 34 is used to restart a program. This is of no concern for binary input.
030	*X*018*IC*1	Put shifting count -1 in A. After-action of *N*31*Q* has made $(B)=2$ for $S=0$, $(B)=1$ for $S=1$. Shifting count $(A)=-1$ is only becoming positive after 7 symbols having been read.
031	*X*8191	If $U7=0$ on 029 go to 8191 as special outlet. This place can only be reached when going to 000 with $U7=0$, $U6=1$. Also used as constant.

Normally the form of the input indication to start input of words at location n has the form *ADnQBC*11*V*1 (in binary form). When it is necessary to fill registers they can only be filled individually by preceding each word with the input indication *X*001*BCEmV*1 (store in m and read next word) when m is the register to be filled. *V*1 enables the input program to replace this store instruction. Only for filling 11 and 12 another trick is needed. After having filled all necessary registers, 11 and 12 can be filled as the last ones by giving an input indication of the form *X*030*QIBCE*12. Hence (11) = *X*030*QIBCE*12 and the next word is stored in 12, replacing the usual *X*001*BCE*11*V*1. Program is directed to 030 with –1 in B. Via 030 control arrives at (018) = *NKKBCK*3 taking in *X*001*BCE*11*V*1 in B but as $(B)=-1$ the *K*3 does not go to 020 but to 019 as next instruction. Hence *X*001*BCE*11*V*1 is executed as instruction putting *X*001*BCE*11*V*1 in 11 without destroying (12). The next word read is overwriting (11) = *X*001*BCE*11*V*1 by suicide action and program starts action on 11.

Bibliography

[1] Samelson, K., Bauer, F. L.: Sequentielle Formelübersetzung. Elektronische Rechenanlagen **1** (Nov. 1959) No. 4, pp. 176—182. (Engl. translation entitled "Sequential Formula Translation", Communications ACM **3** (Febr. 1960) No. 2, pp. 76—83.)

[2] Dijkstra, E. W.: Recursive Programming. Numerische Mathematik **2** (Oct. 1960) No. 5, pp. 312—318.

[3] Davis, G. M.: The English Electric KDF 9 Computer System. The Computer Bulletin **4** (Dec. 1960) No. 3, pp. 119—120.

[4] Lonergan, W., King, P.: Design of the B 5000 System. Datamation **7** (1961) No. 5, pp. 28—32.

[5] Programming for EDSAC 2. The University Mathematical Laboratory, Cambridge, England 1958.

[6] van der Poel, W. L.: The Logical Principles of Some Simple Computers. Dissertation, University of Amsterdam. Uitgeverij Excelsior, 's-Gravenhage 1956.

[7] Kassel, L.: *George* Programming Manual. Report ANL—5995. Argonne National Laboratory, Lemont, Ill. 1959.

[8] WILKES, M. V., WHEELER, D. J., GILL, S.: The Preparation of Programs for an Electronic Digital Computer. Addison Wesley Press, Cambridge, Mass. 1951. Revised Edition 1957.
[9] KUDIELKA, V., WALK, K., BANDAT, K., LUCAS, P., ZEMANEK, H.: Programs for Logical Data Processing. Research Report, Mailüfterl Volltransistor-Rechenautomat, Vienna, February 1960.
[10] WILKES, M. V., STRINGER, J. B.: Micro-programming and the Design of the Control Circuits in an Electronic Digital Computer. Proc. Cambridge Philosoph. Soc. **49** (April 1953) Part 2, pp. 230—238.
[11] BILLING, H.: Die im Max-Planck-Institut für Physik und Astrophysik entwickelte Rechenanlage G 3. Elektron. Rechenanlagen **3** (April 1961) No. 2, pp. 83—84.
[12] BILLING, H., HOPMANN, W.: Mikroprogramm-Steuerwerk. Elektron. Rdsch. **9** (Oct. 1955) No. 10, pp. 349—353.
[13] TR 4 — Telefunken. Digital Computer Newsletter **12** (Oct. 1960) No. 4. Reprinted in Communications ACM **3** (Oct. 1960) No. 10, pp. 586—589.
[14] GILMORE JR., J. T., PETERSON, H. P.: A Functional Description of the TX—0 Computer. Memorandum 6 M—4789, Lincoln Laboratories, Massachusetts Institute of Technology, Cambridge, Mass., November 20, 1956.
[15] CARR III, J. W.: Programming and Coding — Section 17, Microprogramming. In: Handbook of Automation, Computation, and Control Vol. 2 (Eds.: E. M. GRABBE, et al.). John Wiley, New York 1959, pp. 2·251—2·257.
[16] GILL, S.: A Process for the Step-by-step Integration of Differential Equations in an Automatic Digital Computing Machine. Proc. Cambridge Philosoph. Soc. **47** (Jan. 1951) Part 1, pp. 96—108.
[17] Stantec ZEBRA Program Manual. Standard Telephones & Cables, Ltd., England 1958.
[18] DEVONALD, C. H., FOTHERINGHAM, J. A.: The Atlas Computer. Datamation **7** (1961) No. 5, pp. 23—27.
[19] GILL, S.: Neue Wege beim Bau von Großrechenanlagen (Atlas). Elektron. Rechenanlagen **3** (April 1961) No. 2, pp. 81—83.
[20] Bendix G—20 System. Communications ACM **3** (May 1960) No. 5, pp. 325—328.