FTD-TT-65-972

FOREIGN TECHNOLOGY DIVISION

AIR FORCE SYSTEMS COMMAND

THE "KIEV" COMPUTER
A MATHEMATICAL DESCRIPTION

By

V. M. Glushkov and Ye. L. Yushchenko

FOREIGN TECHNOLOGY DIVISION

FTD-TT- 65-972/1+2

UNEDITED ROUGH DRAFT TRANSLATION

THE KIEV COMPUTER; A MATHEMATICAL DESCRIPTION

BY: V. M. Glushkov and Ye. L. Yushchenko

English Pages: 227

TM5001617

PREPARED BY:

TRANSLATION DIVISION
FOREIGN TECHNOLOGY DIVISION
WP-AFB, OHIO.

FTD-TT- 65-972/1+2 Date 28 March 19 66

AFLC-WPAFB-JUN 66 72

S

V. M. Glushkov and Ye. L. Yushchenko

VYCHISLITEL'NAYA MASHINA "KIEV" MATEMATICHESKOYE OPISANIYE

Gosudarstvennoye Izdatel'stvo Tekhnickeskoy Literatury

USSR
Kiev - 1962

Pages: 1 - 183

FTD-TT-65-972/1+2

6P2.15
G55

This book deals with a description of the general-purpose "Kiev" computer developed at the VTs AN UkSSR.

The book discusses the basis for selection of the mathematical parameters of the computer, a description of the set of operations, standard routines, and test routines, and also discusses problems pertaining to automatic programming, including the special algorithm language developed at the VTs AN UkSSR.

The book is designed for engineers, scientific workers, students, and graduate students, working in the field of computer mathematics and computer engineering, as well as for specialists wishing to use electronic computers in their work.

Reviewer,
Dr. of Physical-Mathematical Sciences Prof. L.A. Kaluzhnin

Editorial Board
for Literature on Problems of Power, Radio, and Television

Chief of Editorial Board,
Engineer M.G. Pisarenko

A

TABLE OF CONTENTS

PREFACE

Studies toward the creation of the "Kiev" general-purpose electronic computer were begun in the Laboratory of Computational Techniques of the Mathematics Institute of the AN UkSSR at the initiative of Academician B.V. Gnedenko by a group which had, under the leadership of Academician S.A. Lebedev, developed the first electronic computer in Europe, the Mesm. Together with senior scientific associates L.N. Dashevskiy and Ye.L. Yushchenko, B.V. Gnedenko supervised the initial stage of computer development. The final stages of development of the "Kiev" computer were supervised by Academician of the AN UkSSR V.M. Glushkov, L.N. Dashevskiy and Ye.L. Yushchenko. Final work on the "Kiev" computer (engineering design, assembly, and debugging) was carried out at the Computer Center of the AN UkSSR, created in December of 1957 [4, 6].

Many scientific workers and engineers of the Computer Center of the AN UkSSR participated in development, adjustment, and debugging of the computer. Important independent work was performed by S.B. Pogrebinskiy, Ye.A. Shkabara, as well as by L.M. Abalyshnikova, A.I. Kondalev, V.V. Kraynitskiy, V.D. Losev, A.A. Barabanov, L.P. Bystrova, V.I. Dvortsyn, Z.S. Zorina, A.Ya. Zubatenko, L.N. Ivanenko, A.A. Letichevskiy, V.S. Motorna, and others. V.S. Korolyuk and I.B. Pogrebyskiy, senior scientific associates of the Mathematics Institute, participated during the initial period in studies on the selection of computer mathematical parameters.

Development of the "Kiev" computer raised the problem of creating a fairly powerful computer which could fit in with the fundamental equipment of the Computer Center of the AN UkSSR. This development work was the response of the group to the resolution of the XX Party Congress of the CPSU on the need for broad-scale development of electronic computer technology.

The solution of this problem first required determination of a reasonable compromise among speed, memory size, and programming conveniences on the one hand, and computer reliability on the other.

The design of the computer was based on the asynchronous principle, which is particularly convenient both from the viewpoint of greater simplicity in over-all machine debugging, as well as from the viewpoint of the possibility of successive (unit-by-unit) modernization of the computer. In addition, this principle offerred great possibilities for using not only individual elements, but also entire units of the computer in the creation of specialized computer and control installations.

It should be noted that machine design was subject to modernization and the modernization process continued not only during over-all machine debugging, but even after it was placed into operation. Thus the description of the computer given in this book corresponds to its state as of 1 January 1960. Some of the modifications made or planned are described in this monograph.

This book describes the mathematical principles realized in the "Kiev" computer, the instruction system, the standard and test routines, as well as a system of automatic programming based on the algorithm language developed at the Computer Center of the AN UkSSR, and used as the input language for translator programs on other machines as well.

The first and second chapters of this book were written by V.M. Glushkov; Ye.L. Yushchenko wrote the remaining chapters.

Comments and suggestions pertaining to this book should be addressed to Kiev, 4, Pushkinskaya, 28, Gostekhizdat UkSSR.

Chapter 1

FUNDAMENTAL CHARACTERISTICS OF "KIEV" COMPUTER

GENERAL DATA

The "Kiev" electronic digital computer is a general-purpose automatic computer, designed for scientific and engineering calculations. This is a three-address machine, and works at fairly high speed — up to 15 thousand additions, 5 thousand multiplications, or 3 thousand divisions per second. If we assume that 80% of all operations are of the addition type, 15% of the multiplication type, and 5% of the division type, the average operating speed of the computer (working with the high-speed memory) the actual operating speed is still higher, since the computer contains provision for certain additional measures that will increase the speed (for example, skipping of zero addresses).

The "Kiev" computer operates with 40-digit binary numbers (not counting the sign bit) with a fixed point preceding the highest-order digit position. The fact that the operation set contains a special number normalization operation permits us to realize the floating-point mode fairly simply by means of programming. In subsequent modernization of the computer, it is envisioned that provision will be made for circuit realization of the floating-point mode.

The high-speed storage unit (OZU), with 1024 41-digit binary words uses ferrites with rectangular hysteresis loops. In addition, their is a passive storage element (PZU) (also using ferrites) with a 512-word capacity. We later intend to increase OZU capacity by another 512 words.

OZU access time is about 10 μsec, and the PZU access time requires only 4 μsec. This makes it possible to increase machine speed, since standard routines and universal constants widely employed in quite diverse programs are stored in the PZU. This possibility is actually realized in the machine, since the "Kiev," being an asynchronous machine, makes fullest use of the speed of each separate unit.

The external storage (VZU) for the "Kiev" computer uses three magnetic drums with a total maximum permissible capacity in excess of 9 thousand words (more exactly, 9864 words). Owing to the use of a serial-parallel system of writing words onto drum, the access time for a single word is short (60 μsec) with relatively low drum speeds (1500 rpm).

Each drum is in turn divided into channels. The average writing time (with allowance for the operating time of the channel-switching relays) in a call to a drum is about 25 msec.

A further expansion in external storage is provided by connection of an additional-magnetic tape unit.

As a temporary expedient, modified peripheral equipment for the "Ural" computer and punched-card loading are being used as periphery equipment for the "Kiev" computer (punched-tape loading, number printer, output puncher, keyboard and verifier-reader devices).

Numbers are loaded and read out in the decimal number system (10 decimal digits and the sign), while instructions are loaded and read out in the octal system (14 octal digits). Loading speed is about 75 words (numbers or instructions) per second, and readout speed is about 100 words per minute for printing out, and 150 words per minute for output punching. Numbers are converted directly from decimal into binary during the loading process by means of the standard "load numbers" instruction (no special subroutine is used). When blocks of numbers are read out, it is first necessary to use a special subroutine to convert from the binary system into the binary-coded decimal system.

A special "load instructions" instruction is used to load instructions; it eliminates conversion of codes; conversion from the octal system in which instructions are written to the binary system is carried out while the tape is being punched on the keyboarder. Printout is switched from the decimal number system (four numbers) to the octal system (four instructions) or vice versa by throwing a special switch on the readout unit. The machine also has an input and readout punched-card system.

The main units of the computer are the arithmetic unit (AU), control unit (UU), high-speed memory (OZU), passive memory (PZU), and magnetic-clear recording control (UMZ), installed in separate cabinets, each having a separate power-supply unit and joined to the remaining cabinets by as few connections as possible. The magnetic drums are located in three separate racks together with the UMZ cabinet; the input and output devices are located on separate stands. There is also a central control console.

The energy source is three-phase alternating current at 380/220 v; the power drawn is about 25 kw. The computer uses about 2300 small vacuum tubes of the button series (mainly the 6N1P type) and 10,000 germanium diodes (mainly the D1D).

CODING SYSTEM AND PROGRAM REGISTERS OF THE COMPUTER

For coding of numbers (Fig. 1), the digit positions in a location of the "Kiev" computer are numbered from right to left from the 1st through 40th digits. The 41st digit position is the sign bit; the symbol 0 corresponds to positive numbers, and the symbol 1 to negative numbers. The radix point is located before the highest-order digit position. Thus, the range of numbers represented in the computer is from -1 to $+1$ with an accuracy of up to $2^{-40} \approx 10^{-12}$.

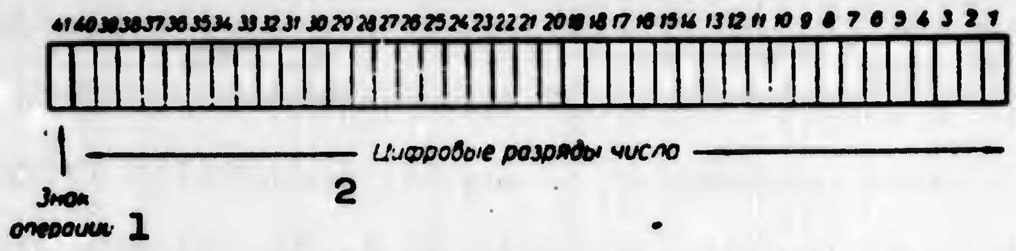


Fig. 1. Sign of operation; 2) numerical digit positions of numbers.

In the binary-coded decimal system, numbers take up all 40 bit positions, not counting the sign bit. Since 4-digit binary codes are used to code decimal digits, in this case the numbers represented will have only 10 decimal digits with an accuracy of representation ranging up to 10^{-10}. As in the preceding case, the radix point is fixed ahead of the highest-order digit position.

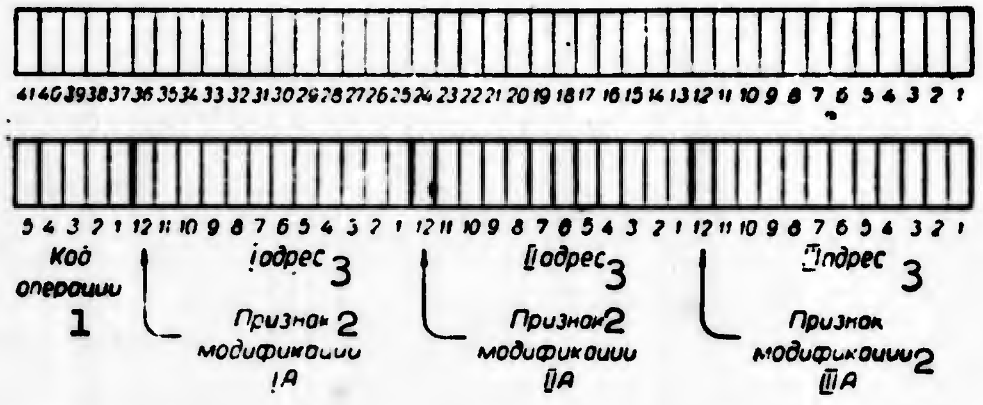


Fig. 2. 1) Operation code; 2) modification indicator; 3) address.

We should note that binary-coded representation of numbers in the computer is used only for numbers punched into tape, and for those numbers in OZU that are suitable for reading out by means of the special subroutine for conversion from binary to binary-coded decimal. In the remaining cases, numbers in memory units are stored in the straight binary code. In the arithmetic unit and the control unit, numbers are transferred in both the straight binary code and the ones complement.

Instructions for the "Kiev" computer take up 41 bit positions each.

The structure of an instruction is illustrated in Fig. 2. As we can see from the figure, the five highest-order bit positions (from 37 to 41, inclusive) of an instruction word are taken up by the operation code. The remaining 36 bits form the address field of the instruction. Thus, 12 bit positions are set aside for each of the three addresses.

In loading and printing out, instructions are represented in the octal code. Here four octal digits are set aside for each of the three addresses, and two octal digits for the operation code. Thus every instruction is represented as a 14-digit octal integer.

Addresses are coded as follows (the same coding is used for all three instruction addresses): locations in high-speed memory (OZU) have octal addresses from 0000 to 1777, inclusive; here the number zero is always stored for the 0 address.* The PZU locations have addresses from 3000 to 3777, inclusive. In this case, the passive memory of the "Kiev" machine has three types of locations;

1) 8 locations for manually set-up words, addresses 3000-3007;

2) 184 locations for fixed wired-in words, addresses 3010-3277;

3) 320 locations for interchangeable wired-in words, addresses 3300-3777.

The last location of the last block in the interchangeable wired-in storage block (with address 3777) is used for connection of the random-number generator.

The hand-assembled words take the form of 41-digit switch-controlled locations (brought out to the control console) in which any words may be set up manually.

The fixed wired-in storage block (PSP) contains permanently wired codes for the constants most frequently used in the solution of problems, together with certain standard subroutines:

The fixed wired-in memory contains:

1. Constants (location numbers 3010 to 3077).

2. Subroutine for converting from the binary system to the decimal system.

3. The subroutine for computing the logarithm ln x.

4. The subroutine for computing $\frac{1}{2}$ sin x.

5. The subroutine for computing $\frac{1}{2}$ cos x.

6. The subroutine for computing $\sqrt{a}$.

7. The subroutine for computing $(1/4)e^x$.

8. The subroutine for computing $(1/\pi)$ arc sin x.

9. The subroutine for computing $(1/\pi)$ arc cos x.

The interchangeable wired-in memory (SSP) is realized in the form of separate blocks with 64 words each, containing library routines. Each of the SSP blocks is set up as a plug-in unit, and can be replaced at any time (independent of the other blocks) by a new unit having the same capacity; depending on requirements, various of these units are connected to the computer.

For coding of OZU and PZU addresses, 11 bits are sufficient; the presence of a one in the 11th bit position of an address is a sign that the PZU is to be used, or the presence of a zero in the 11th bit position of an address is an indication that the OZU is to be used.

The extra 12th bit position available in each address with the memory capacity used is not needed for address coding; it is used as an indicator of address modifiability. Whenever a zero is contained in the 12th digit position of any address instruction, the machine will perform a call to memory on the basis of this address. If, however, the 12th digit of the address is a one, the corresponding call is effected on the basis of the so-called effective address, which is obtained by adding to the address shown in the instruction the content of a special address-modification register or A-register. This register consists of

10 digit positions, and can be used to store positive numbers from zero to 1777, inclusive. With address modification, there is no overflow blocking so that it is possible, for example, to modify an address 2666 with the aid of the number 1323 in the A-register and obtain as the modified address 2667 + 1323 – 4000 = 213.*

The address-modification register is one of the computer program registers. By program registers we mean those registers that carry information from one machine operating cycle (time required to execute one working command) to another.

In addition to address-modification registers, the "Kiev" machine has two additional special program registers: a link register (or R-register) and a loop register (or Ts-register). Both these registers contain 11 bits each.

The program registers also include the 11-bit instruction counter (or S-register) and the 41-bit instruction register (or K-register). The remaining registers of the "Kiev" computer are not used in program writing, and do not carry information from cycle to cycle. They might be called microprogram registers, since exchange of information among them is carried out in accordance with fixed microprograms built into the computer during construction, and not under the control of the programmer.

OPERATIONS PERFORMED BY THE COMPUTER

The operation field of an instruction for the "Kiev" machine contains five binary digits, which makes it possible to code 32 operations. Since the operation with code zero is the zero operation, i.e., is not executed, it is possible to code only 31 operations. In fact, 29 operations are realized in the computer.

The operations executed by the "Kiev" machine are divided into the following five groups: arithmetic, logical, control-transfer, operations with address-modification register, operations with periphery equipment (including magnetic-drum storage).

The execution cycle for each operation begins after the address of the corresponding instruction has been set up on the instruction counter and concludes with formation in the instruction counter of the address of the next instruction to be executed. Here each successive cycle begins with the transfer to the instruction register of the instruction whose address is contained in the instruction counter at the end of the preceding cycle. As a rule, the content of the instruction counter is increased by one in the course of the cycle, so that the next instruction in the sequence will be taken from the storage location whose address is one greater than the address of the preceding instruction. In this case, we shall say that the next instruction in sequence is executed. This natural sequence of instruction execution may be interrupted only by execution of control-transfer or group operations when upon satisfaction of given additional conditions, the address counter receives the content of address III or address II of the current instruction or the content of the link register which (without an increase by one) also serves as the address of the next instruction. For brevity, we agree to say that in the given case the control transfer is governed by address II or III or by the link register.

We agree to let a_1, a_2, a_3 represent, respectively, addresses I, II, and III of an instruction and, moreover, for any address $\underline{c}$, we use 'c to represent the word stored on the basis of this address.

Turing to the characterization of the operations performed by the computer, we shall first describe the arithmetic operations. There are nine such operations in the computer: ordinary addition + (the octal operation code is 01), subtraction – (code 02), subtraction of absolute values $| - |$ (code 06), multiplication without rounding $\times$ (code 10), multiplication with rounding ⊠ (code 11), division : (code 12), number normalization N (code 35). The arithmetic operations also include instruction addition SℓK (code 03), and cyclic addition Ts + (code 07).

All arithmetic operations are characterized by the fact that where there is a one in the 12th digit of any of the three instruction addresses, the content of the A register is added to the corresponding address, i.e., all operations are executed with effective addresses. Control transfer is normal: after execution of an instruction with an arithmetic operation, the next instruction in the sequence is always executed.

When the operations of addition, subtraction, division, and instruction addition are executed, the result may prove to be greater than or equal to unity in absolute value. In this case, there is provision for halting the computer with delivery of an appropriate signal. By setting a special switch on the control console, we can keep the computer from halting when the available number of digit positions is exceeded; the machine will continue to run even when there is an overflow; in this case, it skips the next instruction, naturally losing the highest-order digits of the numbers.

In execution of an ordinary addition operation (+'), to the number $'a_1$ there is added (algebraically) the number $'a_2$, and the resulting sum is stored in accordance with address a_3 (here and in the ensuing discussion, all addresses are effective addresses!).

Instruction addition (SℓK) differs from ordinary addition in that the number $'a_1$ is not added to the number $'a_2$ but to its absolute value (i.e., to the number $'a_2$ taken with a plus sign), while the resulting sum is again given the sign of the number $'a_2$. This operation is used for instruction readdressing: the modified instruction is stored according to address II, while restoration of the sign in the sum ensures that the highest-order digit of the operation code will remain unchanged after readdressing (we recall that the highest-order digit of the operation code in an instruction corresponds to the sign bit in a number word).

Cyclic addition (Ts +) differs from ordinary addition only in the fact that their is no blocking under an overflow. End-around carry occurs from the sign bit to the lowest-order bit of the adder. The cyclic-addition operation is used chiefly in check routines.

With the subtraction operation (–) the difference $'a_1 - 'a_2$ is stored on the basis of address a_3; with the subtract absolute values operation ($| - |$), the same thing is done with the difference of the absolute values $|'a_1| - |'a_2|$.

In the operations of multiplication without rounding ($\times$) and division (:) address II determines the location in which the numbers $'a_1 \times \times 'a_2$ and $'a_1 : 'a_2$, respectively, are stored. Here in the case of multiplication with rounding ($\boxtimes$) a one is added to the highest-order digit discarded.

We should look in more detail at the operation of number normalization (N). We shall say that a number n is normalized (in the binary number system) if it is represented in the form

$$n = \pm 2^k m,$$

where $m = 0.1\, m_1 m_2 \ldots$, while k is a positive or negative integer. The number k is called the exponent of the number n and the number m is called its mantissa. The inequality

$$1 > m > \frac{1}{2}.$$

is always valid for a mantissa.

In the "Kiev" computer, the maximum value of an exponent is taken as 63. Thus, six binary digits are needed to represent an exponent. The normalization operation consists in normalization of the number $'a_1$; the exponent of the normalized number is transferred on the basis of address II (occupying the six lowest-order digits of the word), and the mantissa according to address III.

We now turn to the logical operations, of which there are four in the computer: logical shift L $\rightarrow$ (the octal operation code is 13), logical multiplication $\vee$ (code 14), logical multiplication $\wedge$ (code 15), and the digit-by-digit operation of logical inequality $\simeq$ (code 17). Execution of all the logical operations is accompanied by the normal control flow (the next instruction in the sequence is executed). As in the case of the arithmetic operations, all three addresses may be modified.

In the logical-shift operation (L $\rightarrow$) all 41 bits of the number $'a_2$ are shifted (including the sign bit) by a number of digits equal to the absolute value of the shift constant contained in the six lowest-order bit positions of location a_1. If the shift constant is positive, a left shift occurs; if the constant is negative, their is a right shift. Digits of the number that overflow the digit-position format during shifting are lost, and they are replaced at the opposite end by zeros (a shift by 41 or more digits will convert any number to a zero). The shifted number is transferred in accordance with address III.

The three remaining logical operations are performed digit-by-digit on all 41 bits of the words $'a_1$ and $'a_2$, so that the nth digit of the result will depend solely on the nth digits of the words $'a_1$ and $'a_2$. As in the preceding cases, the results are transferred according to address III.

The logical addition operation (disjunction) is represented by $\vee$ and is defined by the relationships $0 \vee 0 = 0,\ 0 \vee 1 = 1,\ 1 \vee 1 = 1,\ 1 \vee 0 = 1.$

The logical multiplication operation (conjunction) uses the symbol $\wedge$ and is defined by the relationships $0 \wedge 0 = 0,\ 0 \wedge 1 = 0,\ 1 \wedge 0 = 0,\ 1 \wedge 1 = 1,$ while the operation of logical inequality uses the symbol $\simeq$ and is defined by the relationships $0 \simeq 0 = 0,\ 1 \simeq 0 = 1,\ 0 \simeq 1 = 1,\ 1 \simeq 1 = 0.$

In the "Kiev" there are seven control-transfer operations: ordinary comparison Cr1 (octal operation code 04), comparison of absolute values Cr2 (code 05), exact comparison Cr3 (code 16), conditional jump on sign of number UPCh (code 31), conditional jump to subroutine UPP (code 30), jump on link register PRV (code 32), unconditional computer halt Ost (code 33). All addresses can be modified in all of these control-transfer instructions except for the jump on link register and halt instructions.

The ordinary comparison operation (Cr1) consists in comparing the numbers $'a_1$ and $'a_2$. If $'a_1 \leq {'a_2}$, control jumps to the instruction indicated by address a_3; if, however, $'a_1 \gg {'a_2}$, the next instruction in sequence is executed.

The compare absolute values operation (Cr2) differs from the ordinary comparison operation in that it compares not the numbers $'a_1$ and $'a_2$, but their absolute values (moduli).

The exact comparison operation (Cr3) means that control jumps to the instruction indicated in address a_3 if the numbers $'a_1$ and $'a_2$ coincide in magnitude as well as sign. Where the numbers $'a_1$ and $'a_2$ are not equal, the next instruction in sequence is executed.

Conditional jump on sign of number (UPCh) means that control is transferred to the instruction indicated by address a_2 if the sign of number $'a_1$ is positive, and to the instruction indicated by address a_3 in the opposite case. In the computer, zero may be either positive or negative (a zero obtained as the difference of two equal numbers will have a positive sign).

The conditional jump to subroutine (UPP) accomplishes the following operations: if the number $'a_1$ is positive (or zero) the next instruction in sequence is executed; if, however, the number $'a_1$ is negative, control jumps to the instruction shown in address a_3. Simultaneously, the number a_2 (not 'a_2!) is transferred to the link register (the earlier content of this register is erased). As a rule, every UPP is employed to go to a standard routine. Then the number a_2 indicates the address of the instruction to which the machine is to go after executing the subroutine. The standard routine must terminate in a special "jump in accordance with link register" instruction.

The link-register jump (PRV) represents an unconditional transfer of control to the instruction whose address is indicated in the link register. The address field of the PRV instruction itself is not used in this case, so that it may contain any values.

The halt operation (Ost) indicates an unconditional computer halt. As in the preceding case, the address field of this instruction is not used. In the "Kiev" computer, in addition to the program halt there are also two possible check halts. One of them stops the computer when it executes an instruction whose address has been set up on the special register at the central control console. The second check halt occurs where the third address of an instruction executed by the computer coincides with an address set up at the control panel by means of switches.

Let us turn to the description of the operations with address-modification register, which are the most complex operations performed by the "Kiev" computer. This group contains a total of three operations: setting of the A-register according to the specifer (specifier calling) F (octal code 34), beginning of group operation NGO (code 26), and conclusion of group operation OGO (code 27). When operations of this group are executed, any address modification must be particularly specified when each of the operations is described.

For the F operation, the part of the word 'a_1 from the 13th to the 24th digits inclusive (in other words, address II) is transferred to the A-register (the earlier content of this register is forced out).

In addition, the number 'm is transferrred into the location with address a_3; here m is the number that has been transferred into the A-register. Address II is not used with the F operation. Control flow is natural, i.e., after the F operation, the next instruction in sequence is executed. The F operation makes it possible to code cyclic processes of arbitrary complexity in fixed storage.

We should note that the F operation may be used simultaneously to transfer a code given by an address of its address (an address of rank two) into any location in the OZU.

Taking the address 0000 as a_3, we obtain the possibility of using the F operation solely to set the A-register. Here in any case, it is possible to use natural address modification in subsequent program instructions.

The other two operations of this group (NGO and OGO) are used in programming loops, and bear the special name of group operations.

With the NGO operation, the number a_1 (not 'a_1!) is transferred to the loop register, i.e., the content of address I of the NGO instruction, while the number a_2 (not 'a_2!) is transferred to the address-modification register, i.e., the content of address II of the NGO instruction. If after this, the contents of the loop register and address-modification register prove to be the same, control jumps to the instruction indicated by address a_3, otherwise to the next instruction in sequence.

With the OGO operation, the content of the address-modification register is increased by the content of address I of the OGO instruction (the number a_1, and not 'a_1:). If after this the contents of the loop register and the address-modification register are the same, control jumps to the instruction indicated by address a_3, otherwise to the instruction indicated by address a_2.

In order to perform an identical set of operations on a group of numbers situated in locations whose addresses form an arithmetic progression with exactly the same difference p, we can use the NGO operation to transfer a number equal to the content of the address-modification register into the loop counter; this is needed to execute the indicated set of operations on the last numbers of the group. Then the program segment realizing the given set of operations on the group of numbers may begin with the NGO instruction and end with the OGO instruction. Between these instructions there should be the given set of instructions, for which it is necessary to insert ones in the 12th digit positions — the address-modifiability indicators. The NGO instruction must be executed exactly once, while the OGO instruction is repeated cyclicly until the operations have been performed on all numbers of the group. The second address of the OGO instruction should give the address of the initial instruction.

The last group of "Kiev" machine operations contains six operations with periphery equipment. Three of these are operations with magnetic drum: preparation for call to magnetic drum MBP (octal operation code 25), operative instruction for writing words MBZ (code 23), and operative instruction for reading words MBCh (code 24). The three other operations perform data loading and readout: load numbers V1 (code 20), load instructions V2 (code 21), and readout Pech (code 22).

All operations with peripheral equipment are group operations. They disrupt the content of the address-modification register.

The MBP operation is a preparatory operation for exchange of words with magnetic drum. The content of address II of instruction a_2 (not $'a_2$!) indicates the number of the magnetic drum and the number of the channel with which words will be exchanged upon execution of the following operation (MBZ or MBCh). The content of address III of instruction

a_3 (and not 'a_3!) specifies the number of the number in the selected channel with which the exchange of words begins. The content of address I of the MBP instruction may take on only two values: zero, if the instruction prepares for writing onto magnetic drum, and one if it prepares to read from it. After execution of the MBP operation, control jumps to the next instruction in sequence, which must be MBZ or MBCh. It is necessary to note a special feature of the formation of address II in the MBP instruction. Two binary digits are sufficient to code three drum numbers. Actually, in order to preserve the convenience of writing instructions in the octal system, the third octal digit position of address II is set aside for coding the drum number.

The MBZ operation accomplishes writing of codes from OZU or PZU onto magnetic drum. The numbers of the locations with address from a_1 to a_2, inclusive, are written onto the magnetic-drum channel that has been prepared by the preceding MBP operation. After execution of the MBZ operation, control is transferred to the instruction associated with address a_3.

The MBCh operation represents readout from magnetic drum of a code block, and writing of the block into OZU. The codes for the drum called for by the preceding MBP operation are written into OZU locations with addresses from a_1 to a_2, inclusive. After execution of MBZ or MBCh operations, control jumps to address a_3.

Execution of both operations (MBZ and MBCh) is accompanied by automatic switching to the next channel in order of the same magnetic drum when the numbers being transferred will not fit into a single channel. Switching does not occur from the last (24th) channel of each drum.

The load numbers operation means that a block of numbers is introduced from punched tape or punched cards into OZU locations with addresses from a_1 to a_2, inclusive. Here the numbers are automatically converted from the decimal system into binary. With loading from punched cards, the zone number is specified in address III of the instruction; the first of the numbers to be loaded is determined by the beginning of the given zone, and the last number by the quantity of numbers to be loaded.

The load instruction operation differs from the load number instruction only in that there is no conversion from decimal to binary.

The readout operation means that a block of words is read out from OZU locations with addresses from a_1 to a_2, inclusive. Following execution of the operation, control is transferred to the instruction with address a_3. By means of a special switch, the machine can be made to either print out or punch out the codes. In like manner, an additional switch may be used to select the type of printout: decimal (four numbers) or octal (four instructions). We recall that for decimal printout, the number block being printed out must first be converted with the aid of a special routine to the binary-coded decimal number system.

The computer may be initially started up in two different ways. First, following preliminary establishment of the computer in the initial configuration, we can use the control-console switches to set up the load instruction (four numbers or instructions). By starting the machine to run in the console-instruction read mode, we accomplish loading of the initial information into the computer. Following this, it is necessary to set up at the control console the address of the first program instruction, and to start the computer for solution of the program.

With the second method, the initial load instruction is set up on one of the switch registers of the PZU, while the address of this instruction is set up at the control console. After the button is pushed, loading begins, and then the computer automatically goes to execution of the program that has been loaded.

REPRESENTATION OF CODES IN THE COMPUTER

Representation of Numbers

The "Kiev" computer operates with binary numbers, represented in a fixed-point system. Of the cell digit positions, 40 are used for the number mantissa, and one (the 41st) for the sign of the number (see Fig. 1): 0 corresponds to a plus sign, and 1 to a minus sign. Thus the computer operates with numbers in the range from $-1 + 2^{-40}$ to $1 - 2^{-40}$. Numbers smaller in absolute value than 2^{-40} are written into a location as a machine zero. In this case, $-0 < +0$.

The following system has been adopted to shorten the representation of binary numbers. The binary number is divided into groups, of which the first contains two binary digits, including the sign bit, while the rest consists of three bits each (a total of 14 groups). Next the groups are coded by means of the corresponding octal digits. Thus, the negative number with code 1·1 110 010 011 010 100 001... is written as 3 623 241 ..., while a positive number with code 0.1 110 010... is written as 1 62... .

To convert numbers from the decimal system into the representation used for the machine, the following method is convenient: an n-digit number is multiplied by 2: the (n + 1)-th digit of a positive number is the first number in the desired representation. The (n +)-th digit of a negative number, increased by two, is the first digit of the desired presentation. The remaining n digits of the result (remainder) are multiplied by 8. The (n + 1)-th digit is the second number of the desired representation. All successive remainders are similarly multiplied by 8. The process is continued until a 0 appears in the remainder, or until the required accuracy is obtained. As an example, conversion of the number 0.625

```
0  625 × 2
1  250 × 8
2  000
```

yields the result 12 0000 0000 0000. Conversion of the number —0.625 yields the representation 32 0000 0000 0000.

The binary-number notation used results in a uniform representation for numbers and instructions, and this is the essence of its convenience.

Decimal numbers are loaded into the computer in binary-coded decimal form. In this case, four binary digits are used to represent one decimal digit. Thus, a maximum of ten decimal digits can be loaded into one machine location or printed out.

<u>Representation of Commands (Instructions)</u>

In the computer, numbers and instructions can be coded in exactly the same ZU locations. A word of the "Kiev" machine contains four groups of digits: operation code (address 0) and three addresses (I, II, III). The operation code, contained in the zero address determines the interpretation of the addresses. The five highest-order digits are set aside for the operation code, and 12 digits for each of the addresses. The highest-order digit position in each of the addresses is used as an indicator of address modifiability (group-operation flag).

When we work on coding sheets in octal numbers, the address is coded together with the modifiability indicator. Thus, in the octal system, the modifiability indicator of an address will be the relationship a > 3777. To show that an address is modifiable, we must add the number 4000 to it.

The computer will not execute an instruction whose code consists of zeros. After the computer has idled for a working cycle, control jumps to the instruction with the next number.

Manuscript Page No.	[Footnotes]
7	All addresses and address-modification constants are given in the octal number system here and in the ensuing discussion.
9	See the footnote to page 7.
11	Here and in the ensuing discussion, we use $'a_1$ to mean the content of the memory location having address a_1.

Chapter 2

PROBLEM OF VALIDATING THE CHOICE OF FUNDAMENTAL CHARACTERISTICS

UNIVERSAL ECONOMIC CRITERION FOR EFFICIENCY OF AUTOMATIC DIGITAL COMPUTER

When starting work on the creation of the "Kiev" electronic computer, the development group had no fundamental criteria available for determination of the degree of efficiency of the machine to be built. Thus selection of basic parameters for the "Kiev" machine was accomplished by intuition and generalization of experience accumulated at that time. It was only at the second stage of the work (1958) that V.M. Glushkov [2] proposed a universal economic criterion making it possible to compare the efficiency of computers of extremely different types. This criterion came to be called the <u>cost of effective speed criterion</u>.

The cost of effective speed criterion is based on two notions. The first of these consists in an economics approach to evaluation of computer productivity, which is easily understood on the basis of the following example. Assume that computers A and B have been developed to solve a certain class of problem; computer A is twice as fast as computer B. Can we say that computer A is more efficient than computer B? Obviously not. It might turn out that computer A was far more expensive than computer B, requires greater operating expenses, etc., etc. It therefore might turn out that the total expenditure for solution of some class of problems on computer A is, for example, six times the analogous expenditures for computer B. Thus, for exactly the same sum, we might have six type B computers in place of one A computer; these six computers, in exactly the same time, might perform three times the work that computer A can do. Thus, from the viewpoint of economic planning, the slower computer might prove to be more efficient than the higher-speed machine. It is precisely for this reason that it proves desirable in the solution of many problems to employ desk calculating machines, whose operating speed is less by factors of tens of thousands than that of electronic digital computers. The determining factor is not speed in itself, but the value of the speed.

Of course we must remember that allowance for economic factors, at least in the form just discussed, cannot be made the overriding factor in every case. Actually it may turn out, for example, that it is so important to solve a problem in the least possible time that it is necessary to attempt to increase speed as much as possible, without reckoning with expense. This might also be true with respect to certain other factors (machine size, power requirements, etc.). The economic criterion may be modified, however, so as to cover these exceptional cases as well.

The second idea underlying the criterion of cost of effective speed consists in the notion that the rate at which the computer executes various elementary operations (addition, multiplication, etc.) does not determine computer speed, even for program-controlled automatic computers. In fact, it is well known that computer operating speed may be substantially increased by intelligent selection of control operations and program registers (so-called machine logic). Such computer parameters as size of high-speed memory, rate of transfer from external memory, loading and readout speed may have a substantial influence on the speed with which the machine solves problems. Finally, we must not forget that the true speed of a machine is reduced sharply if its reliability is low, since in this case it is necessary to expend much time not only to eliminate machine malfunctions, but also to perform multiple check calculations, duplication of modifications, etc.

In talking of the value of speed, we have in mind not the ordinary nominal speed, i.e., the rate at which the computer executes the fundamental arithmetic operations, as shown in the specifications for the computer, but the effective speed; this concept takes into account the reduction in speed owing to operations involving peripheral equipment, duplication of calculations to obtain adequate reliability of the result, and other similar factors.

Let us now turn to a more exact formulation of the criterion of cost of effective speed. Let there be specified a certain class K of problems which are to be solved by computer A. We shall assume that we have been given a statistic for class A such that for each problem k in K, we are given the relative frequency f(k) with which this problem will be encountered in solving problems of class K in the course of an infinitely long period of time.

We further specify a certain typical set O of operations for a general-purpose digital computer and assume that for each problem k in K, we know the number $\varphi(k)$ of operations in O needed to solve the problem. Then any work toward solution of problems in class K performed by computer A can be characterized by the number of operations of the typical set needed to perform the same work. We now specify a certain fairly long time interval T (it is desirable to take a time of the order of 10 years for T) and compute the amount of work that machine A will perform over this period, with allowance for time lost for preventive maintenance, duplicate calculations, operations with peripheral equipment, etc. Let this volume of work be represented by the number $N = N(T)$ of operations from the typical set. Dividing N by the number of seconds in

T, we obtain a quantity n_e, which we call the effective speed of the computer.

We also compute the total expenditures Z(T) for the manufacture, amortization, and operation of the computer over the time T. The total expenses will contain, in particular, expenditures for operating personnel and electric power. It is not desirable, however, to include expenditures for programming of problems in Z(T), since these expenses depend not so much on machine design as on available programming prerequisites (primarily on the size of the standard-routine library). Now the cost of effective speed q is computed from the formula

$$q = \frac{Z(T)}{N(T)} = \frac{Z(T)}{Tn_e}. \qquad (1)$$

We can understand without difficulty that one of the fundamental problems facing the computer designer is to reduce the cost of effective speed as much as possible.

In order to exclude the effect of indeterminacy in the choice of the time interval T, we could go to the limit in Formula (1)

$$q = \lim_{T\to\infty} \frac{Z(T)}{N(T)}.$$

In practice, however, this limit can be attained with adequate adequacy for values of T of the order of 10 years or more. In addition, we should not forget that when $T \to \infty$, we eliminate allowance for obsolescence and the need to replace old equipment. It is thus obviously more sensitible to select a value of T equal to 10 years.

We note that while the quantity Z(T) can be calculated without two much difficulty, N(T) depends heavily on the statistic for the class K of problems to be solved which, as a rule, will not be known. Even where we know the statistic for class K, however, it is very difficult to compute the value of N(T) exactly. In many cases, therefore, the best method involves a rough calculation of the coefficient p, equal to the ratio of the effective computer speed to its nominal speed. To compute the coefficient p, we may make use of the approximate formula

$$p = p_0 p_{\text{п}} p_{\text{м}} p_{\text{в}} p_{\text{к}}. \qquad (2)$$

Here p_0 is the ratio of the rated computer speed, expressed in operations of the typical set, to the rated speed, expressed in actual machine operations. The coefficient p_0 is determined by the machine operation set and by the statistic for the problems to be solved. It may be greater than or less than unity. We use p_p to represent the computer time utilization factor (equal to the amount of time spent in solving problems); it will always be less than unity. As a rule, p_p will have a value in the range 0.7-0.9.

We use p_m and p_v to represent coefficients indicating the decrease in speed owing to operations with magnetic drum and magnetic tape (p_m) and owing to operations with input-output devices (p_v). These coefficients are determined by the capacities of working memory, rate of exchange of information with peripheral equipment, and the statistic of the class of problems solved. They will always have values less than unity.

Finally, p_k is coefficient indicating the reduction in machine speed owing to program control, duplicate calculations, and other methods for checking correctness of results yielded by solution of a problem. This coefficient will also be less than unity at all times. Its value is primarily determined by the dynamic reliability of the computer (malfunction probability), and also depends on the statistic for the class of problems solved.

The "Kiev" computer must solve fairly complicated problems, so that no large error will result if we take a value of 0.1 for the coefficient p in this case. Obviously, the same value of the coefficient p may be taken for other computers having working-memory capacity of the order of 1000-4000 numbers.

We use n to represent the nominal machine speed (average number of operations per second performed by the computer when working with high-speed memory); since there will be about $3 \cdot 10^8$ sec in 10 years, we obtain the following formula for the cost of effective speed

$$q = \frac{Z(T)}{3 \cdot 10^8 pn}. \qquad (3)$$

We note that even when Z(T) is expressed in kopecks, Q will be considerably less than unity. It is therefore desirable to express q in kopecks per thousand operations or in rubles per million operations.

For the "Kiev" machine with T equal to 10 years, the value of Z(T) will equal roughly 1.2 million rubles (0.4 million rubles initial cost, 0.5 million rubles amortization, 0.1 million rubles electric power, 0.2 million rubles operating-personnel salaries, including keypunching). Thus the cost of effective speed for the "Kiev" machine is represented by the approximate value

$$\frac{1{,}2 \cdot 10^6 \cdot 100}{3 \cdot 10^8 \cdot 0{,}1 \cdot 10\,000} = 0{,}0004 \text{ kopeck/operation}$$

or 0.4 kopecks per thousand operations, or 4 rubles per million operations.

For the "Ural" computer, in the early models, the quantity Z(T) was reduced to roughly 0.4 million rubles. The effective speed (the value pn) for the "Ural" was barely more than 15 operations/sec. Thus the cost of effective speed for the "Ural" computer is represented by a quantity of the order

$$\frac{4 \cdot 10^5 \cdot 100}{3 \cdot 10^8 \cdot 15} = 0{,}01 \text{ kopeck/operation,}$$

or 10 kopecks per thousand operations or 100 rubles per million operations.

Thus, under the given conditions, the "Kiev" machine is a computer that is roughly 25 times as efficient as the "Ural." We should remember, naturally, that such an estimate is valid only provided the same class of problems is solved on both machines, and these problems are fairly complex, characterized by a fairly large ratio of the amount of internal machine time to input-output time. If this stipulation is not met, the picture changes sharply, since in the solution of short problems with much input data, the advantage may prove to rest with the "Ural" computer.

To conclude, we note that allowance for all factors affecting the effective speed of a computer frequently proves to be extremely difficult. It is very hard, for example, to compute the value of the coefficient p_k with any great accuracy. Thus in practical work it is desirable to use simplified computations based on the fact that one or several of the coefficients p_0, p_p, p_m, p_v, p_k is assumed to equal unity. Such a simplification turns out to be quite acceptable where the criterion of cost of effective speed is used to determine the optimum value of a parameter that has little influence on the coefficients that have been set equal to unity. Moreover, instead of minimizing the cost of effective speed, we could attempt to maximize the effective speed itself without reference to cost.

It is also necessary to note that with the modern state of the theory of electronic computers, as a rule, an exact estimate of effective speed proves possible only on an a posteriori basis, since there exist no satisfactory methods for the a priori estimation of machine dynamic reliability (malfunction probability). In addition, the statistics characterizing the class of problems to be solved will not be known in advance, as a rule.

ADDRESS FORMAT AND WORD LENGTH

The question of the desirable number of addresses to use in a general-purpose electronic digital computer may be solved on the basis of the cost of effective speed criterion. Here the statistic characterizing the class of problems to be solved will play a fundamental role. Since for the "Kiev" machine, no such statistic was available, the problem of its address format could only have been solved on the basis of elementary qualitative considerations. It is easy to understand that the cost of effective speed criterion depends on the efficiency with which the equipment is utilized and, in particular, on the presence or absence of superfluous information transfers within the computer. From this viewpoint, in machines using parallel word transfer, the single-address instruction system may prove to be most efficient.

The loss of efficiency in equipment utilization that appears when multiaddress instructions are employed is easily demonstrated, using the example of obtaining the sum of three or more terms. Actually, when the sum $x + y + z$ is computed, where a three-address instruction system is used, the result of the first addition $x + y$ is placed into some storage location only so that the next instruction can again extract it for transfer to the adder.

When single-address instructions are used, however, we usually must face the problem of the difference in number-word and instruction-word lengths, which leads either to a decrease in efficiency of machine storage utilization or to complication of the wiring. Taking these additional considerations into account, we may assume that a three-address instruction system is roughly equivalent to a single-address system. The decisive factor, dictating the choice of the three-address system for the "Kiev" machine was experience gained by the staff of the Computer Center of the AN UkSSR in working with three-address machines.

The statistic characterizing the class of problems to be solved proves to have a decisive influence on the selection of number system and number representation (fixed or floating point). Of the two number systems — decimal and binary — common in world-wide mathematical machine design, the binary system provides more efficient use of equipment from the viewpoint of internal machine operations. The decimal system offers definite advantages in information readin and readout, however, since no additional operations are needed to convert numbers from one system to the other.

Thus, the choice of a given number system is dictated primarily by the nature of the problems to be solved by the computer. For problems involving relatively small volumes of data readin and readout operations as compared with the number of operations performed on the information itself within the machine, it is better to use the binary number system. This is the situation for solution of the majority of complex scientific problems, for which the "Kiev" computer was basically designed. Thus we selected the binary number system for its internal system.

The choice of a number-representation system (floating- or fixed-point system) is dictated primarily by the nature of the problems to be solved by the computer. Where the machine is to solve chiefly non-arithmetic problems and arithmetic problems for which the number exponents are roughly of the same order, the fixed-point system provides a lower lost of effective speed, and as a consequence is preferable. For problems in which it is frequently necessary to change the scales of numbers in the course of the calculations, the floating-point representation offers great advantages and great convenience in programming of problems.

It should be noted, however, that use of a floating-point system frequently does not eliminate the necessity of estimating in advance the orders of magnitude for numbers appearing in the course of problem solution. If this is not done, we may encounter a continuous loss of accuracy when the difference between two numbers of similar magnitude is computed. Especial danger is presented by the fact that when the floating-point system is used, such a loss in accuracy may occur undetected, since the mantissa will always retain exactly the same number of significant figures. For example, when the formula $u = (x - y)z$ is used in computations for $x = 0.1278$, $y = 0.1273$, $z = 0.9812$, the result $u = 10^{-3} \cdot 0.4906$ will give a completely false impression of the number of valid digits, while where the fixed-point system is used, the result $u = 0.0005$ will at once indicate that a loss of accuracy has occurred.

Finally, let us consider the last question associated with the representation of numbers in the computer — the choice of word length. As a point of fact, for the "Kiev" machine, this problem was determined by the condition requiring number words and instruction words to be of the same length; satisfaction of this condition offers important advantages from the viewpoint of efficient use of memory. The length of an instruction word was determined by the necessity of coding 29 operations and three addresses. Since memory size (allowing for a reserve) was set at $2048 = 2^{11}$ numbers, while one bit in each address was set aside for the modification indicator, the instruction word length is $5 + 3(11 + 1) = 41$ binary digits. The length of a number word (with the sign bit) was taken to be the same.

To solve the problem of whether the number-word length is adequate, we must consider the nature of the problems to be solved on the computer. Two characteristics are of basic importance here: the maximum result accuracy required and the maximum length of a sequence of consecutive calculations in which rounding error will accumulate.

Let us study the effect of the last factor in more detail. For simplicity we shall assume that for the execution of each individual operation in the sequence of operations the rounding errors are independent random quantities, while the total rounding error for each sequence will equal the sum of the rounding errors for each individual elementary stage of calculation. The rounding error for an arbitrary (ith) stage of calculations will be distributed uniformly between minus one-half and plus one-half of a one in the lowest significant digit.

The total mean-square rounding error σ_n (in terms of the lowest-order significant digit) for n stages will then be determined by the formula

$$\sigma_n = \sqrt{\int_{-\frac{1}{2}}^{\frac{1}{2}} \int_{-\frac{1}{2}}^{\frac{1}{2}} \cdots \int_{-\frac{1}{2}}^{\frac{1}{2}} (x_1 + x_2 + \ldots + x_n)^2 \, dx_1 dx_2 \ldots dx_n}. \quad (4)$$

Since the integrals of the products $x_i x_j$ vanish, Formula (4) will take the form

$$\sigma_n = \sqrt{n \int_{-\frac{1}{2}}^{\frac{1}{2}} x^2 dx} = \sqrt{\frac{n}{12}}. \quad (5)$$

It is well known that the distribution of sums of independent random quantities of the type under consideration are described by the normal law with sufficient accuracy for fairly small values of n (of the order of several tens). Thus with probability 0.9, the total rounding error α_n will not exceed twice the mean-square error

$$|\alpha_n| < 2\sqrt{\frac{n}{12}} = \sqrt{\frac{n}{3}}. \quad (6)$$

We shall now assume that we are using a number system with base a and that we have m additional digits to compensate for the rounding errors. In this case, a one in the lowest significant digit in the last analysis will amount to, obviously, a^m units in the lowest-order significant digit retained in the course of calculations. As a consequence, to retain an accuracy of one unit in the lowest-order significant digit in the final result with probability 0.9, we must ensure that the inequality

$$2\sigma = \sqrt{\frac{n}{3}} < a^m. \qquad (7)$$

is satisfied. Since a will always be greater than unity, then

$$m > \frac{1}{2}\log_a \frac{n}{3}. \qquad (8)$$

Formula (8) and its variant form

$$n < 3a^{2m} \qquad (9)$$

make it possible to estimate the required number of additional digits m needed to compensate for rounding errors (with an accuracy of 0.9) in sequences of calculations of length n.

In particular, letting $m = 1$ and $a = 10$ we find that the familiar rule of A.N. Krylov on the need to retain one superfluous decimal digit in intermediate calculations yields satisfactory results when the length of the calculation sequence does not exceed $3 \cdot 10^2 = 300$. This case usually occurs in manual calculations, but when automatic digital computers are used, the maximum calculation-sequence length rises sharply. We shall assume that it is necessary to obtain a calculation accuracy for a sequence length of the order of three million. We then find from Formula (8) that to compensate for rounding errors we must keep an additional $\frac{1}{2}\log 10^6 = 3$ decimal digits during the calculations, or $\frac{1}{2}\log_2 10^6 \approx 10$ binary digits.

As we know, in the "Kiev" computer there are 40 bits available for coding words. Discarding the 10 bits used to compensate for rounding errors, we obtain 30 binary (or about 10 decimal) digits, which gives us an idea of the possible greatest accuracy of results. Actually, result accuracy is reduced owing to incomplete utilization of the available number of digits in fixed-point calculations. An exact determination of the magnitude of this decrease is possible only for each concrete case, however. We note, moreover, that the decrease in accuracy due to incomplete utilization of the digits of a number may be eliminated, in great measure, by careful scaling of the numbers.

STORAGE CAPACITY

As any programmer who has handled complicated problems for computer is well aware of the inconvenience of the fairly small OZU capacity in present-day computers. Thus, all other conditions being equal, it is desirable to attempt to make the OZU capacity as large as possible. We must not forget, however, that an increase in OZU volume will lead to more complicated circuitry, increasing transient time in control circuits and reducing operating speed. Thus if we are to be guided only by the criterion of maximum effective speed, it is necessary to solve the problem of finding optimum OZU volume.

Let us look at one possible method for solving this problem. We must at once note that any method inevitably must deal with large numbers of unknown factors, which at the present time can only be estimated roughly. Thus the value of the method discussed should not be overestimated: it can only give a very rough estimate for optimum OZU volume.

We agree that we shall consider only with an OZU of a particular type, namely, a ferrite-core array. If we let x be the capacity (number of locations) of such an array, it is natural to assume that the access time for a single location is represented by the formula

$$t_{\text{обр}} = a + b\sqrt{x}, \qquad (10)$$

where a is a constant representing the time spent in ferrite magnetic reversal; b is a constant characterizing the time lost due to transients in control circuits.

We must also specify a certain function f(x) representing the average number of operations performed in an OZU with a capacity of _x_ locations without recourse to external storage units VZU. Naturally, this function will depend on the class of problems to be solved, and thus cannot be determined once and for all. It is apparent that the relationship

$$f(x) = x^2. \qquad (11)$$

will reflect the essence of the matter fairly well for a general-purpose machine.

It is also necessary to specify at least two parameters characterizing the VZU, namely: the average waiting time _c_ and the time required to pick out and write a single word _d_. In addition, it is also desirable to introduce the function φ(x) equal to the mean relative proportion of problems that may be solved by the use of an OZU with a capacity of _x_ locations without recourse to the VZU.

As for the actual call to the VZU, to keep the discussion simple we shall assume that this call consists in complete regeneration of the OZU contents with preliminary readout of the old contents into one of the VZU.* For simplicity, we shall also assume that the call occurs only for one (initial) point in a VZU for writing, and also to a single point in reading.

If we now let τ be the average time required to execute one operation in the arithmetic unit of the computer, while _p_ is the number of addresses in an instruction, the mean time _t_ required to execute one instruction in the computer (allowing for a call to the VZU) will evidently be represented by the formula

$$t = \{[(p+1)(a+b\sqrt{x})+\tau]f(x)+(1-\varphi(x))(2c+2dx)\}:f(x) =$$
$$=(p+1)(a+b\sqrt{x})+\tau+2\frac{1-\varphi(x)}{f(x)}(c+dx).$$

Discarding the constants, we face the problem of minimizing the expression

$$v = (p+1) b\sqrt{x} + 2\frac{1-\varphi'(x)}{f(x)}(c+dx). \qquad (12)$$

Let us solve this problem for a three-address machine on the assumption that

$$f(x) = x^2, \ \varphi(x) = 0,$$

so that

$$v = 4b\sqrt{x} + \frac{2c}{x^2} + \frac{2d}{x}.$$

After differentiation with respect to x we set the resulting expression equal to zero, arriving at the equation

$$bx^2\sqrt{x} - dx \ - 2c = 0. \qquad (13)$$

Expressing the time in microseconds and assuming that the transient time in the control circuits of the array is 1 μsec per thousand numbers, we find from Formula (10) that b ≈ 1/30. It is not difficult to see that Eq. (13) will give an optimum high-speed memory volume that is smaller the smaller the coefficients d and c, i.e., the faster the VZU operate.

In order to reduce OZU volume in the "Kiev" machine, we employed magnetic-drum external storage, which provides more rapid access than magnetic-tape memory. There are three magnetic drums with a total capacity of 9000 words. The drums turn at a rate of 15,000 rpm, corresponding to an average waiting time of 20 μsec. If we also take into account the relay-switching time, we obtain an average waiting time of about 25 msec, or 25,000 μsec. The access time for a single word is 60 μsec. Inserting this data into Eq. (13), we arrive at the equation

$$\frac{1}{30}x^2\sqrt{x} - 60 - 50\,000 = 0. \qquad (14)$$

This equation has a single positive root equal roughly to 340. We should note, however, that the calculation did not allow for the necessity of magnetic-drum channel switching during larger transfers or the possibility of small transfers from different points in the same channel. If we take these possibilities into account, we would have to increase the optimum OZU capacity. Finally, if we discard even these factors, which are difficult to compute, and assume that the function fx equals not x^2, but $x^{\frac{3}{2}}$, we have in place of Eq. (4) the equation

$$2bx^3 - dx - 3c = 0, \tag{15}$$

which has a positive root equal roughly to 1600 if we use the same values for the parameters b, c, and d as before.

Thus, the high-speed memory capacity of 1024 numbers used in the "Kiev" machine is at least of the same order as the calculated optimum OZU volume.

SET OF OPERATIONS

When the designer specifies the set of operations for a digital computer, being guided by the special features of the problems to be solved by the machine, he must ensure the maximum possible reduction in effective-speed cost while simultaneously providing adequate programming convenience.

Since a priori estimates for the cost of effective speed criterion turn out, as a rule, to be impossible, we must resort to other types of simplication. One of the most common simplifications consists in varying the set of computer operations within reasonable limits, so as to increase as far as possible the value of the coefficient p_0 in Formula (2). In other words, the set of operations is so selected that in solving the problems most typical for the given computer, the machine can make the most efficient possible use of these operations. It is usual to take as a standard the operations in some predetermined standard set of operations O.

For a general-purpose digital computer, moreover, it is necessary to ensure the universality of its operation set. Since there has so far been considerable confusion as to the nature of the concept "general-purpose computer" we shall first of all define it more precisely. To do this, we must become acquainted, in general terms, with the concept of the Turing machine, proposed by the English mathematician Turing in 1936.

The Turing machine is an automatic device Q capable of assuming some finite number of different states q_1, q_2, ..., q_n and of moving to the right and to the left along a tape extending to infinity in both directions and divided into separate cells. One of the finite set of symbols s_1, s_2, ..., s_n may be written into each cell of the tape. The automatic device Q is provided with a reader capable of perceiving these symbols and a writing element capable of writing any of these symbols into a scanned cell.

The machine Q is a strictly determined device: its behavior at each instance is determined by the pair of symbols (q_i, m_j), i.e., by the state of the machine q_i at the given instant and by the symbol m_j being read. The machine is capable of scanning just one cell of the tape at each instant). For each such pair (q_i, m_j) the following operations of the machine Q are defined: it writes a new symbol m_{j_i} into the scanned tape cell (replacing the previous symbol m_j), moves along the tape to the right or to the left by one cell, and goes over to the new state q_{ij}. After all of these operations have been performed, the process recommences. Letting d_1 represent the shift to the right and d_2 a shift to the left, we note that the behavior of the Turing machine is determined completely by the <u>initial content of the tape</u> (initial information) and by the <u>program</u> for the machine, which consists of a finite number of ordered quintuplets $q_i m_j m_{j_i} d_{ij} q_{ij}$, where the q_i and m_i independently run through the set of all machine states and, correspondingly, the set of all symbols.

In mathematical logic, there is a postulate stating that any transformation of alphanumeric information performed in accordance with a finite system of exact rules (of any nature) may be accomplished by some Turing machine constructed specially for this purpose.

Let us now have an automatic data processer with program control (for example, an automatic digital computer). It is natural to call such a processer universal if by changing the program by which it operates we can simulate on it the operation of any Turing machine. In virtue of what we have just said, such a general-purpose processer may be used to transform alphanumeric information in accordance with any system of exactly formulated rules (algorithms).

It turns out that it is sufficient to introduce into the operation set of the automatic digital computer a very small number of fundamental operations in order to convert it to a universal information processer or, as we ordinarily say, a general-purpose computer. These are operations with peripheral equipment (including external storage), as well as conditional-jump and readdressing (addition of instructions) operations, and transfers from one storage location into any other storage location (this last operation is usually realized in three-address machines as a special case of the addition operation, namely addition to zero).

We should at once note that, strictly speaking, for a machine to be universal it would have to have an infinite memory capacity. At the same time, it is well known that high-speed storage volume in modern computers is not only not infinite, but actually extremely limited. The situation is saved by the fact that external storage with present-day computers is nearly unlimited, and can be used at any time to make up for the limited amount of high-speed storage.

We agree to call the operation set of an automatic digital computer algorithmically complete if it ensures simulation of any Turing machine. We say that a computer A is universal if, first, its operation set is algorithmically complete and, second, it has a high-speed storage capacity adequate to contain the universal simulation program that enables our machine to simulate the operation of any Turing machine as soon as the program for the latter is written into the high-speed or external memory of computer A.

For simplicity, we shall assume that our machine has three external storage units (VZU-1, VZU-2, VZU-3), which contain, respectively, the operating program for the simulated Turing machine (in the form of a set of quintuplets of symbols similar to those discussed above), the state of this machine at the given time, and the contents of its tape.

Let us now discribe the universal simulating program in general terms. The basic difficulty lies in the fact that both the number of states of the simulated Turing machine and the number of different symbols on its tape may be so large that they cannot be represented by words of the length used for the numbers with which the simulating computer A works. In this case, each state q_i of the Turing machine will be represented by a series of numbers. It is desirable to store these numbers in successive even locations of the VZU, while the odd locations are used for auxiliary indices indicating the basic numbers by some method or another. Morover, it is necessary to use different numbers to represent the states (q_i), symbols (m_j), and nature of tape motion (d_k); special numbers must also be used to represent the beginning and end of the sequences of numbers representing q_i and m_j.

Then the universal simulation program will contain the following basic segments. The first segment, controlling the sequential selection (with interval 2) of numbers from VZU-1 will compare the numbers and in case of a disagreement will jump to the beginning of VZU-2 and to the beginning of the next sequence for q_i in VZU-1. The second program segment begins to run after complete agreement has been established between the sequence q̲ contained in VZU-2 and one of the sequences q_i in VZU-1. It selects and compares sequences of numbers from VZU-3 (the scanned symbol m̲) and VZU-1 (the symbol m_j of the Turing-machine program following the selected state q_i).

When the sequences are not the same, control again jumps to the first segment, while where they are the same, the next segments begin to run; they write into the VZU-3 the new symbol m_{ji} from the selected quaduplet of symbols from the program in VZU-1, write the new state q_{i_j} (from the same quadruplet) into VZU-2, and readdress the instructions pertaining to VZU-3 for the beginning of the following or preceding sequence.

It is easy to see that all of these segments may be realized in the following instructions of the "Kiev" computer: MBP, MBZ, MBCh, addition of instructions, and exact comparison (while where punched tape is used for one of the VZU, the instructions "load instructions" and "readout" as well). The entire universal simulation program will consist of dozens of instructions and, in any case, can be stored in the OZU of the "Kiev" computer.

Thus, according to the definition adopted above, the "Kiev" may be considered to be a universal computer. The algorithmic completeness of its operation system is ensured by the operations with peripheral equipment (including external storage), addition of instructions, and exact comparison. Where part of the program for the simulated Turing machine is stored in OZU, we must add to these operations the operation of number transfer from one location to any other location (in the "Kiev" computer, such an operation may be accomplished either with the aid of the F operation, or by addition with zero). Where the universal simulation program is stored in the PZU, we must add to the operations ensuring algorithmic completeness the F operation as well. All remaining operations of the "Kiev" computer are superfluous from the strictly algorithmic viewpoint. There purpose is mainly to simplify programming and increase the effective speed of the machine.

We note that it is not absolutely necessary to include the exact-comparison operation in the fundamental operations ensuring algorithmic completeness of the operation set; this operation may be replaced, for example, by the ordinary comparison operation (Sr. 1) which is then used to compare the given pair of numbers twice (in regular and reverse order).

Further justification for the choice of operation system for the "Kiev" computer can be given only in the form of illustrations demonstrating the increase in effective computer speed owing to the presence of the given operation in frequently encountered standard routines. This sort of analysis went on throughout the entire period of work on the "Kiev" machine. As a result, numerous changes were made in the operation set in order to improve it, so that the final set, described below, was adopted only in 1959. Unfortunately, during this improvement process, it was necessary to reckon with the fact that the fundamental computer units had already been installed, so that we could only use those improvements not involving significant circuit modification. For this reason, the final operation set does not completely satisfy the mathematicians working on the machine. Most of the current suggestions for improving the operation set will be carried out at the next modernization of the computer.

Example 1. Computation of the scalar product of two vectors

$$a_1b_1 + a_2b_2 + \dots + a_nb_n.$$

Four instructions are sufficient to solve this problem on the "Kiev" machine: NGO, ×, +, and OGO. Here the NGO instructikn will be executed just once, and the remaining instructions n times each. Thus, the total number of operations needed to compute the scalar product of two n-dimensional vectors will be 3n + 1 for the "Kiev" machine. In the absence of the NGO and OGO operations, 4n operations would be required (in addition to multiplication and addition in each computation cycle, it would also be necessary to use the instruction-addition operation and comparisons).

Example 2. We are required to execute a certain set of operations O on a group of numbers whose first address is not known in advance and which is obtained only in the course of problem solution. With the aid of the operation set of the "Kiev" computer, such a problem is easily solved owing to the possibility of setting the address-modification register on the basis of the specifier (it is possible that it will also be necessary to employ a logical shift if the first address of the group does not appear in the digit positions that are transferred into the modification register by the F operation). In the absence of the F operation, solution of this problem would require formation of the NGO instruction with the aid of the instruction-addition and shift operations, while in the absence of the modification register, it would be necessary to readdress all the instructions of set O during each loop.

Manuscript Page No.	[Footnotes]
36	Another approach to the nature of interchange of codes between external memory and OZU has been developed in [1].

Chapter 3

ADDRESS PROGRAMMING AND THE "KIEV" ELECTRONIC COMPUTER

THE ADDRESS LANGUAGE

In the rest of the discussion in this book, we shall use an algorithm language introduced for formal description of algorithms, and called the address language.

The address language is a universal algorithm language very similar, on the one hand, to the generally adopted language of formulas and, on the other hand, to machine languages on the whole; as a result, algorithms in this language may be considered to be programs for general-purpose machines. The similarity of the address language and machine-code languages is explained by the fact that it reflects the fundamental algorithmic principles realized in present-day universal computers: the principle of program control and the addressing principle.

The translation of algorithms from the general formal language to the language of a concrete machine may be automated (using the same computer) by creating appropriate translator programs PP.

The idea of creating such a language appeared as a result of the work of the seminar in the theory of algorithms (1957-1958), which worked under the guidance of V.M. Glushkov, L.A. Kaluzhnin, V.S. Korolyuk, and Ye.L. Yushchenko.

The work on creation of the language itself was first carried out jointly by V.S. Korolyuk and Ye.L. Yushchnko [10]. Later Ye.L. Yushchenko [15] made substantial improvements in the language, some of which were discussed in the joint publication by B.V. Gnedenko, V.S. Korolyuk, and Ye.L. Yushchenko [5]. In this book, we discussed the language in the form in which it was proposed by Ye.L. Yushchenko at the end of 1959, with slight modifications.

The work toward creation of the address language had its influence on the selection of operations for the "Kiev" machine. Here we should mention first of all F operation, included at the suggestion of Ye.L. Yushchenko, and, to some extent, the NGO and OGO operations, which were developed with the participation of V.S. Korolyuk [9], in addition to Ye.L. Yushchenko.

The hardware realization of all these operations was carried out under the guidance of L.N. Dashevskiy and Ye.A. Shkabara. Ye.L. Yushchenko further suggested that conditional control transfers carried out on the basis of the address of higher rank and, finally, that relative control transfers be used, making it possible to store subroutines in arbitrary memory areas without changing the way in which they are written. The hardware implementation of this last proposal will be realized in the near future.

The address language contains all symbols used in mathematics to represent variables, vectors, functions, sets, and symbols for mathematical operations. The system of concepts (and symbols) of the address language contains special concepts (and corresponding symbols), however, owing to which it is better suited to the description of algorithms. The address language is the ordinary symbolic language of mathematics (whose alphabet contains a finite number of signs), supplemented by special concepts and symbols.

<u>Lines</u>. An address algorithm consists of lines. Each line contains one or several algorithmic operations. The representation of each operation is called a formula; all admissible formulas will be described below. The formulas in a line (if there are several of them) are separ-

ated by the ";" symbol (semicolon), or the "," symbol (comma). The first of these indicates that it is not permissible to interchange the formulas in a line, while the second indicates that this is permissible. Lines are separated from one another by the fact that they are written one above the other (in coding for the computer, a "." (period) may be used as a separating symbol.

<u>The prime operation and address mapping</u>. The prime operation defines some function of a single variable. Its symbol (a prime) is written above and to the left of the argument

$$'a = b,$$

where <u>a</u> is the argument and <u>b</u> the result of the operation.

We read this as follows: prime <u>a</u> equals <u>b</u> (or <u>b</u> is the content of <u>a</u>). The ' (prime) function defines a certain mapping of the set of addresses A onto the set of contents B, which we shall call the address mapping.

The mapping of set A onto set B must be single-valued (to one address there must correspond just one content). The inverse mapping may be one-many. No more precise meaning should be read into the prime operation.

In abstract discussions, no restrictions are imposed on the sets A and B, i.e., the prime function may be defined on an arbitrary set of arguments A, and will have values in an arbitrary finite set B. For purposes of applications, however, both sets are restricted, and are sets consisting of certain codes.

To represent the members of the address set, we use the letters of some initial (finite) alphabet with or without indices, the positive integers, as well as words formed from the letters of the initial alphabet and digits. As such an initial alphabet, we might take a combination of the upper- and lower-case Latin, Russian, and Greek alphabets. Then to represent addresses we might use, for example, the notation: α_2, 1012, $\alpha 3$, sum, sum 2, ω.

If $x\bar{\epsilon}A$, then the expression $'x$ is meaningless. We shall assume that the prime operation is applicable to those elements that are used as its arguments.

The repeated application of the prime operation leads to the concept of an address of rank two. Thus, if $'a_1 = a_2$ and $a_2 \epsilon A$, i.e., a_2 is itself an address, there is a d such that $'a_2 = d$. This relationship is represented symbolically as

$$^2a_1 = '('a_1) = {}''a_1 = 'a_2 = d.$$

In this case we say: a_1 is an address of rank two or the specificier by d.

In like manner, we can introduce the concept of addresses of higher rank. Thus, the notation

$$^4a = d$$

indicates that there are elements $x_1; x_2; x_3 \epsilon A$ such that $'a = x_1$; $'x_1 = x_2$; $'x_2 = x_3$; $'x_3 = d$, i.e.,

$$^4a = {}^3x_1 = {}^2x_2 = 'x_3 = d.$$

We agree to call d the address of rank zero for the quantity d.

Address functions. The generally accepted concept of a function and the corresponding expression constructed from symbols for one-term and two-term operations, parentheses, and symbols for variables are expanded; we include the prime function as one of the admissible elementary operations. Such functions are said to be address functions.

In writing address functions, we assume that all symbols are employed in regular manner. Thus, any function whose expression contains no prime-function symbol will be a special case of an address function.

In the language, expressions for address functions are used to indicate dependent algorithmic operations (formulas or flags for unconditional jumps), as well as to construct representations for other algorithmic operations (transfer, interchange, predicate, entrance, relative-jump, and looping formulas). If we are given the mapping defining a prime operation, then for any address function we can find its value.

An address function may be written formally, however, i.e., without preliminary specification of the mapping of set A onto set B, defining the meaning of the prime operation. In this case, the address function symbolizes operations on the as-yet unknown content of expressions entering into it by way of the ' symbol. If we now specify a mapping defining the prime operation (where set A includes all symbols contained in the functions to which this symbol applies), the function will acquire some particular meaning.

Transfer operation. To define and modify an address mapping, we employ a transfer operation, whose symbol is $\Rightarrow$ (an arrow connecting two elements). The operation expression $d \Rightarrow a$ (d transferred according to address a) means that:

1) element a is contained in address set A;

2) element d is included in content set B;

3) the relationship $'a = d$ is established;

4) all previously established relationships of the form $'x = y$, where $x \neq d$, remain unchanged.

Thus after the transfer $d \Rightarrow a$, there will be a change in the value of all address functions containing the expressions $'a$, ^2a, ^3a, etc., while all address functions not containing these expressions will remain unchanged.

Labels. A label may be a number, letter, or word made up of numbers and letters with or without indices.

Labels may be values of address functions. Thus labels are considered to be a special case of an address function whose value is constant and which is a label.

Labeled lines. In order to indicate the order of application of algorithm lines, various of the lines may be marked off by one or several (different) labels. In this case, the label followed by the symbol "..." (ellipsis) is placed to the left of the labeled line. An ellipsis in the representation of an algorithm always means that to the left of it there is a label, marking off a given line. The corresponding lines of the algorithm are said to be labeled.

Let us now list the *admissible algorithmic operations* and describe the *language formulas* corresponding to them.

Computable-transfer formulas. The representation of an address function whose values may be labels (in the special case, simply a label) may represent an individual algorithmic operation called a computed-transfer formula. Such an algorithmic operation consists in proceeding to execute the algorithm line indicated by the label equal to the value of this address function calculated for a given address mapping.

For the special case in which the address function is simply a label, the formula is called an *unconditional-jump lable*. Thus, a label (not followed by an allipsis) may form a separate line of the algorithm and in this case the operation designated by it consists in jumping to the line indicated by the same label.

The execution of a computed-jump formula (or unconditional-jump label) does not change the address mapping.

Stop formulas. The two special symbols 8 and ! are used; they are called, respectively, relative- and unconditional-stop formulas. The employment of the 8 symbol is associated with entrance formulas; the ! symbol indicates an operation — the end of an algorithm. The 8 can also represent the end of an algorithm provided it has not been preceded by the corresponding entrance formula.

The stop formulas either form separate lines of the algorithm, or occur as components of other formulas in the address language (see predicate, entrance, looping, and substitution formulas).

Relative-jump formula. Expressions of the form

$$\wr N$$

may form separate lines of the algorithm; they are called relative-jump formulas. Here N is an address function whose values are positive or negative integers or simply nonzero integers; $\wr$ is a special symbol.

A relative-jump formula is a representation of the following algorithmic operation: a transfer is made to the line located above or below the given line by a number of lines equal to the value of N, calculated for the given address mapping: the jump is downward if the value is positive and upward where it is negative.

Transfer formulas. Expressions of the form

$$f_1 \Rightarrow f_2$$

may form separate lines of the algorithm; they are called transfer formulas. Here f_1 and f_2 are address functions.

A transfer formula symbolizes the following algorithmic operation: the value of the address function f_1 is transferred according to an address equal to the value of the function f_2. Thus, execution of a transfer formula changes the address mapping and thus the value of several address functions. Here it is understood that all of what we have said in defining the transfer operation applies to the values of the functions f_1 and f_2. We should note that the definition of transfer formula should be taken in the sense that $'f_2$ will equal the value of f_1 prior to the transfer after the transfer formula has been executed. This is important where the expression f_1 contains $'f_2$, $^2f_2$, etc.

It is also permissible to write address formulas in the form

$$\varnothing \Rightarrow a,$$

where $\varnothing$ represents the "empty" symbol (a position for placing the actual parameters). Here it is assumed that at the instant the call to the address formula containing an "empty" left side is made, the algorithm will specify which symbol should be located in this position.

Interchange formula. Expressions of the form

$$f_1 \Leftrightarrow f_2$$

may form separate lines of the algorithm; they are called interchange formulas. Here f_1 and f_2 are address functions. An interchange formula represents the following algorithmic operation: for a given address mapping, the values of the address functions f_1 and f_2 are computed; they are taken as addresses. The operation consists in interchanging the contents of these addresses (the latter must be specified at the time the operation is executed). The contents of the remaining addresses remain unchanged. Thus, the formula

$$a \Leftrightarrow c$$

(where a and c are addresses) represents an operation equivalent to operations represented with the aid of a sequence of three transfer formulas:

$$\begin{gathered} 'a \Rightarrow r \\ 'c \Rightarrow a \\ 'r \Rightarrow c \end{gathered}$$

In the general case, where f_1 and f_2 are arbitrary address functions, this operation is equivalent to the following four transfer formulas executed in sequence:

$$\begin{gathered} a \Rightarrow r \\ 'c \Rightarrow r_1 \\ 'a \Rightarrow c \\ 'r_1 \Rightarrow r \end{gathered}$$

Here r and r_1 are addresses that are free for the given algorithm (working addresses).

List of formulas. In one line of the logarithm, we may write several transfer or interchange formulas. In this case, we must place between them one of the following symbols: ";" (semicolon) or "," (comma). The first of these symbols indicates that it is not admissible to change the order of execution of the operations separated by it, while the second indicates that it is possible to interchange these operations or to execute them simultaneously. Such a line is called a formula list, and indicates that all of the operations enumerated in it are to be executed with the specified indications as to the order of execution.

List of formulas with jump. At the end of a list of transfer or interchange formulas given in a single line (or following one such formula), after the ";" symbol there may be one of the formulas for a calculated or relative jump. Such a line is called a formula list with a jump and, as in the preceeding case, indicates execution of all the operations which it enumerates, the last being a jump.

Entrance and subroutine formulas. Entrance formulas are used to represent a given transformation, which for some reason cannot be written out at the given location. Such a transformation is called a subroutine and it is assumed that it is discribed in some manner or other (not necessarily in the address language). It is necessary, however, that the description of the subroutine specify:

1) an ordered list of input and output parameters;

2) the instant at which calculations are concluded;

3) the label assigned to the given subroutine (its name).

Such parameters include, in addition to information on arguments, sizes of blocks, etc., the labels for those subroutines whose use is to be determined by the main program. Thus, for example, if the subroutine is a procedure for computing a definite integral in accordance with a particular scheme then, in addition to the limits of integration, the interval, etc., the input data must include specification of the label for the subroutine used to compute the values of the integrand.

Subroutines may be written in the address language. In this case, the requirements for the representations are as follows:

1) the initial (entrance) line of the subroutine is indicated by a label which serves as its name;

2) the initial line of the subroutine consists of address formulas with "empty" left sides (in the course of execution of the subroutine, they will be replaced by elements of the list on the basis of the entrance formula). The order of execution of these formulas plays no role, but their representation is ordered and agrees with the list of the entrance formula;

3) at least one B symbol is to be used in the subroutine.

The formula for entering a subroutine into an algorithm is

$$\Pi\alpha\{a_1, \ldots, a_n\}\beta,$$

where Π is the symbol for an entrance formula; α is the formula (or label) of a computed jump; β is the formula (or label) for a computed or relative jump or one of the stop formulas; $a_1, \ldots, a_n$ is the list of address functions.

The formula means:

a) go to the subroutine with label α (or equal to the value α);

b) the first expressions of the list $a_1, \ldots, a_n$ are assumed to be an ordered list of the subroutine arguments;

c) executing the subroutine, transfer the ordered list of results into the remaining addresses of list $a_1, \ldots, a_n$ of formulas of Π. Thus the number of terms in this list must equal the sum of the number of input and output parameters for the given subroutine;

d) go to the algorithm line with label β (or equal to the value of β).

For a subroutine given in the address language, an entrance formula indicates a jump to the line with label α, insertion (maintaining the sequence) of the list of address expressions $a_1, \ldots, a_n$ at places indicated by $\varnothing$ symbols in the left sides of the formulas of the first subroutine line, and replacement of the B symbol by label β (all substitutions are made not while the algorithm is being written, but only in the course of its execution).

A relationship is established between entrance formulas and stop formulas, resembling the relationship between initial and terminal parentheses in algebraic formulas.

A subroutine is said to be the domain of operation of an entrance formula.

Predicate formulas. Let L represent a certain statement. Expressions of the form

$$P\{L\}\alpha\downarrow\beta$$

may form separate lines of an algorithm; they are called predicate formulas, where P is the predicate-formula symbol; $\downarrow$ is a separating symbol; α and β are the upper and lower values of the predicate formula and each of them may be one of the previously described lines.

The algorithmic operation corresponding to a predicate formula consists in execution of the line representing the upper value if the corresponding statement is true and of the line representing the lower value if it is false.

If the value of L is always true, then in place of a predicate formula we may write the upper value, while if it is always false, we may write the lower value. In this sense, any of the lines presented earlier will be a special case of a predicate formula.

If one of the values of a predicate formula is the label for a following line, the latter may be omitted (together with the $\downarrow$ symbol for the lower value).

We note that we can write out any algorithm using just the ! halt formula, transfer formulas, and the special predicate-formula case where the values are labels. All the remaining formulas are introduced for the sake of convenience.

Substitution formulas. Expressions of the form

$$З(a_1 \to c_1, \ldots, a_n \to c_n)\alpha, \beta$$

are said to be substitution formulas, where Z is the formula symbol; α, β are formulas for computed or relative jumps or simply labels; $a_1 \to c_1, \ldots, a_n \to c_n$ is the list of substitutions.

The algorithmic operation represented by a substitution formula consists in the following:

a) for a given address mapping, labels are determined – values of the formulas α and β; the lines of the algorithm between these labels define the domain of operation of the substitution formula;

b) the lines of the algorithm corresponding to the domain of operation of the substitution formula are executed; in the course of execution, the symbols c_1 corresponding to the symbols a_1 are replaced in these formulas;

c) the machine may exit from the domain of operation of the substitution formula owing to its exhaustion (it will go to the next line of the algorithm) or owing to the operation of some jump formula; in either case, when the domain of operation is left, the initial representation of the algorithm will be restored in it.

If α is the label for the line following the substitution formula, it may be omitted in the representation of the substitution formula. In this case, the formula will take on the form

$$З(a_1 \to c_1, \ldots, a_n \to c_n), \beta.$$

If the domain of operation of the substitution formula consists of the single line following it, then both formulas α and β are dropped in the representation of the formula.

Substitution formulas permit us to employ still another convenient representation for subroutines. Here instead of the line of transfer formulas with "empty" left side (see entrance formula) the subroutine begins with a substitution formula, whose substitution list contains the same symbol $\varnothing$ in place of the left sides.

Subroutines with a first line consisting of address formulas shall be called subroutines with transfer. Their representation remains unchanged even while the subroutine is being run. Subroutines whose first line contains a substitution formula shall be called subroutines with substitution (with readdressing). Their representation will change while they are being run, since in this case one symbol is replaced by another. After running, the representation is restored. Mixed subroutines are admissible.

<u>Loop formula</u>. In finite ordered sets, the concept of the succession operation is introduced. The symbol C represents "next." In each concrete case, the succession operation is given in some manner. Without imposing any restrictions on the succession operation, we shall require that its application to any element of a set yield (and uniquely) the next element.

An information element is said to be scanned by an algorithm if the representation of the algorithm contains an address for it in some rank. Algorithms including in their representation the addresses in some rank for these information elements are called information-element scanning schemes.

It is natural that in the general case a successor operation cannot be established directly in the set of initial-information elements. The successor operation for the initial-information elements. The successor operation for the initial-information elements may be established with the aid of a prime-operation with respect to the successor operation in the address set. We shall assume that we are free to select the address set as we wish. The task of programming consists in constructing schemes for surveying blocks with the aid of the successor operation in their address blocks.

We write an ordered set in the form

$$\{a,\ C\varnothing,\ P\{L\}\},$$

where <u>a</u> is the first element; C is the succession operation (we may insert any element of the set in place of the symbol $\varnothing$); L is the condition requiring that an element belong to the set.

We shall assume that in some algorithm we are required to execute the F operation successively on all elements of the set $\{a,\ C\varnothing,\ P\{L\}\}$ and go to the line with label ℓ. The address program for such an algorithm may be written in the form

$$\begin{array}{ll} & a \Rightarrow \pi \\ K \ldots & P\{L\}\downarrow l \\ & \Phi('\pi) \\ & C'\pi \Rightarrow \pi \\ & K \end{array}$$

To shorten the representation of such schemes, we introduce the concept of loop formulas on the address π

$$\underset{\Phi('\pi)}{Ц\{a,\ C\varnothing,\ P\{L\}\Rightarrow\pi\}\,\alpha,\ l}$$

or a loop formula on a parameter π

$$\underset{\Phi(\pi),}{Ц\{a,\ C\varnothing,\ P\{L\}\to\pi\}\,\alpha,\ l}$$

where $Ц$ is the formula symbol, α is the label for the next line of transformation F.

We introduce the concept of the domain of operation of the loop formula. The domain of operation of a concrete loop formula includes the lines of the algorithm between the formula and the line with label α. The domain of operation of a loop formula contains another loop, substitution, or entrance formula, the domain of operation for the latter will be contained in the domain of operation of the first.

A line with a loop formula may be labeled, and it may be transferred to other lines of the algorithm. It is established that if a jump to a loop formula occurs from outside its domain of operation, scanning of the elements begins with the first; if, however, the transfer takes place from within the domain of operation, scanning is continued.

For sets with a succession operation defined by relationship $C\varnothing = \varnothing + b$, the notation

$$\{a(b)P(L)\} \text{ or } \{a(b)c\},$$

where c is the last element of the set.

In this case, the loop formulas will have the form

$$\text{Ц}\{a(b)P(L) \Longrightarrow \pi\}$$

or

$$\text{Ц}\{a(b)c \rightarrow \pi\} \text{ etc.}$$

If the domain of operation of a loop formula consists of a single line or coincides with the domain of operation of a loop formula written in the following line, the label α is omitted

$$\text{Ц}\{\ldots\},\ \beta.$$

If β is the label for the line following directly after the domain of operation of this formula, it may also be omitted

$$\text{Ц}\{\ldots\}\alpha \quad \text{or} \quad \text{Ц}\{\ldots\}.$$

A loop formula may provide for simultaneous scanning of several sets, for example,

$$\text{Ц}\{a_1(b_1)c_1 \rightarrow \pi_1;\ a_2(b_2)c_2 \Longrightarrow \pi_2\}\alpha,\ \beta.$$

In this case, scanning continues as long as all of the specified membership conditions remain valid, and just one of them may be indicated in the formula.

Looping is also permitted on the basis of a single address in the set, where one succession operation is specified for the initial group of elements, another for the next group, etc. As an example,

$$Ц\{a_1(b_1)c_1;\ a_2(b_2)c_2 \longrightarrow \pi\}\ \alpha,\ \beta.$$

<u>Formal definition of address algorithm</u>. An address algorithm consists of the initial address mapping and a finite number of lines written one on the other, and an indication of the address set for the resulting mapping. Each line may contain one formula of the following types: computed jump (or unconditional-jump label); stop; relative jump (or relative-jump label); transfer; interchange; entrance; predicate; substitution; loop; and also lists of formulas or formulas with jumps. Any line of the algorithm may be labeled.

The initial address mapping is specified in the form of a sequence of equations

$$'a = b,$$

where <u>a</u> is the address and <u>b</u> the content.

The resultant address mapping is given by listing the addresses whose contents will be the solution of the problem following completion of running of the algorithm.

The sequence of algorithm execution is governmed by the following rules:

1. The line specified by a specially indicated initial label is executed first. If there is no such indication, the line occurring first in the representation is executed first.

2. The order of execution of lines containing formula lists is specified when they are declared.

3. Formulas for computed and relative jumps, lists of formulas with a jump, entrance, substitution, and loop formulas, as well as predicate formulas whose values contain jump formulas are indicated by a

successor line.

4. After execution of the line that does not indicate its successor, we go to the next line in the representation.

5. The formula for an unconditional stop! or the formula for a relative stop 8. corresponding to no entrance formula represents the end of algorithm operation.

From the viewpoint of content, an address algorithm is a certain manipulation of information on the problem that produces its solution. This manipulation, in form, is the establishment and variation of some address mapping.

Information on the problem is specified by a certain address mapping. In many cases, therefore, the algorithm begins with the establishment of a certain initial address mapping by means of several transfers or a table for such a mapping.

The result obtained by running an algorithm is the content of the addresses indicated as the resultant set.

ADDRESS-LANGUAGE LEVELS AND STYLES

The address language, although based on the addressing principle, imposes no requirements on the order of the address set. An address mapping may be defined on an arbitrary address set. For problems whose nature involves the order of the elements we may introduce succession operations not even described algorithmically. For problems whose nature involves ording of the elements, we may introduce succession operations not even described algorithmically. The level of ordering of the addresses, the level of the algorithmization of the succession operations introduced in a certain sense is the level of the language.

It is necessary to consider three fundamental levels of the address language [15, 18]:

1. The general algorithmic level, at which the set of addresses

most natural for the given concrete problem is used, and the succession operations are described by general mathematical means, for example, with the aid of indices. At this level, the address language is very similar to Algol [13].

2. The level of arbitrary (or symbolic) addresses, which requires ordering of the addresses of the elements making up the blocks processed by the algorithm. At this level we consider the linearity and boundedness of the set of addresses, which match the technical potential of the computers; as a rule, unless otherwise specified, we speak of a language at the level of symbolic addresses.

3. The level of absolute addresses, which assumes that the address set is completely ordered in association with a particular machine, with allowance for the set of absolute location addresses and their ranking.

Provision has been made in the address language itself for going from level to level, from the most abstract algorithmic language to the level of complete allocation of addresses for a given computer. This is the reason for the particular suitability of the address language as the source language for translator programs. Separate levels may serve as intermediate internal languages for translators [PP] for the purpose of translating the algorithm as written into a computer program in steps, through intermediate representation.

The free utilization of the representational abilities of the language permits us to describe the same algorithm by different methods. To simplify translation from the free address language into the language of an actual computer, we may define the styles of the language.

The style of the language imposes restrictions on the use of its expressive abilities such that translation into the language of the given computer will be simpler. Translation from the general language into a particular style of the language is possible at any level. The task of a PP is to translate from the general address language. This translation may be divided into stages, however, by the introduction of a a series of styles approaching ever closer to the machine language. Thus the work of the PP is reduced fundamentally, to formal transformations within the address language, to translation from style to style.

Depending on how complex a PP we are able to realize, the breadth and flexibility of the address language permits us to select a point before which the problem program must be written manually so that the rest of its processing may be carried out by the PP. Owing to this, today it is possible in practice to automate the programming of certain program segments. Thus the introduction of a PP with the address address language as source for the "Kiev" computer — the PP-AK (see below), as well as PP using the same source language for other computers (the "Ural" and the UMShN, the broad-function control computer, and others) has made it possible to increase programmer productivity by many times. Owing to the generality of the source languages, the existence of these PP makes it possible, where needed, to transfer problem solution from one computer to another with no special difficulties, as well as to expand the group of people able to write programs (in the address language) for computers to include those concerned with direct algorithmization of various information-handling processes and even to those in total ignorance of machine codes. This last factor must be considered especially important in view of the increasing needs for personnel caused by the increase in the number of computers in use and their operating speed.

Thus work ordinarily performed by a skilled programmer can be divided into too essentially different segments. Work on the refinement and writing of algorithms, which requires a high degree of mathematical skill, will be performed by programmer engineers. The result of this work will be an address program. Work on the translation of the latter into machine language is carried out by translator programs. Coding of address programs (and sometimes, translation into the required style as well) together with work at the computer is performed by moderately skilled operators. The prospective introduction of automatic reading devices promises to minimize this work.

EXAMPLE OF ADDRESS-PROGRAM WRITING

Let us write an algorithm for solving a system of linear algebraic equations with a symmetric coefficient matrix using the improved Gauss method [14].

Let $A = \{a_{ij}\}$ $(i, j = 1, 2, \ldots, n; a_{ij} = a_{ji})$ be the initial matrix and $F\{a_{i,n+1}\}$ the vector for the right sides.

In accordance with the method selected, the algorithm is split into two stages — direct stage and inverse stage.

Direct and inverse stages

Direct stage. The elements $a_{i\ell}$ are transformed directly, line-by-line, by means of the recursion formulas:

$$\bar{a}_{1j} = a_{1j} \ (1 \leqslant j \leqslant n+1);$$

$$\bar{a}_{ij} = a_{ij} - \sum_{k=1}^{i-1} \frac{\bar{a}_{ki}\bar{a}_{kj}}{\bar{a}_{kk}} \ (2 \leqslant i \leqslant n;\ i \leqslant j \leqslant n+1).$$

Inverse stage — solution of the system of equations with the triangular matrix $\{\bar{a}_{ij}\}$ $(i = 1, 2, \ldots, n;\ i \leqslant j \leqslant n+1)$, yielded by the direct stage.

The calculation formulas have the form

$$x_i = \frac{\bar{a}_{i,n-1} - \sum_{k=i+1}^{n} a_{ik}x_k}{\bar{a}_{ii}} \ (i = n, n-1, \ldots, 1).$$

1. Writing of programs at the general-algorithmic level. We take as the initial mapping

$$'a_{ij} = a_{ij} (i = 1, 2, \ldots, n;\ 1 \leqslant j \leqslant n+1).$$

Let the s_i be the addresses set aside for the components of the vector for the solutions $x_i (i = 1, 2, \ldots, n)$. Let the elements $\bar{a}_{ij} (1 < i \leqslant n;\ i \leqslant j \leqslant n+1)$ be calculated from left to right, line-by-line. Then it is not difficult to see that the newly obtained elements $\bar{a}_{ij}$ may be matched to the same addresses α_{ij}.

<u>Forward-stage algorithm</u>

<u>Initial address mapping</u>

$'a_{ij} = a_{ij}$
$(i = 1, 2, \ldots, n;$
$j = 1, 2, \ldots, n+1)$

<u>Address algorithm 1</u>

```
FORWARD STAGE...
$Ц\{2(1)n \to i\}.В$
$Ц\{i(1)n+1 \to j\}$
$Ц\{1(1)i-1 \to k\}$
$'a_{ij} - \frac{'a_{ki} \cdot 'a_{kj}}{'a_{kk}} \Rightarrow a_{ij}$
```

<u>The resultant address set</u> coincides with the initial set.

<u>Inverse-stage algorithm</u>

<u>Initial address mapping</u>

$'a_{ij} = \bar{a}_{ij}$
$(i = 1, 2, \ldots, n;$
$j = 1, 2, \ldots, n+1)$

<u>Address algorithm 2</u>

```
INVERSE STAGE...
$Ц\{n(-1)1 \to i\}.M.В$
$a_{i,n+1} \Rightarrow s_i$
$Ц\{i+1(1)n \to k\}$
$'s_i - 'a_{ik} \cdot s_k \Rightarrow s_i$
$M\ldots 's_i : 'a_{ii} \Rightarrow s_i$
```

<u>The resultant address set is</u>

$$s(i = 1, 2, \ldots, n).$$

2. To go to the symbolic-address level, it is necessary to declare the successor operations algorithmically for the address set; in the programs for the direct and inverse stages, these have declared with the aid of indices. In our case, this reduces to constructing an algorithm for determining, in the linearly ordered address set, the address of a matrix element on the basis of its assigned indices.

The algorithm used to find the address of an element from its indices will be determined by the method used to code the initial information.

The presence of a scheme for information processing and a symmetric initial matrix naturally raises the notion of sequentially coding, column-by-column or row-by-row, only those matrix elements for which $j \geq i$. To be specific, we shall assume that we code column-by-column. As an example, for $n = 4$ the following scheme for allocating addresses to the initial information suggests itself:

$'(\alpha+1)=a_{11}$	$'(\alpha+2)=a_{12}$	$'(\alpha+4)=a_{13}$	$'(\alpha+7)=a_{14}$	$'(\alpha+11)=a_{15}$
	$'(\alpha+3)=a_{22}$	$'(\alpha+5)=a_{23}$	$'(\alpha+8)=a_{24}$	$'(\alpha+12)=a_{25}$
		$'(\alpha+6)=a_{33}$	$'(\alpha+9)=a_{34}$	$'(\alpha+13)=a_{35}$
			$'(\alpha+10)=a_{44}$	$'(\alpha+14)=a_{45}$

The ordinal number of an information element a_{ij} in the sequence

$$\alpha+1,\ \alpha+2,\ \ldots,\ \alpha+\frac{n^2+n}{2}+n$$

is called its reduced index $\pi(i, j)$.

It is not difficult to see that

$$\pi(i,\ j)=i+\sum_{s=1}^{j-1}s=i+\frac{j}{2}(j-1).$$

So that the programs will not depend on the parameters n (matrix dimension) and α(beginning of block in which elements are stored), we also let

$$'\alpha=n;\ '\varphi=\alpha.$$

Forward-stage program

Initial address mapping

$$'\alpha = n$$
$$'\varphi = \alpha$$
$$'(\alpha + i + \tfrac{j}{2}(j-1)) = a_{ij}$$
$$(1 \leqslant i \leqslant n;\ i \leqslant j \leqslant n+1)$$

Address algorithm 3

FORWARD STAGE...

$$\text{Ц}\,\{2\,(1)\,''\varphi \to i\},\ \text{ᗺ}$$
$$\text{Ц}\,\{i\,(1)\,''\varphi + 1 \to j\}$$
$$\text{Ц}\,\{1\,(1)\,i - 1 \to k\}$$
$$'('\varphi + i + \tfrac{j}{2}(j-1)) - \frac{'\left('\varphi + k + \tfrac{i}{2}(i-1)\right) \times {}'\left('\varphi + k + \tfrac{j}{2}(j-1)\right)}{'\left('\varphi + \tfrac{k}{2}(k+1)\right)} \Rightarrow {}'\varphi + i + \tfrac{j}{2}(j-1)$$

The resultant address set coincides with the original set.

To make it possible to obtain the result for the inverse-stage program – the solution vector – in an arbitrary address block, we introduce the address ψ

$$'\psi = s.$$

Inverse-stage program

Initial address mapping

$$'\varphi = \alpha$$
$$'\alpha = n$$
$$'(\alpha + i + \tfrac{j}{2}(j-1)) = a_{ij}$$
$$(1 \leqslant i \leqslant n;\ i \leqslant j \leqslant n+1)$$

Resultant address set

$$s + i\ (i = 0, 1, 2, \ldots, n)$$

Address algorithm

INVERSE STAGE...

$$\text{Ц}\,\{''\varphi\,(-1)\,1 \to i\}\ M,\ \text{ᗺ}$$
$$'\left('\varphi + i + \frac{''\varphi + 1}{2}\,''\varphi\right) \Rightarrow {}'\psi + i$$
$$\text{Ц}\,\{i + 1\,(1)\,''\varphi \to k\}$$
$$'('\psi + i) - {}'\left('\varphi + i + \tfrac{k}{2}(k-1)\right)'('\psi + k) \Rightarrow {}'\psi + i$$
$$M\ \ldots\ \frac{'('\psi + i)}{'\left('\psi + \frac{i(i+1)}{2}\right)} \Rightarrow {}'\psi + i$$

Each of the algorithms given may be set up as a subroutine. This makes sense since, for example, an inverse-stage program may be used to solve a system of linear equations in the general case using the method of elimination, etc.

With the method which we are using to code problem information, to obtain subroutines we need only first add to the indicated programs the corresponding lines, to which initial labels have been annexed:

$$\text{FORWARD STAGE}\ldots\ \varnothing \Rightarrow \varphi$$
$$\text{INVERSE STAGE}\ldots\ \varnothing \Rightarrow \varphi,\ \varnothing \Rightarrow \psi$$

The entrance formulas to these subroutines will have the form

$$\Pi\ \text{FORWARD STAGE}\ \{\alpha\}$$
$$\Pi\ \text{INVERSE STAGE}\ \{\alpha,\ s\}$$

We repeat that the representation of the entrance formulas as well as the representation of the algorithms agree with the information-coding method.

ADDRESS ALGORITHMS AND MATHEMATICAL MACHINES

From the mathematical viewpoint, present-day general-purpose digital computers are based on two principles: addressing and program control.

Under the addressing principle, representation of algorithmic operations in machine language is accomplished by specifying the addresses at which the elements of transformed information are to be stored. Under the program-control principle, in the solution of each concrete problem, computer operation is controlled directly by the solution algorithm, represented in the language of the given computer as a special program.

The address representation of algorithms represents these principles of addressing and program control, which are common for all modern computers. On the other hand, any algorithm is described by a finite number of operations. As a consequence, it is in principle possible to construct an IAVM (ideal address computer) for which a given address algorithm, as for all algorithms formed from the same operations and using the same addresses, will be a program [17]. To this end, we must include in the computer's elementary-operation set all operations encountered in the algorithm, and we must use an instruction-coding system corresponding to the address representation.

Actual general-purpose computers differ from an IAVM not only in having restricted memories, but also in that there elementary-operation set is restricted to a certain number of operations, even though it may be complete, i.e., it permits construction of any algorithmic operation (by successive application of elementary operations).

For the sake of simplicity, we use as addresses for various computers the initial segment of the natural-number series; the length of this segment will characterize the capacity of the computer high-speed memory. But the program registers occupy an important place in the set of locations used to store information in a computer. To permit description of operations involving these registers, the latter must be assigned address names other than the numerical addresses of ZU locations if we wish to avoid confusion. We note that in computer codes, indications of the use of program registers are either included in the operation code, or else they are coded into specially allocated digit positions in conjunction with the operation code.

Simulation of the operation of any actual computer on an IAVM reduces to selecting from its elementary operation only those contained in the operation set of the given real computer. The language thus generated will be a certain style of the address language, and the address program written when the requirements of the given style are observed requires only recoding to become a program for the actual computer. In this sense, the language of any actual computer is an address language.

On the other hand, any elementary IAVM operation may be reproduced by some sequence of operations — a routine for any actual general-purpose computer. Naturally, in going from the address program to the program for an actual computer it is necessary to allow for machine-memory capacity (and its linearity), as well as for the presence of various program registers.

The analysis of address programs shows that address programming reduces to construction of various schemes for scanning the information [10]. By using the concepts of addresses of higher rank and schemes for scanning the information on the basis of such addresses, we can considerably simplify the problem of programming and are able to write algorithms (their address programs) in a form that does not vary during the process of execution and that does not depend on the parameters of the particular problems (in contrast to programs written with instruction readdressing).

An arbitrary address algorithm may be represented in an equivalent form (equivalent in the sense of results obtained) such that all of the address functions entering into the algorithm representation appear in rank not exceeding two [16]. But algorithms describable with the aid of only first-rank address functions and not including substitution (readdressing) formulas represent a very narrow class of algorithms [11]. Thus in order to obtain algorithmic completeness, we must include in the set calling by second address or a substitution operation.

We should note that the first of these operations possess certain conveniences that the second does not offer in connection with the fact that it is better in practice to use programs that do not change while they are being run. Moreover, such programs are easily built in as blocks or consolidated elementary operations, for example, or matrix operations and in general for the solution of linear-algebra problems, for realization of various methods for solving differential equations, etc.

The transfer of information to built-in multiterm operators or to subroutines may prove impossible without specification of parameters (dimensions of vectors, matrices, orders of systems, etc.); if, however, we use standard methods for specifying information together with the operation of calling by rank-two address, it becomes possible to transfer all the needed information on blocks of complicated structure, regardless of the values of these parameters, using a single master address. We shall discuss this in more detail in the next chapter.

The development of mathematical specifications for the "Kiev" computer paralled the word on creation and development of the address language. The idea of constructing an interchangeable wired memory (suggested by Ye.L. Yushchenko) in which standard routines for realization of various algorithms could be stored in the form of wired modules was initially implemented with the aid of the NGO and OGO group operations. This same idea served as the stimulus to development of the concept of an address of higher rank (rank two), and together with it the initial concept of the address algorithm, first formulated in [10]. The analysis of several address algorithms, including algorithms for translator programs, led to the idea of a group operation based on an address of rank two, the F operation, which was also included in the operation set, and which later became very popular with the programmers. This also facilitated the further development of the address language, since it is precisely the F operation that plays a substantial role in the practical application of this language to the "Kiev" computer.

The suggestions entailed by the analysis of address algorithms were made when the computer circuits had been completely developed, and some of them had already been wired in. In particular, this affected the proposals of Ye.L. Yushchenko for the F operation, the jump on address content operation, relative jumps, etc. The attempt to maintain hardware simplicity and the existence of already developed circuitry permitted realization of only some of these operations, even though complete hardware designs were made.

ADDRESS DESCRIPTION OF "KIEV" MACHINE

Let us describe the operation set for the "Kiev" computer in address language.

We shall introduce designations for the program register and switches of the "Kiev" machine:

C represents the instruction counter — the instruction-loading device (UVK), 11 bits;

K represents the instruction register (RK), 41 bits;

P represents the link register (RV), 11 bits;

$\mathcal{U}$ represents the loop register (RTs), 10 bits;

A represents the address-modification register — address counter (SchA), 10 bits;

Tp represents the emergency-stop flipflop register (1 bit);

Tb represents the emergency-stop blocking switch ('Tb = 1 when the switch is on while 'Tb = 0 when it is off).

In describing the operations, including the operation of these registers, we shall use only the letter designations — the addresses, which shall assume to be different from the codes for the address of the internal memory unit (VZU).

According to the program-control principle used, a separate machine-operation cycle consists in execution of an instruction 'K whose code is stored at the given moment in the instruction register K, and in the transfer to this register of the code for the next instruction, i.e., the instruction that is to be executed during the next cycle.

The instruction counter C is used to store the number (address) of the next instruction in the "Kiev" computer. The execution of instruction 'K depends upon its content (code).

For convenience in describing the set of elementary operations executed by the individual instructions of the "Kiev" computer, we shall isolate the following portions of the instruction register K and label them appropriately:

A_0 represents the operation-code address register, bits 41-37;

E_1 represents the address I modifiability indicator bit, bit 36;

A_1 represents the address-I register, bits 35-25;

E_2 represents the address-II modifiability indicator bit, bit 24;

A_2 represents the address-II register, bits 23-13;

E_3 represents the address-III modifiability indicator bit, bit 12;

A_3 represents the address-III register, bits 11-1.

In connection with the number of bits set aside for the specified registers, we may store the following octal codes, respectively:

$$'A_0 = 00,\ 01,\ \ldots,\ 37;$$
$$'E_1,\ 'E_2,\ 'E_3 = 0, 1;$$
$$'A_1,\ 'A_2,\ 'A_3 = 0000,\ 0001,\ \ldots,\ 3777.$$

Depending on $'A_0$ (operation code), the computer will execute the following operations:

Fundamental Arithmetic Operations

1) $'A_0 = 01$. Addition operation +. The following address program is realized:

$$a \ldots\ '('A_1 + 'E_1'A) + '('A_2 + 'E_2'A) \Rightarrow 'A_3 + 'E_3'A$$
$$P\{'|'A_3 + 'E_3'A| > 1\} \downarrow 'C + 1 \Rightarrow C,\ б$$
$$P\{'T_6 = 1\}'C + 2 \Rightarrow C \downarrow !$$
$$б \ldots\ ^2C \Rightarrow K$$

The line with the label a means that when $'E_1 = 1$, the corresponding address is modified, i.e., it is increased by the amount of the content of the address-modification register A, while when $'E_1 = 0$, the operation is performed directly according to the address. The next line represents a test to see whether the result causes an overflow, together with a jump to the instruction carrying the next number $('C + 1 \Rightarrow C)$, if there is no overflow. When the result causes an overflow, the second predicate formula is used to test the condition of the emergency-stop switch, and when the switch is on, the computer halts; when it is off, the single instruction $('C + 2 \Rightarrow C)$ is skipped. The latter is used for automatic intervention in the computational process (rescaling, etc.). These features of instruction execution apply to all operations in the given group.

The following two operations are executed in like manner:

2) $'A_0 = 02$. Subtraction $-$.

3) $'A_0 = 12$. Division $:$. The operation of division by integral powers of 2 is equivalent to an arithmetic shift, since division is performed without rounding.

4) $'A_0 = 10$. Multiplication without rounding $\times$. The address program

$$\begin{gathered} '('A_1 + 'E_1'A) \times '('A_2 + 'E_2'A) \Rightarrow 'A_3 + 'E_3'A \\ 'C + 1 \Rightarrow C \\ {}^2C \Rightarrow K \end{gathered}$$

is realized.

5) $'A_0 = 11$. Multiplication with rouding $\boxed{\times}$; execution of this operation resembles that of the preceding operation.

<u>Auxiliary Arithmetic Operations</u>

6) $'A_0 = 03$. Instruction addition SℓK.

$$\begin{gathered} |'('A_1 + 'E_1'A) + |'('A_2 + 'E_2'A)\| \vee \mathrm{sign}'('A_2 + 'E_2'A) \Rightarrow 'A_3 + 'E_3'A \\ 'C + 1 \Rightarrow C \\ {}^2C \Rightarrow K \end{gathered}$$

The $S\ell K$ operation differs from the addition operation in that in this case we add to the absolute value of the content of address $'A_2 + 'E_2'A$ (instruction code) the content of the address $'A_1 + 'E_1'A$ (readdressing constant) and give the result the sign $('A_2 + 'E_2'A)$ (the "sign" of the instruction).

7) $'A_0 = 06$. Operation of subtraction of absolute values "$| - |$."

8) $'A_0 = 07$. Cyclic addition Ts +; this is addition of words with elimination of overflow from the highest-order digit position. As with the preceding operations, address modification is permitted.

Logical Operations

After each of the operations in this group, control jumps to the instruction bearing the next number.

9) $'A_0 = 13$. Logical shift L $\rightarrow$. The code $'('A_1 + 'E_1'A)$ (including the sign bit) is shifted through the number of bits indicated in address III of the word having its address $('A_2 + 'E_2'A)$; a right shift takes place if $'('A_2 + 'E_2'A) < 0$ and a left shift occurs in the opposite case; the result is stored according to the address $'A_3 + + 'E_3'A$.

10) $'A_0 = 35$. Normalization N. The number $'('A + 'E_1'A)$ is normalized, the exponent of the number is stored in the six lowest-order bits on the basis of the address $('A_2 + 'E_2 'A)$, while the mantissa is stored according to $'A_3 + 'E_3'A$. Moreover,

$$'C + 1 \Longrightarrow C$$
$$^2C \Longrightarrow K$$

11) $'A_0 = 14$. The operation of digit-by-digit logical addition $\vee$

$$'('A_1 + 'E'A) \vee '('A_2 + 'E_2'A) \Longrightarrow 'A_3 + 'E_3'A$$
$$'C + 1 \Longrightarrow C$$
$$^2C \Longrightarrow K$$

The following two operations are executed in similar fashion:

12) $'A_0 = 15$. Operation of digit-by-digit logical multiplica-

tion $\wedge$.

13) $'A_0$ = 17. Digit-by-digit logical inequality operation $\tilde{=}$.

<u>Control-Transfer Operations</u>

All control-transfer operations leave the content of the VZU unchanged; the results of their execution affect only certain special registers.

14) $'A_0$ = 16. Operation of conditional control transfer on equality SrZ. This operation realizes the program

$$\begin{gathered} P\{'('A_1 + 'E_1'A) = '('A_2 + 'E_2'A)\}\,'A_3 + 'E_3'A \Rightarrow C \downarrow 'C + 1 \Rightarrow C \\ {}^{2}C \Rightarrow K \end{gathered}$$

15) $'A_0$ = 04. Conditional control transfer on relationship "less than or equal to" Crl

$$\begin{gathered} P\{'('A_1 + 'E_1'A) \leqslant '('A_2 + 'E_2'A)\}\,'A_3 + 'E_3'A \Rightarrow C \downarrow 'C + 1 \Rightarrow C \\ {}^{2}C \Rightarrow K \end{gathered}$$

16) $'A_0$ = 05. Conditional control transfer on relationship "less than or equal to, neglecting signs" Cr2

$$\begin{gathered} a \ldots,\ P\{|'('A_1 + 'E_1'A)| \leqslant |'('A_2 + 'E_2'A)|\}'A_3 + \\ + 'E_3'A \Rightarrow C \downarrow 'C + 1 \Rightarrow C \\ C \Rightarrow K \end{gathered}$$

17) $'A_0$ = 31. Conditional control transfer on sign of number UPCh

$$\begin{gathered} P\{'('A_1 + 'E_1'A) \leqslant -0\}'A_3 + 'E_3'A \Rightarrow C \downarrow 'A_2 + 'E_2'A \Rightarrow C \\ {}^{2}C \Rightarrow K \end{gathered}$$

18) $'A_0$ = 30. Conditional jump to subroutine UPP

$$\begin{gathered} P\{'('A_1 + 'E_1'A) \leqslant -0\}\,'A_3 + 'E_3'A \Rightarrow C; \\ 'A_2 + 'E_2'A \Rightarrow P \downarrow C + 1 \Rightarrow C \\ {}^{2}C \Rightarrow K \end{gathered}$$

Thus if the indicated condition is satisfied, the content of the link register P is replaced by the quantity $'A_2$ + $'E_2A$ and the machine proceeds to execute the command whose number is specified in address III. Here $'A_3$ + $'E_3'A$ is the address of the initial subroutine instruction, while 'P is the number of the instruction which is to receive control after execution of the subroutine. If this condition is not satisfied, control goes to the program instruction carrying the next number.

In this case, the corresponding subroutine concludes with the special instruction PRV.

19) $'A_0 = 32$. Jump governed by link register PRV. This instruction causes execution of the following operations:

$$\begin{gathered} 'P \Rightarrow C \\ 0 \Rightarrow P \\ ^2C \Rightarrow K \end{gathered}$$

The result of the operation does not affect the contents of registers A_1, A_2 or A_3.

<u>Peripheral-Equipment Calling Operations</u>

All operations in this group are group operations, i.e., they refer to sequences of codes.

20) $'A_0 = 20$. Load numbers V1. Into high-speed storage (OZU) locations having addresses

$$'A_1,\ 'A_1+1,\ \ldots,\ 'A_2$$

the computer is to load words first converted from the binary-coded decimal system into the binary system. We shall arbitrarily represent this group operation in the form

$$'(ПЛ)\ \text{converted into binary} \Rightarrow \binom{'A_1}{'A_2}$$

In addition, the following transfers are made:

$$\begin{gathered} 'C+1 \Rightarrow C \\ ^2C \Rightarrow K \end{gathered}$$

21) $'A_0 = 21$. Load instructions V2; this is executed in the same way as the V1 operation, except that words are transferred without conversion

$$'(ПЛ) \Longrightarrow \binom{'A_1}{'A_2}$$
$$'C + 1 \Longrightarrow C$$
$$^2C \Longrightarrow K$$

22) $'A_0 = 22$. Code readout Pech. This is an inverse operation: the contents of locations $'A_1$, $'A_1 + 1$, ..., $'A_2$ are read out. In addition,

$$'A_3 \Longrightarrow C$$
$$^2C \Longrightarrow K$$

23) $'A_0 = 23$. Exchange of OZU codes with external ZU (MB) in write mode — MBZ. The codes contained in the sequence of OZU locations

$$'A_1,\ 'A_1 + 1,\ \ldots,\ 'A_2,$$

are transferred to MB:

$$\binom{'A_1}{'A_2} \Longrightarrow MБ;$$

and in addition

$$'A_3 \Longrightarrow C$$
$$^2C \Longrightarrow K$$

24) $'A_0 = 24$. Transfer of codes between OZU and MB in read mode — MBCh. Codes are transferred from MB into a sequence of OZU locations $'A_1$, $A_1 + 1$, ..., $'A_2$, i.e.,

$$'(MБ) \Longrightarrow \binom{'A_1}{'A_2};$$

and, in addition,

$$'A_3 \Longrightarrow C$$
$$C \Longrightarrow K$$

25) $'A_0 = 25$. Preparatory operation for MBZ and MBCh operations, ensuring proper preparation of the magnetic drum — MBP. Here $'A_1 = 1$ for the 23 operation and equals 0 for the 24 operation; $'A_2 = n \cdot 2^{-22}$; $n = 1, 2, 3$ — number of the MB from which the codes are to be taken; $'A_3$ — number of number on MB with which the corresponding operation must commence. In addition, the following operations are performed:

$$'C + 1 \Longrightarrow C$$
$$^{2}C \Longrightarrow K$$

26) $'A_0$ = 33. Stop.

<u>Address-Modification Operations</u>

The address-modification operations are group operations in the sense that the modification-register content formed by them may be used as an instruction group.

27) $'A_0$ = 26. Beginning of group operation (NGO). With the NGO operation:

a) the number characterizing the number of loops in the cyclic process is transferred to the loop register Ts,

$$'A_1 \Longrightarrow Ц;$$

b) the readdressing constant

$$'A_2 \Longrightarrow A;$$

is transferred into the address-modification register A;

c) the predicate formula

$$P\,\{'Ц = 'A\}\,'A_3 \Longrightarrow C \downarrow 'C + 1 \Longrightarrow C;$$

is realized;

d) $^{2}C \Longrightarrow K$.

Thus the NGO instruction prepares the content of the modification register A so as to provide the required modification of variable addresses.

If we know the number of loops N and the readdressing interval <u>p</u> in advance, then we let

$$'A_1 = 'A_2 + Np.$$

In this case, at each repetition of the loop the content of the A register is increased by <u>p</u>. Since $'Ц \neq 'A$, loop calculations continue; at $'Ц = 'A$, the computer exits from the loop, going to the instruction with number $'A_3$.

28) $'A_0 = 27$. End of group operation OGO. With this instruction:

a) the content of the A register is increased by the readdressing interval $'A_1 + 'A \Rightarrow A$;

b) the predicate formula

$$P\{'U = 'A\}\, 'A_3 \Rightarrow C \downarrow 'A_2 \Rightarrow C$$

is realized.

c) $^2C \Rightarrow K$.

Here $'A_3$ is the number of the instruction that receives control after looping has terminated; $'A_2$ is the number of the instruction to which control is transferred when looping continues. (We note that $'A_2$ is not the number of the NGO instruction, since the latter will not be repeated here when we go from one loop to another, since its repetition would lead to restoration of the initial setting of the A register).

29) $'A_0 = 34$. Operation of calling by address of rank two (calling by specifier) F.

In addition to the NGO operation, which sets the address-modification register, the computer can realize the F operation, which also performs this function. While the value to be transferred to the A register is specified explicitly in the NGO instruction (as $'A_2$) with the F instruction, this value is specified only by its address. The F instruction involves execution of the following operations:

$$\begin{aligned}
&{}^2('A_1 + 'E_1'A)_{II} \Rightarrow A \\
&'('('A_1 + 'E_1'A)_{II}) \Rightarrow 'A_3 + 'E_3'A \\
&'C + 1 \Rightarrow C \\
&{}^2C \Rightarrow K
\end{aligned}$$

($'A_2$ is not used in execution of the instruction).

Here we let $'\alpha_{II}$ represent the content of address II of location α; with $'A_1 = \alpha$; $'\alpha_{II} = \beta$, we have with the F operation:

$$\begin{aligned}
&\beta \Rightarrow A \\
&'\beta \Rightarrow 'A_3 \\
&'C + 1 \Rightarrow C \\
&{}^2C \Rightarrow K
\end{aligned}$$

For arbitrary cyclic parametric processes, the availability of the F operation permits us to write programs that do not change in the course of their execution (they do not use instruction readdressing).

GROUP OPERATIONS OF THE "KIEV" COMPUTER

Problems pertaining to the checking out of programs in the solution of problems by TsAM are of primary importance. If two different programs can be used to solve the same problem in accordance with the selectected method, from the viewpoint of convenience in loading the program into the machine and speed of debugging, preference should be given to the program that takes up the fewest memory locations, i.e., which contains information on the problems in the more compact form. The more condensed program may be selected even where the direct calculation governed by it takes somewhat longer to perform. It is better to use a program with a larger number of storage locations if its instructions may be stored in passive memory, since this is more convenient from the viewpoint of monitoring and loading.

Savings in program instructions may frequently be achieved by the inclusion in the computer's elementary-operation set of instructions for readdressing and ordered storage of quantities in a sequence of storage locations.

In the general case, programs will include schemes for scanning several sequences or tables, i.e., they will be complex loop processes depending on parameters.

There are several methods known that occassionally permit us to reduce program length (transfers to standard addresses, transfer from standard addresses into a sequence, etc.). We can consider the problem of shortening loop programs, not by clever programming, but by using computer circuitry to realize certain special instructions.

Special instructions for which information on the execution of loop programs is specified in compact form with the aid of a reduced number of words are called group instructions. The introduction of group operations also serves the purpose of using computer passive storage.

We shall now give a more complete description of the group operations for the "Kiev" computer.

I. Let a cyclic address program contain a group of addresses varying from loop to loop in accordance with the law

$$a_0 + a + ip, \tag{16}$$

where i is the loop number, α_0 is the initial value of the variable address, α is the value of the initial (first) address shift, and p is the readdressing interval.

The possibility of executing an initial shift by a quantity α differing from the readdressing interval is especially convenient for scanning of information stored in different areas of memory. Thus, the information required for a loop depending on a parameter will be:

a) the set of computational operations for the loop in the initial form;

b) specifications for the variable addresses;

c) the pair of numbers (α, p) determining the loop parameter.

The information in a) may be specified with the aid of appropriate computational instructions. To permit the machine to recognize variable address, i.e., the need to form addresses in accordance with Rule (16), an extra bit (the 12th) is provided in each address of the "Kiev" computer. Two operations are used for this purpose: NGO — beginning of group operation, and OGO — end of group operation. The first of these may be used independently, and the other when the F or NGO operations are used.

The NGO operation is coded as follows:

НГО	M	α	k

Here NGO is the operation code; M is a number equal to $\alpha + Np$, where N is the total number of loop repetitions (if some other condition is used for loop termination, then address I will contain the word 11...1); α is the value of the initial shift, and it will have the value indicated above, while k is the number of a certain instruction to which control is to be transferred after termination of the loop (for subroutines, we may indicate the address of the permanently connected location 3146, which contains the code for the jump according to link register instruction).

The functional meaning of the NGO operation is as follows:

1) the constant M characterizing the total number of loops is transferred to the loop register Ц;

2) the constant α, the magnitude of the initial shift for the variable addresses, is transferred to the address register A;

3) the contents of the address register and the loop register are compared, and if they agree, control is transferred to the instruction whose address k is contained in address III of the NGO instruction, i.e., $'A_3 \Rightarrow C$; if this is not the case, control jumps to the instruction with the next number, i.e., $'C + 1 \Rightarrow C$.

There are two ways of changing the readdressing interval — the content of the address register:

a) by readdressing the NGO instruction through an appropriate change in address II. In this case, the program will contain the variable instruction NGO; for each of the loops, provision is made for repeated execution of NGO instruction, and thus a new value for the variable-address modification constant is prepared. This method is convenient in the coding of complex looping processes;

b) by using the OGO instruction, which is coded in the form

ОГО	$4000+p$	k_1	k_2

Here the OGO is the operation code; p is the readdressing interval (the interval p, increased by 4000, is shown in address I); k_1, k_2 are the numbers of certain instructions.

With the OGO instruction:

1) the readdressing interval p is added to the content of the address register: $p+'A \Rightarrow 'A$ (i.e., $'A_1+'A \Rightarrow A$);

2) the contents of the address register and loop register are compared, and where they agree exactly, control is transferred to the instruction whose address k_2 is shown in address III of the OGO instruction (i.e., $'A_2 \Rightarrow C$), while if this is not the case, control is transferred to the instruction whose address k_1 is shown in address II of this instruction (i.e., $'A_3 \Rightarrow C$).

It is important to note that when the OGO instruction is used, the program provides for execution of the NGO instruction for the first loop; when the loop is repeated, the NGO instruction is not repeated, and remains unchanged. Thus by employing the OGO operation together with the NGO operation, we can code looping programs that include parameters into fixed, which is especially convenient for loop processes involving initialization, as well as where the number of loops is known in advance. These operations are widely employed in standard routines.

The following circuitry is used in the control unit to realize the processes described: loop register, address register, address adder, and coincidence unit.

The adder is used to add addresses specified in an instruction to the content of the address register, and during execution of an OGO instruction to add the readdressing interval to the content of the address register.

II. It is easy to see that in the coding of complex looping processes, the availability of one address register permits us to use the OGO operation only for the coding of inner loops, which in turn do not include F or OGO operations. Outer loops must be coated with the aid of NGO instruction readdressing.

To permit wide utilization of the interchangeable built-in memory, which is realized in the form of modules that can be switched into (or out of) the internal passive memory of the computer as needed, the "Kiev" machine has the F group-type operation, which permits us to load any program into passive storage.

The F operation is codes as

Φ	φ		a

Here F is the operation code, and φ, a are the addresses of certain locations.

Both the F and NGO operations prepare the address register A, which is then used to form the variable addresses. In contrast to the NGO operation, however, where the readdressing constant is explicitly coded, the quantity α + ip is now coded in address II of the location whose number φ is specified in address I of the F instruction:

φ			$\alpha + ip$		$('\varphi_{II} = \alpha + ip)$.

i.e., $'\varphi_{II} \Longrightarrow A$.

In addition, when the F operation is executed, address a receives the code contained in location α + ip, whose number is stored in address II of location φ, i.e., $''\varphi_{II} \Longrightarrow 'A_3$. Since the value of group readdressing constant is now no longer explicitly contained in the program, the latter will not change in form. Changing the readdressing constant now reduces to changing the content of a certain address φ. Address II of the Φ instruction is not used.

Thus, outer loops in complicated looping processes may be coded either with the aid of the NGO operation and subsequent readdressing of this operation, or they may be coded in fixed storage with the aid of the F operation. In connection with the fact that the code φ in address I of instruction F may be taken as the address that "specifies" the amount of the shift in the sequence of certain addresses, the F operation has come to be called the operation of calling by address of rank two.

The first function of the OGO operation — changing the content of the address register — can also be used in connection with the F operation.

The F operation is realized in the following manner: the address φ of rank two is transferred to the address register in high-speed storage; the content of this address $\alpha + ip = '\varphi_{II}$ (address of rank one $'\varphi_{II}$) is transferred out of high-speed memory to the instruction register in the control unit (to its free address II); next the code $'\varphi_{II}$ is again transferred to the address register of the OZU from the instruction register. The content of this address ($'\varphi_{II}$) the content based on an address of rank II) is taken from the OZU and held there in the number register. The code from address III of the F instruction is transferred to OZU, and used as a basis for writing the content of the number register $'('\varphi_{II})$.

Example 1. The principal element of a square matrix with side n, whose elements are arranged (along rows or columns) in accordance with the addresses $\alpha + 1$, $\alpha + 2$, ..., $\alpha + n^2$, is stored in the location with number γ.

For the quotes "Kiev" machine, the program will have the form

$N+1$	$+$			γ
$N+2$	НГО	M	α	
$N+3$	$\lvert<\rvert$	4001	γ	$N+5$
$N+4$	$+$	4001		γ
$N+5$	ОГО	4001	$N+3$	N_1

Here $M = \alpha + n^2$; by writing the number 4000 in instructions N + 3 and N + 4, we indicate that there is a unit in the 12th digit position of the address (group-operation indicator); N_1 is the address of the instruction to which control is transferred after the given routine has been run.

Example 2. We are to evaluate the sum

$$S = \sum_{j=1}^{m} \sum_{i=1}^{n} \frac{a_i x_j + b_i y_j}{c_i x_j + d_i y_j}.$$

Let the given values be stored in accordance with the following addresses:

$\alpha+1$	$\alpha+2$	$\dots$	$\alpha+n$	$\beta+1$	$\dots$	$\beta+n$	$\gamma+1$	$\dots$	$\gamma+n$
a_1	a_2	$\dots$	a_n	b_1	$\dots$	b_n	c_1	$\dots$	c_n

$\delta+1$	$\dots$	$\delta+n$	$\varphi+1$	$\dots$	$\varphi+m$	$\varphi+m+1$	$\dots$	$\varphi+2m$
d_1	$\dots$	d_n	x_1	$\dots$	x_m	y_1	$\dots$	y_m

The program will have the form

$N+1$	$+$			γ
$N+2$	НГО	$\varphi + m$	φ	k
$N+3$	СлК	δ	$N+2$	$N+2$
$N+4$	$+$	4001		ω_1
$N+5$	$+$	$4001 + m$		ω_2
$N+6$	НГО	n		
$N+7$	$\times$	$4001 + \alpha$	ω_1	τ_1
$N+10$	$\times$	$4001 + \beta$	ω_2	τ_2
$N+11$	$+$	τ_1	τ_2	τ_1
$N+12$	$\times$	$4001 + \delta$	ω_2	ρ_1
$N+13$	$\times$	$4001 + \gamma$	ω_1	ρ_2
$N+14$	$+$	ρ_1	ρ_2	ρ_1
$N+15$	:	τ_1	ρ_1	τ_1
$N+16$	$+$	τ_1	γ	γ
$N+17$	ОГО	4001	$N+7$	$N+2$

Here <u>k</u> is the address of the instruction to which control is transferred after the program has been run; the location with address δ contains the code

		0001	

The desired sum is obtained in address γ.

<u>Example 3</u>. The sequence of addresses

$$'b+0,\ 'b+1,\ \ldots,\ 'b+35$$

contains the coefficients for seven polynomials of which the first polynomial is of degree one, the second of degree two, etc. The coefficients of the polynomials are stored in order of increasing power. The values of all the polynomials are computed by Horner's method for a fixed value of the variable contained in address ξ, and they are stored in the sequence of addresses

$$'B+0,\ 'B+1,\ \ldots,\ 'B+6,$$

whose contents then form the result obtained by running the algorithm.

The address program may be written in the following form:

$$0 \Longrightarrow\rangle\ ''B,\quad 0 \Longrightarrow\rangle\ 'b;\quad 1 \Longrightarrow\rangle\ \Delta 1;\quad 2 \Longrightarrow\rangle\ \Delta 0$$

$$M\ \ldots\ 0 \Longrightarrow\rangle\ 'B + ''B$$

$$
\begin{gathered}
M1 \ldots {}'({}'B + {}'\delta B) \times {}'\xi + {}'({}'b + {}'\delta b) \Longrightarrow {}'B + {}'\delta B \\
{}'\delta b + 1 \Longrightarrow \delta b \\
P\,\{{}'\delta b < {}'\Delta 1,\ M1 \\
{}'\Delta 0 + 1 \Longrightarrow \Delta 0;\quad {}'\Delta 1 + {}'\Delta 0 \Longrightarrow \Delta 1,\quad {}'\delta B + 1 \Longrightarrow \delta B \\
P\,\{{}'\delta B < 6\}\ M\downarrow!
\end{gathered}
$$

Let us look in more detail at the way in which the F operation is used to program the second and third lines, which contain address functions of rank two. A total of four addresses (B, δB, b, δb) are subject to the operation of two primes. Since these addresses are encountered in the algorithm only in the combinations 'B + 'δB and 'B + 'δb, the two lines indicated are conveniently replaced by the following lines:

$$
\begin{gathered}
M \ldots {}'B + {}'\delta B \Longrightarrow r_1 \\
0 \Longrightarrow {}'r_1 \\
M1 \ldots {}'b + {}'\delta b \Longrightarrow r_2 \\
{}''r_2 \Longrightarrow r_2 \\
{}''r_1 \times {}'\xi + {}'r_2 \Longrightarrow {}'r_1
\end{gathered}
$$

or, in "Kiev" machine codes:

$M \ldots$	+	B	δB	r_1
	Φ	r_1		
	+			4000
$M1 \ldots$	+	b	δb	r_2
	Φ	r_2		r_2
	Φ	r_1		
	×	4000	ξ	r_3
	+	r_3	r_2	4000

Here r_1, r_2, and r_3 are working locations.

REPRESENTATION OF ADDRESS FUNCTIONS IN THE "KIEV" COMPUTER

Let us introduce the concept of the rank R of an address function f:R(f).

1. An address function a that does not contain a prime operation will have a rank R of zero:

$$R(a) = 0.$$

2. Let the rank of a function $\underline{f}$ equal $\underline{s}$. Then the rank of a function $'f$ will equal $R(f) + 1 = s + 1$, i.e.,

$$R('f) = R(f) + 1.$$

3. Any function $O_1(x)$ of a single variable $\underline{x}$, other than a prime function, will have a rank equal to the rank of its argument:

$$R(O_1(x)) = R(x).$$

4. Let $R(a) = s_1$, $R(b) = s_2$. The rank of a function $a\theta b$, where θ is any arithmetic or logical operation in two arguments will equal the largest of the ranks s_1 and s_2:

$$R(a\theta b) = \max(R(a), R(b)).$$

An arbitrary address algorithm may be represented in a form that is equivalent (from the viewpoint of meaning) such that all of the address functions entering into its representation will have rank not exceeding two.

As an example, the address formulas

$${}^{n}\alpha \Rightarrow \beta \quad (n > 3)$$

and

$$\alpha \Rightarrow {}^{m}\beta \quad (m > 2)$$

may be written, respectively, in the form

$$\begin{array}{ll}
{}^{2}\alpha \Rightarrow \beta & {}^{2}\beta \Rightarrow \omega \\
\left.\begin{array}{c} {}^{2}\beta \Rightarrow \beta \\ \vdots \\ {}^{2}\beta \Rightarrow \beta \end{array}\right\} \begin{array}{l} n-3 \\ \text{times} \end{array} & \left.\begin{array}{c} {}^{2}\omega \Rightarrow \omega \\ \vdots \\ {}^{2}\omega \Rightarrow \omega \end{array}\right\} \begin{array}{l} m-2 \\ \text{times} \end{array} \\
{}^{2}\beta \Rightarrow \beta & \alpha \Rightarrow {}'\omega
\end{array}$$

Here and below ω is some working address whose content has no effect on running of the algorithm.

We also note that any address formula

$$a \Rightarrow b,$$

where $\underline{a}$, $\underline{b}$ are address functions of rank $\underline{n}$ and $\underline{m}$, respectively, may be replaced by an equivalent program of two lines

$$a \Longrightarrow \omega$$
$$'\omega \Longrightarrow b$$

A distinguishing feature of address-function representation in the "Kiev" computer consists in the following method of executing operations according to rank two, rank zero, or higher ranks:

1) the components of any (arithmetic or logical) operation taking the form of address functions of second or higher rank are formed solely with the aid of the A-register;

TABLE 1

Representation of Address Functions in "Kiev" Computer Codes

1 Адресная программа	2 Программа в кодах «Киева»			
$'(a+\gamma_1{}'d)\,0\,'(b+\gamma_2{}'d) \Longrightarrow\, \Longrightarrow c+\gamma_3{}'d$	34	d		—
	0	$\gamma_1 \cdot 4000+a$	$\gamma_2 \cdot 4000+b$	$\gamma_3 \cdot 4000+c$
$'(a+\gamma_1{}'d)\,0\,'(b+\gamma_2{}'d) \Longrightarrow\, \Longrightarrow c+\gamma_3{}'d$; $''d \Longrightarrow l$	34	d		l
	0	$\gamma_1 \cdot 4000+a$	$\gamma_2 \cdot 4000+b$	$\gamma_3 \cdot 4000+c$
$'(a+\gamma_1 d)\,0\,'(b+\gamma_2 d) \Longrightarrow\, \Longrightarrow c+\gamma_3 d$	НГО	—	d	—
	0	$\gamma_1 \cdot 4000+a$	$\gamma_2 \cdot 4000+b$	$\gamma_3 \cdot 4000+c$
$d \Longrightarrow A$	НГО	—	d	—
$'A+p \Longrightarrow A$	ОГО	$4000+p$	—	—
${}^2a\,\theta\,{}^2b \Longrightarrow c$	34	a		r
	34	b		—
	0	r	4000	c
3	или			
	34	a		r_1
	34	b		r_2
	0	r_1	r_2	c

Continued

1 Адресная программа	2 Программа в кодах «Киева»				
$'('a+b)\,0$ $'('c+d) \Rightarrow b$	31	a		—	
	01	$4000+b$		r_1	
	31	c		—	
	01	$4000+d$		r_2	
	0	r_1	r_2	b	
4 Формирование содержимого A-регистра по формуле $b+'a+('c+d) \Rightarrow A$	34	c		—	
	01	$4000+d$		r	
	34	a		—	
	27	$4000+b$	—	—	
	34	$4000+r$		—	
! $^n a \Rightarrow c$	34	a		c	
	34	c		c	
		⋮			$n-2$ раза 5
	34	c		c	
$'('('a+a)+b) \Rightarrow c$	34	a		—	
	01	$4000+a$		c	
	34	c		—	
	01	$4000+b$		c	

Continued

1 Адресная программа	2 Программа в кодах «Киева»				
$'('a + '\beta) \Longrightarrow c$	01	a	β	c	
	34	c		c	
$'('a + '\beta + a) \Longrightarrow c$	01	a	β	c	
	34	c		—	
	01	$4000 + a$		c	
4 Формирование содержимого A-регистра по формуле $c + 'a'b$ $('3010 = 2^{-16})$ $('3040 = 2^{-12})$	12	a	3010	r	
	12	b	3040	r_1	
	10	r	r_1	r	
	34	r		—	
	27	$4000 + c$	—	—	
$'a\theta'\beta \Longrightarrow '('a + b) + c$	34	a		—	
	34	$4000 + b$		—	
	θ	a	β	$4000 + c$	
$'a\theta'\beta \Longrightarrow 'a\theta_1'b + c$	θ_1	a	b	r	
	34	r		—	
	θ	a	β	$4000 + c$	
$'a\theta'\beta \Longrightarrow {}^n a$	34	a		r	
	34	r		r	$n - 2$
	⋮				
	34	r		r	
	θ	a	β	4000	

1) Address program; 2) program in "Kiev" codes; 3) or; 4) formation of content of A-register according to the following formula; 5) times.

2) addresses of rank zero can only be components of operations involving addition with the content of the A-register or with zero, where the result is stored according to the A-register.

Table 1 shows the fundamental relationships between address functions and programs in "Kiev" computer language; here a, b, and c represent nonnegative octal integers $\leq$ 3777 (addresses of rank zero), while δ_1, δ_2 and δ_3 represent numbers equal to zero or unity; three dots in the instruction column indicates that any number of instructions can be inserted at this location, provided they do not change the content of the A-register; a dash in an instruction address indicates that the content of this address is not used for the given operation; θ is the symbol for an arbitrary (arithmetic or logical) elementary operation; A represents the modification register (A-register).

ALGORITHM FOR FORMAL TRANSLATION OF ADDRESS FUNCTIONS INTO "KIEV" MACHINE CODES

The special feature of algorithmic languages lies in the fact that in contrast to natural languages each of the syntactical signs (or set of signs) carries its own information as to meaning, which does not depend on relationships with other signs. As it applies to the address language, this feature permits us to give a programming method in the form of a new principle of translator-program operation.

The general operating principle of previously known translator programs (see, for example [7]), including the above-mentioned PP-2 and PP-AK for the "Kiev" computer, consists in searching through the algorithm as written for a representable set of symbols to be used as the basis for formation of one or several computer instructions. Where parentheses are used in representation of formulas, such a representable set will be, for example, a performable operation — the symbol for a two-term operation connecting two adjacent symbols for variables, or the symbol for a one-term operation immediately followed by a variable. On the basis of the representable combination of codes that has been found, the translator program writes into the working-location block that has been set aside the appropriate instruction or group of instructions (depending on the computer address format, etc.), selecting a working location for the result where necessary; in the initial information representation, the computer replaces this combination of symbols by the number of the working location. In one way or another, condensation of the initial information continues, since the number of elements in the initial information will be reduced under such treatment, and the process is repeated until the entire representation is exhausted. The implementation of various improvements (for example, reduction in the number of working locations) causes no difficulties in principle.

The following principle may be proposed for element-by-element translation of address formulas into machine language:

1) to write a program corresponding to a single address formula, a working field of sufficient volume is set aside;

2) the initial information is scanned element-by-element in natural order, just once, and depending on the information element scanned at the given instant, the computer writes into the working field.

The principle of one-shot element-by-element scanning of the information was proposed by Ye.L. Yushchenko [22] in order to realize an algorithm for formal checking of the correctness of unparenthesized and parenthesized representations of formulas. Later, M.M. Bushko-Zhuk proposed to use this principle for translator-program operation.

Let us consider an algorithm of this type for translating address functions into the language of the "Kiev" computer. For symplicity, we shall assume that the address-function representation may include: an address of arbitrary rank, except for addresses of rank zero; arbitrary two-term machine operations (arithmetic or logical); one-term operations with entrance address 0002 and exit address 0003 that can be coded by the addresses of the initial instructions of the subroutines realizing these operations.

We shall use the strict parenthesized representation of the formulas, in which the order of execution of the operations is determined solely by the parentheses (the hierarchy of operations is not considered), and where there are no superfluous parentheses).

We use metalinguistic formulas [13] to describe an address function in strictly parenthesized form. Strings of signs contained within corners < > are metalinguistic variables, whose values are strings of symbols; the signs :: = and | are metalinguistic copulas: the sign :: = represents "equal by definition" while the sign | represents "or." A symbol in a formula that is not a variable or copula will represent itself or a similar class of symbols. The connection of signs or variables indicates the connection of the strings represented.

According to the adopted convention, the fundamental symbols are:

(is an initial parenthesis;

) is a terminal parenthsis;

0_1 represents one-term operations, including the prime operation;

0_2 represents two-term operation.

<u>a</u> represents an address.

Thus,

$< \text{fundamental symbol} > :: = (|)|\ 0_1|0_2|a.$

We first introduce the concept of a <u>component</u>:

$< \text{component} > :: = a|0_1< \text{component} > | < \text{component} > 0_2 < \text{component} >).$

We can now define the strictly parenthesized representation of an address function as follows:

$<$ strictly parenthesized representation of a function $> ::= a | O_1 <$
$<$ component $>$ | $<$ component $> O_2 <$ component $>$.

The formula gives a recursive rule for forming address functions in a strictly parenthesized representation.

The working block set aside for writing of the program is filled from the bottom upward. The instruction components are established as the computer scans the coded information (in the course of operation, certain instructions may be completely formed, while others, possibly including some to be executed earlier, will still not be formed or will be formed only in part). The machine reduces the number of working locations needed as it runs.

We introduce the following definitions:

φ_0 is the specifier for an initial information element;

φ is the specifier for elements of the input information; first $'\varphi = C^{-1}(\alpha)$;

C is the succession operation in the sequence of addresses for the elements of the initial information, the first of which is represented by α (several elements may be placed into one machine address);

ψ is the specifier for the working block set aside for recording of the working-program instructions; the address of the last location in this block is s;

κ is the specifier for the block of working locations for the working program (RP);

r is the first working location of this block;

Σ, μ, and λ are working specifiers.

The specifier Σ is used when parentheses are manipulated. With the aid of this specifier, upon processessing of the initial-parenthesis symbols, in some block of translator-program (PP) working locations that contains k addresses ($\gamma + 1$, $\gamma + 2$, ..., $\gamma + k$) there are formed the appropriate addresses of the Rp instructions; the process of their formation is completed when the machine processes the corresponding paired terminal parenthesis. The length of the block k determines the admissible maximum complexity of parenthesis structure in the expressions for the address functions.

The specifier μ is used to store information to indicate that at the given instant a one-term operation is being programmed – a call to an appropriate subroutine (here a one is placed into μ).

The specifier λ is used to store the address of the Rp instruction which the machine must proceed to program after processing the symbols for prime-ranks exceeding the first or a symbol for a one-term operation.

In addition, we shall also assume that pr f will represent an indicator for the element f. There are the following types of indicators:

ω, the indicator for the end of a function representation;

'adr, the indicator of an address of rank one;

(, an initial parenthesis;

), a terminal parenthesis;

'ne-adr, the indicator of the presence of a prime operation not pertaining directly to an address;

O_1, the indicator for a one-term operation;

O_2, the indicator for a two-term operation;

rab, the indicator for a working location.

In an algorithm, the element indicators ω, 'adr, (,), 'ne-adr, O_1, and O_2 are used as labels.

We use $'\psi_O$, $'\psi_I$, $'\psi_{II}$, $'\psi_{III}$ to represent, respectively, the contents of addresses O, I, II, and III of location ψ. Rectangular formats are used for the "Kiev" computer codes employed for an address program.

We should take special note of the role played by the unconditional-jump address formula $pr^2\varphi$ (third line of the algorithm).

The specifier φ scans the elements of the input information (address algorithm), i.e., at any given instant of algorithm operation, exponent 2φ is the element being processed. Depending on the element indicated, the formula

$$pr\ {}^2\varphi$$

is used as the basis for going to the algorithm line marked off by the appropriate label.

The function $pr\ {}^2\varphi$ can take on one of seven values: ω, 'adr, 'neadr, (,), O_1, or O_2; thus this formula represents a jump to one of the seven algorithm branches, each of which is indicated by an appropriate label.

Algorithm for Element-by-Element Translation of Address Functions into "Kiev" Machine Codes

Initial address mapping

$$\begin{array}{l} {}'\varphi_0 = \alpha \\ \left.\begin{array}{l} {}'(\alpha+1) \\ {}'(\alpha+2) \\ \quad\vdots \\ {}'(\alpha+N) \end{array}\right\} \text{element-by-elemement coded address function} \end{array}$$

Resultant address mapping

$$\left.\begin{array}{l} \quad\vdots \\ {}'(s-2) \\ {}'(s-1) \\ {}'s \end{array}\right\} \text{program for computing value of address function in "Kiev" machine codes.}$$

Algorithm

```
B ... $0 \Rightarrow \mu;\quad 0 \Rightarrow \lambda;\quad r \Rightarrow \varkappa;\quad s \Rightarrow \psi;\quad \gamma + 1 \Rightarrow \Sigma$
H ... $C('\varphi) \Rightarrow \varphi$
      $np^2\varphi$
 ( ... $0 \Rightarrow \mu$
      $P\{'\lambda = 0\}'\psi \Rightarrow{}'\Sigma \downarrow {}'\lambda \Rightarrow{}'\Sigma;\ 0 \Rightarrow \lambda$
      $'\Sigma + 1 \Rightarrow \Sigma$
      $P\{'('\psi_1) \neq 0\}(b$
      $'\varkappa \Rightarrow{}'\psi_1$
      $'\psi - 1 \Rightarrow \psi$
(a ... $'\varkappa \Rightarrow{}'\psi_{111}$
      H
(b ... $P\{np'('\psi_1) = pa\delta\}'\varkappa + 1 \Rightarrow \varkappa$
      $'\varkappa \Rightarrow{}'\psi_{11}$
      $Ц\{'\psi(-1)P\{^2\psi \neq 0\} \Rightarrow \psi\}$
```

```
     $L$ ...
             ($a$
     ) ...  $'\Sigma - 1 \Rightarrow \Sigma$
            $0 \Rightarrow \lambda$
            $''\Sigma \Rightarrow \psi$
            H
   $O_2$ ...  $P\{'\lambda \neq 0\}\ '\lambda \Rightarrow \psi,\ 0 \Rightarrow \lambda$
            ${}^2\varphi \Rightarrow\ '\psi_0$
            $0 \Rightarrow \mu$
            H


PRIME $a$ ...  $P\{'('\psi_1) = 0\}\ {}^2\varphi \Rightarrow '\psi_1 \downarrow {}^2\varphi \Rightarrow '\psi_{11}$
            H
  PRIME   ...  $P\{'('\psi_1) \neq 0\}$ Шб
            $P\{'\mu = 0\}$ Ша
            $0 \Rightarrow \mu;\ |\overline{\underline{\Phi}}| \Rightarrow '\psi_0$
            H
     Ша ...  $'\varkappa \Rightarrow '\psi_1$
            $P\{'\lambda = 0\}\ '\psi \Rightarrow \lambda$
            $'\psi - 1 \Rightarrow \psi$
            Шв
     Шб ...  $P\{np'('\psi_1) = pa\delta\}\ '\varkappa + 1 \Rightarrow \varkappa$
            $'\varkappa \Rightarrow '\psi_{11}$
            $Ц\{'\psi(-1)\ P\{{}^2\psi \neq 0\} \Rightarrow \psi\}$
      $Z$ ...
     Шв ...  $\begin{array}{|c|c|c|c|}\hline \varphi & & & '\varkappa \\ \hline\end{array} \Rightarrow '\psi$
            H
   $O_1$ ...  $P\{'('\psi_1) \neq 0\}\ O_1b$
            $|\overline{\underline{0003}}| \Rightarrow '\psi_1$
            $P\{'\lambda = 0\}\ '\psi \Rightarrow \lambda$
  $O_1a$ ...  $'\psi - 1 \Rightarrow \psi$
            $\begin{array}{|c|c|c|c|}\hline УПП & 3026 & '\psi + 1 & {}^2\varphi \\ \hline\end{array} \Rightarrow '\psi$
!
            $'\psi - 1 \Rightarrow \psi$
            $\begin{array}{|c|c|c|c|}\hline + & & & 0002 \\ \hline\end{array} \Rightarrow '\psi$
            $1 \Rightarrow \mu$
            H
  $O_1b$ ...  $P\{('('\psi_1) = 0003) \vee np'('\psi_1) = pa\delta\}\ '\varkappa + 1 \Rightarrow \varkappa$
            $'\varkappa \Rightarrow '\psi_{11}$
            $Ц\{'\psi(-1)\ P\{{}^2\psi \neq 0\} \Rightarrow \psi\}$
      $Y$ ...
            $\begin{array}{|c|c|c|c|}\hline + & 0003 & & '\varkappa \\ \hline\end{array} \Rightarrow '\psi$
            $O_1a$
      $\omega$ ... В
```

Let us consider algorithm operation using as an example programming of the address function

$$({}''a - \sin\cos' ({}'a + {}'b)) \times \sin'' c\omega,$$

where ω is the termination indicator.

We shall note the change in the address mapping at each step of algorithm operation.

First $'\mu = 0$; $'\lambda = 0$; $'\varkappa = r$; $'\psi = s$; $'\Sigma = \gamma + 1$; $'(s - i) = 0$, where $i = 0, 1, 2, \ldots$.

1. The initial-parenthesis code is processed; $pr^2\varphi$ = (:

$$'(\gamma + 1) = s$$
$$'\Sigma = \gamma + 2$$
$$'s = \begin{array}{|c|c|c|c|}\hline \quad & r & \quad & \quad \\ \hline\end{array}$$
$$'\psi = s - 1$$
$$'(s - 1) = \begin{array}{|c|c|c|c|}\hline \quad & \quad & \quad & r \\ \hline\end{array}$$

2. The code "'"; $pr^2\varphi$ = PRIME is processed:

$$'(s - 1) = \begin{array}{|c|c|c|c|}\hline \quad & r & \quad & r \\ \hline\end{array}$$
$$'\lambda = s - 1$$
$$'\psi = s - 2$$
$$'(s - 2) = \begin{array}{|c|c|c|c|}\hline \Phi & \quad & \quad & r \\ \hline\end{array}$$

3. The code 'a; $pr^2\varphi$ = PRIME $\underline{a}$ is processed:

$$'(s - 2) = \begin{array}{|c|c|c|c|}\hline \Phi & a & \quad & r \\ \hline\end{array}$$

4. The code "—"; $pr^2\varphi = O_2$ is processed:

$$'\psi = s - 1$$
$$'\lambda = 0$$
$$'(s - 1) = \begin{array}{|c|c|c|c|}\hline - & r & \quad & r \\ \hline\end{array}$$

5. The code sin; $pr^2\varphi = O_1$ is processed:

$$'\varkappa = r + 1$$
$$'(s - 1) = \begin{array}{|c|c|c|c|}\hline - & r & r+1 & r \\ \hline\end{array}$$

$'\psi = s - 3$

$'(s-3) =$

+	0003		r + 1

$'\psi = s - 4$

$'(s-4) =$

упп	3026	s — 3	sin

$'\psi = s - 5$

$'(s-5) =$

+			0002

$'\mu = 1$

6. The code cos; $pr^2\varphi = O_1$ is processed:

$'(s-5) =$

+	0003		0002

$'\lambda = s - 5$
$'\psi = s - 6$

$'(s-6) =$

упп	3026	s — 5	cos

$'\psi = s - 7$

$'(s-7) =$

+			0002

$'\mu = 1$

7. The code "'"; $pr^2\varphi$ = PRIME is processed:

$'\mu = 0$

$'(s-7) =$

Ф			0002

8. The code "("; $pr^2\varphi$ = (is processed:

$'(\gamma + 2) = s - 5$
$'\lambda = 0$
$'\Sigma = \gamma + 3$

$'(s-7) =$

Ф	r + 1		0002

$'\psi = s - 8$

$'(s-8) =$

			r + 1

9. The code "'"; $pr^2\varphi$ = PRIME a is processed:

$'(s-8) =$

	a		r + 1

10. The code "+"; $pr^2\varphi = O_2$

$'(s-8) =$	+	a		$r+1$

11. The code "'v"; $pr^2\varphi$ = PRIME $\underline{a}$ is processed:

$'(s-8) =$	+	a	b	$r+1$

12. The code ")"; $pr^2\varphi$ =) is processed:

$$''\Sigma = \gamma + 2$$
$$'\psi = s - 5$$

13. The code ")"; $pr^2\varphi$ =) is processed:

$$'\Sigma = \gamma + 1$$
$$'\psi = s$$

14. The code "x"; $pr^2\varphi = O_2$ is processed:

$'s =$	×	r		

15. The code sin; $pr^2\varphi = O_1$ is processed:

$$'x = r + 2$$

$'s =$	×	r	$r+2$	

$$'\psi = s - 9$$

$'(s-9) =$	+	0003		$r+2$

$$'\psi = s - 10$$

$'(s-10) =$	УПП	3026	$s-9$	sin

$$'\psi = s - 11$$

$'(s-11) =$	+			0002

$$'\mu = 1$$

16. The code "'"; $pr^2\varphi$ = PRIME is processed:

$$'\mu = 0$$

$'(s-11) =$	Φ			0002

17. The code "'s"; $pr^2\varphi$ = PRIME a is processed:

'(s − 11) =

Ф	c		0002

In "Kiev" computer language, the entire program looks like this:

s − 11	Ф	c		0002
s − 10	УПП	3026	s − 9	sin
s − 9	+	0003		r + 2
s − 8	+	a	b	r + 1
s − 7	Ф	r + 1		0002
s − 6	УПП	3026	s − 5	cos
s − 5	+	0003		0002
s − 4	УПП	3026	s − 3	sin
s − 3	+	0003		r + 1
s − 2	Ф	a		r
s − 1	−	r	r + 1	r
s	×	r	r + 2	

Chapter 4

SUBROUTINES FOR "KIEV" COMPUTER

GENERAL PRINCIPLE OF STANDARD-BLOCK CONSTRUCTION AND SUBROUTINE METHOD

Analysis of address programs shows that the availability in TsAM of algorithmic operations permitting calling by addresses of rank two simplifies the algorithmic language and thus considerably broadens the scope of the computer. It must be noted that the utilization of such operations simplifies programs, while for machines having fixed storage, it becomes possible to store arbitrary programs and to ease the task of creating a library of standard routines and of using them in the course of problem solution.

As we know, where an extensive library is present, it is possible to construct complex programs almost completely from library routines, using so-called compiler or interpreter systems [12].

The creation of a library of subroutines for the "Kiev" computer provides a comprehensive set of standard routines, for example, for evaluating scalar and vector functions of scalar, vector, and matrix arguments. The availability of library simplifies work on programming automation, checking, loading, and the debugging of individual program segments, and thus reduces the probability of programming errors and yields a considerable gain in time.

The subroutine library for the "Kiev" computer includes a set of programs placed into the passive storage unit (PZU), into the high-speed memory, and onto magnetic drums. Its distinguishing characteristic is the presence of interchangeable built-in blocks, which may be connected into machine passive storage as needed.

One part of the standard subroutine such as, for example, sin x, ln x, arc tan x, e^x, etc., as well as certain constants, are wired into fixed PZU blocks. The interchangeable built-in memory contains universal translator programs; subroutines for problems of linear algebra; subroutines implementing floating-point operation, etc. In addition, here we also have programs for evaluating exactly the same function with different degrees of computational accuracy, as well as different amounts of storage and running time. The latter makes it possible to make opti-

TABLE 2

Fixed Built-In Constants

1 Восьмеричный номер ячейки	2 Содержимое в 8 й системе	3 Примечания
3010	00 0001 0000 0000	2^{-16}
3011	00 0000 0001 0000	2^{-28}
3012	00 0000 0000 0001	2^{-40}
3013	00 0001 0001 0001	
3014	00 0001 0001 0000	
3015	00 0000 0001 0001	
3016	00 0001 0000 0001	
3017	00 0002 0000 0000	2^{-15}
3020	00 0000 0002 0000	2^{-27}
3021	00 0000 0000 0002	2^{-39}
3022	00 7777 0000 0000	
3023	00 0000 7777 0000	
3024	00 0000 0000 7777	
3025	37 0000 0000 0000	$-\frac{15}{16}$
3026	20 0000 0000 0000	-0
3027	37 0000 7777 7777	
3030	00 7777 7777 7777	
3031	37 7777 0000 7777	
3032	37 7777 7777 0000	
3033	00 7777 0000 7777	
3034	00 0000 7777 7777	
3035	17 7777 7777 7777	Машинная единица 4 $1-2^{-40}$
3036	37 7777 7777 7777	$-(1-2^{-40})$
3037	04 6420 2324 1220	lg2
3040	00 0020 0000 0000	2^{-12}
3041	00 0000 0020 0000	2^{-24}
3042	10 0000 0000 0000	2^{-1}
3043	01 0000 0000 0000	2^{-2}
3044	02 0000 0000 0000	2^{-3}
3045	01 0000 0000 0000	2^{-4}
3046	00 2000 0000 0000	2^{-6}
3047	00 0000 0000 0020	2^{-36}
3050	12 0000 0000 0000	$\frac{10}{16}$
3051	13 0562 0577 3722	ln 2
3052	01 4631 4631 4632	0,1
3053	00 1217 2702 4366	0,01
3054	00 0101 4223 3514	0,001
3055	03 1463 1463 1463	$0{,}2 = \frac{1}{5}$

Continued

3056	04 6314 6314 6315	0,3
3057	14 6314 6314 6315	$0{,}8 = \frac{4}{5}$
3060	00 0203 0446 7230	$0{,}002 = \frac{1}{500}$
3061	02 4365 6050 7534	$0{,}16 = \frac{4}{25}$
3062	05 0753 4121 7270	0,32
3063	05 0574 6033 3447	$\frac{1}{\pi}$
3064	11 0156 5650 1025	$\frac{1}{\sqrt{\pi}}$
3065	05 7055 2615 4264	$\frac{1}{e}$
3066	14 4417 6652 1042	$\frac{\pi}{4}$
3067	12 1371 4066 7116	$\frac{2}{\pi}$
3070	13 2404 7463 1772	$\frac{\sqrt{2}}{2}$
3071	11 1715 1642 6202	$\frac{1}{\sqrt{3}}$
3072	16 3765 6134 5604	$\frac{e}{3}$
3073	06 3011 0514 7521	$\frac{1}{\sqrt{2\pi}}$
3074	12 6770 2505 4243	$\frac{e}{4}$
3075	05 2525 2525 2525	$\frac{1}{3}$
3076	01 4760 1366 1043	$\frac{1}{\pi^2}$
3077	06 7455 7305 2237	$M = \log_{10} e$

1) Octal number of location; 2) content in base-8 system; 3) remarks; 4) machine unit.

mum use of the computer in the solution of concrete problems.

Many particular problems appear in connection with the use of standard routines.

Depending on the degree of diversity of the programs to be solved with the aid of a standard routine, in addition to the direct input data, the standard routine will be furnished with a given number of different auxiliary parameters (dimensions of vectors, matrices, tables, orders of differential-equation systems to be integrated, etc.). As a rule, all of the input information for a standard routine is stored in fixed memory areas, and this often involves the occupation of considerable amounts of high-speed memory (in the case of multiposition programs), and takes much time for the transfers of data from working locations into standard subroutine locations and, conversely, for transfers from standard locations into working locations.

Moreover, the various auxiliary parameters may include information on the storage of initial data in machine memory and as a rule this involves modification of the programs for the given parameters.

The use of operations based on addresses of rank two in the "Kiev" machine permits us to eliminate modification of standard routines for new parameters and, in particular, makes it possible to store any standard routine in fixed memory and, moreover, enables us to minimize requirements for standardization of storage allocation for initial information in computer high-speed memory. For the majority of standard routines (and, in principle, for all such routines) it is sufficient to use one or two fixed addresses where we store information on the allocation of initial numerical blocks. In the simplest case of evaluation of one- and two-term functions thesese addresses (let us call them φ_0 and φ_1) will contain the arguments or values of the functions.

We make the following assumptions for the specification of more complicated forms of information:

A. Let the initial data for a subroutine be a vector; then its components a_1, ..., a_n are stored in the address sequence

$$C\alpha,\ (C^2)\alpha,\ \ldots,\ (C^n)\alpha, \tag{17}$$

where C is some successor operation in the set of addresses (here a machine address may correspond to several addresses or vice versa). For the most frequently employed successor operation, defined by the relationship $C\alpha = \alpha + 1$, we have instead of (17) the sequence

$$\alpha+1,\ \alpha+2,\ \ldots,\ \alpha+n. \tag{18}$$

In addition, on the basis of the address α, called the governing address of the sequence, the machine will always store information on the vector in the form of its dimension n (for the "Kiev" machine, quantities of this type are best stored in location second addresses).

The sequence of addresses

$$\alpha,\ C\alpha,\ \ldots,\ (C^{'\alpha})\alpha$$

will be called an α-sequence with successor operation C (vector type).

It is sufficient to assign to a standard routine at a fixed area φ_0 and address, which we shall call the governing address of sequence (17):

$$'\varphi_0 = \alpha \qquad (''\varphi_0 = '\alpha = n).$$

Thus, information on the vector is furnished on the basis of the content of the single-address φ_0.

B. Information on a square matrix; the matrix components a_{ij} are stored in the address sequence

$$C\alpha,\ (C^2)\alpha,\ \ldots,\ (C^{(i-1)n+j})\alpha$$

by rows or by columns depending on the requirements of the problem; information on the matrix is specified in the form of a single number (the content of location α) -- size n of the matrix.

We shall also call the sequence

$$\alpha,\ C\alpha,\ \ldots,\ (C^{('\alpha)^2})\alpha$$

an α-sequence (square-matrix type).

The standard routine is furnished with information only in the form of the content of address φ_0, which equals the address of the governing element of the sequence:

$$'\varphi_0 = \alpha;$$
$$^2\varphi_0 = n.$$

C. Information on a rectangular matrix; the addresses α and Cα are used as the basis for specifying the dimensions m and n of the matrix, while addresses $(C^2)\alpha$, $(C^3)\alpha$, ..., $(C^{mn+1})\alpha$ are used as the basis for storage of the matrix elements by rows or by columns, depending on the requirements of the standard routine.

We shall call a sequence of this type and α-sequence (rectangular-matrix type). As before, information is furnished to the standard routine on the sequence in the form of the content of a single address — address of the governing element of the sequence:

$$'\varphi_0 = \alpha;$$
$$^2\varphi_0 = m;$$
$$'('\varphi_0 + 1) = n.$$

D. The initial information for the standard routine is a square matrix whose elements are vectors of different dimension n_{ij}. The components of the vectors — the matrix elements — are stored in type A sequences. Let the governing addresses of the sequences be

$$\alpha_{ij} \qquad ('\alpha_{ij} = n_{ij}).$$

The quantities α_{ij} are stored, in turn, in a type-B sequence

$$\alpha,\ C\alpha,\ \ldots,\ (C^{('\alpha)^2})\alpha.$$

E. The initial information for a subroutine is a triangular (symmetric) nth-order matrix. The information is specified by rows (or columns) in the form of a sequence

$$\alpha,\ C\alpha,\ \ldots,\ (C^{\frac{('\alpha)^2+'\alpha}{2}})\alpha,$$

where the order n of the matrix is specified in accordance with address α.

As in the preceding cases, information on a block is prescribed as the content of a single address φ_0 used as the basis for storing the address of the governing element of the sequences. All of this makes clear the general principle for construction of standard information blocks and the method for developing information on them.

As a rule, each standard routine terminates with the instruction

PRV, whose function consists in transferring control to the instruction whose address is contained in the link register. The content of the link register is prepared appropriately in the call to the subroutine. With the "Kiev" computer, in virtue of the presence of just a single link register, this method of going to subroutines is possible only for programs that do not themselves contain calls to subroutines. In other cases, linkage between subroutines and routines is realized with the aid of a previously prepared unconditional-jump indicator. Depending on the level of the transfer to the subroutines, fixed addresses can be used.

SUBROUTINES IN FIXED BUILT-IN STORAGE (PSP)

For all routines for one-term elementary functions, the entrance address, i.e., the location that must contain the argument $\underline{x}$ prior to the call to the subroutine will be 0002; the exit address, i.e., the location into which the subroutine will write the value of the desired function will be 0003. The routine for ln x is an exception; this is not a one-term function; to evaluate ln x, the argument $\underline{x}$ is first transferred into location 0002 while the result appears in two locations: 0003 contains the value of the function and 0004 the scale factor.

All PSP routines terminate with an indicator for a jump on the basis of the link register (PRV), and this must be taken into account in programming.

Frequently employed constants are placed into the PSP. Table 2 shows a list of them. Table 3 gives a list of PSP subroutines.

Program 1

[2 → 10; conversion from binary number system into decimal system]

```
2→10...3100 01 0000 0000 0003      3105 01 0004 0003 0003
          1 26 0012 0000 0000         6 12 0004 3047 0004
          2 11 0002 3050 0002         7 02 0002 0004 0002
          3 10 3047 0002 0004      3110 12 0002 3045 0002
          4 12 0003 3045 0003         1 27 4001 3102 3146
```

TABLE 3

Subroutines for Interchangeable Built-In Memory

1 Название программы	2 Область изменения аргумента	3 Аргумент x, результат y	4 Начальная команда	5 Рабочие ячейки	Количество команд и констант	Точность десятичных знаков 7
8 Перевод 2→10	(—1,1)	'0002 = x '0003 = y	3100	0004	10	10
$\frac{1}{2}\sin x$	(—1,1)	'0002 = x '0003 = y	3152	0004	9	7—8
$\frac{1}{2}\cos x$	(—1,1)	'0002 = x '0003 = y	3147	0004	12	7—8
$\frac{1}{2}\sin x$	(—2π; 2π)	'0002 = $\frac{x}{2\pi}$ '0003 = y	3264	0004	20	6—7
$\frac{1}{2}\cos x$	(—2π; 2π)	'0002 = $\frac{x}{2\pi}$ '0003 = y	3217	0004—0005	19	6—8
$\frac{1}{n}\ln x$	(0,1)	'0002 = x '0003 = y '0004 = $\frac{1}{n}$	3116	0004—0006	29	6—8
$\sqrt{a}$	(2^{-32},1)	'0002 = a '0003 = y	3163	0004	6	7—8
$\frac{1}{4}e^x$	(—1,1)	'0002 = x '0003 = y	3202	0004—0006	13	8—9
$\frac{1}{\pi}\arcsin x$	(—1,1)	'0002 = x '0003 = y	3242	0004—0006	17	7—8
$\frac{1}{\pi}\arccos x$	(—1,1)	'0002 = x '0003 = y	3244	0004—0006	16	7—8

1) Name of program; 2) argument range of variation; 3) argument x, result y; 4) initial instruction; 5) working locations; 6) number of instructions and constants; 7) accuracy in decimal digits.

Argument '0002 (binary number); result '0003 (decimal number); initial instruction 3100.

In order to evaluate $\frac{1}{2}\sin x\left(\frac{1}{2}\cos x\right)$ for $-\frac{\pi}{2} < x < \frac{\pi}{2}$ the polynomial of best approximation

$$\frac{1}{2}\sin x = \frac{1}{2}\sin\frac{\pi}{2}y = ((((C_9 y^2 + C_7)y^2 + C_5)y^2 + C_3)y^2 + C_1)y.$$

is used where

$$C_9 = 0{,}000\,079\,742\,095;\quad C_7 = -0{,}000\,233\,688\,278;$$
$$C_5 = 0{,}03\,984\,463\,964;\quad C_3 = -0{,}32\,298\,185\,553;$$
$$C_1 = 0{,}785\,398\,159\,235.$$

Using this same polynomial, the machine evaluates $\frac{1}{2}$ cos x with preliminary reduction of the argument in accordance with the formula

$$\bar{x} = \frac{\pi}{2} - |x| \text{ or } \bar{y} = 1 - |y|.$$

<u>Program 2 and 3</u>

```
                   1                               2
         [вычисление y = 1/2 sin x  и  y = 1/2 cos x (— 1 < x < 1)]
          3172 [illegible] 0004 7553 6722       3175 25 1256 7405 5264
             3 [illegible] [illegible] 1146 1443   6 14 4417 6651 0101
             4 [illegible] [illegible] 2127 5453   7 20 0000 0000 0001
   cos ...3147 11 0002 3067 0002                3155 01 3172 0000 0003
          3150 06 3035 0002 0002                   6 11 0003 0004 0003
             1 05 0000 0000 3153                   7 01 0003 7173 0003
   sin ...   2 11 0002 3067 0002                3160 27 4001 3156 3161
             3 11 0002 0002 0004                   1 11 0003 0002 0003
             4 [illegible] 0004 0000 0000          2 32 0000 0000 0000
```

1) Evaluation; 2) and.

Argument '0002 = x; result '0003 = $\frac{1}{2}$ sin x ($\frac{1}{2}$ cos x accordingly); initial instructions 3152 for $\frac{1}{2}$ sin x and 3147 for $\frac{1}{2}$ cos x.

<u>Program 4</u>

```
                       1
                  [вычисление 1/2 sin x   (— 2π < x < 2π)]
   sin ...3264 04 0000 0002 3266        3272 02 0000 0002 0002
             5 01 3035 0002 0002           3 04 0002 3043 3275
             6 04 0002 3042 3273           4 02 3042 0002 0002
             7 02 3035 0002 0002           5 12 0002 3241 0002
          3270 04 0002 3043 3272           6 04 0000 0000 3153
             1 02 3042 0002 0002           7 22 0003 0003 3146
```

1) Evaluation.

Argument '0002 = x/2π; result '0003 = $\frac{1}{2}$ sin x; initial instruction 3264.

Program 5

[вычисление $\frac{1}{2}\cos x$ $(-2\pi < x < 2\pi)$]

```
cos ...3217 05 0002 3042 3221     3231 11 0003 0004 0003
       3220 06 3035 0002 0002        2 01 0003 7173 0003
 1        1 05 0002 3043 3237        3 27 4001 3231 3234
          2 06 3042 0002 0002        4 11 0003 0002 0003
          3 01 3026 0000 0005        5 14 0005 0003 0003
          4 12 0002 3241 0002        6 32 0000 0000 0000
          5 06 3035 0002 0002        7 01 0000 0000 0005
          6 11 0002 0002 0004     3240 04 0000 0000 3224
          7 26 0004 0000 0000        1 04 0000 0000 0001
       3230 01 3172 0000 0003
```

1) Evaluation of.

Argument '0002 = $x/2\pi$; result '0003 = $\frac{1}{2}$ sin x; initial instruction 3217.

Evaluation of ln x is also accomplished on the basis of a polynomial of best approximation in the following manner.

The function ln x is evaluated by means of the formula

$$\ln x = M \log_2 x,$$

where M = 0.6931 471 806 = '3051 is the conversion modulus between base-two logarithms and natural logarithms;

$$\log_2 x = 4(C_1\tau + C_2\tau^3 + C_3\tau^5 + C_4\tau^7);$$

$$\tau = \frac{x-1}{x+1};$$

$$C_1 = 0{,}721\,347\,518\,185;$$
$$C_2 = 0{,}240\,450\,190\,571;$$
$$C_3 = 0{,}144\,146\,085\,514;$$
$$C_4 = 0{,}108\,564\,937\,823.$$

This polynomial ensures evaluation of $\log_2$ x on the segment $\left[\frac{1}{\sqrt{2}}, \sqrt{2}\right]$ with an accuracy of within $\varepsilon = 0{,}000\,000\,000\,005$.

For x = < $1/\sqrt{2}$ an n is selected so that

$$\frac{1}{\sqrt{2}} < (\sqrt{2})^n x < 1,$$

and the relationship

$$\ln x = M \log_2 x = M\left[\log_2 (\sqrt{2})^n x - \frac{n}{2}\right].$$

is used. Since when $\frac{1}{(\sqrt{2})^{n+1}} < x < \frac{1}{(\sqrt{2})^n}$ we have $n < 2|\log_2 x| < n+1$, while $|\ln x| < |\log_2 x|$, then for x the scale factor 1/n + 2 is introduced for this segment. The calculation is carried out Horner's method:

$$4(((C_4\tau^2 + C_3)\tau^2 + C_2)\tau^2 + C_1)\tau.$$

Addresses 3112-3115 contain the coefficients C_4, C_3, C_2, and C_1, respectively.

<u>Program 6</u>

1 (вычисление $\ln x$)

```
         3112 01 5712 7226 4561     3131 12 0006 0002 0002
            3 02 2346.6040 1441        2 11 0002 0002 0006
            4 03 6616 1114 4322        3 12 0002 3043 0002
            5 13 4252 1661 7650        4 26 0003 0000 0000
ln ...      6 01 3020 0000 0005        5 01 3112 0000 0003
            7 05 3070 0002 3123        6 11 0003 0006 0003
         3120 12 0002 3070 0002        7 01 0003 7113 0003
            1 01 0005 3011 0005     3140 27 4001 3136 3141
            2 05 0000 0000 3117        1 11 0003 0002 0003
            3 12 3011 0005 0004        2 11 0003 0004 0003
            4 02 0005 3020 0006        3 11 0005 3042 0005
            5 12 0006 0005 0005        4 02 0003 0005 0003
            6 11 0002 3042 0002        5 11 0003 3051 0003
            7 02 0002 3042 0006        6 32 0000 0000 0000
         3130 01 0002 3042 0002
```

1) Evaluation of.

Argument '0002; function '0003; scale factor '0004; initial instruction 3116.

Evaluation of $y = \sqrt{a}$ for $2^{-22} < a < 1$ is performed in accordance with the following scheme:

$$y_{n+1} = y_n + \Delta y_n;$$

$$\Delta y_n = \frac{1}{2}\left(\frac{a}{y_n} - y_n\right);$$

$$y_0 = 1.$$

Program 7

(evaluation of $\sqrt{a}$)

```
√‾ ...3163 01 0000 3035 0003        3167 01 0003 0004 0003
         4 12 0002 0003 0004        3170 05 3047 0004 3164
         5 02 0004 0003 0004           1 32 0000 0000 0000
         6 11 0004 3042 0004
```

Argument '0002 = a; result '0003 = $\sqrt{a}$; initial instruction 3163.

To evaluate $(1/4)e^{-x}$ for $-1 < x < 1$, we use the continued fraction

$$\frac{1}{4}e^{x} = \frac{1}{4} + \cfrac{\frac{x}{4\cdot 16}}{\frac{1}{16} - \frac{x}{2\cdot 16} + \cfrac{\frac{x^2}{4\cdot 16^2}}{\frac{3}{16} + \cfrac{\frac{x^2}{4\cdot 16^2}}{\frac{5}{16} + \cfrac{\frac{x^2}{4\cdot 16^2}}{\frac{7}{16} + \cfrac{\frac{x^2}{4\cdot 16^2}}{\frac{9}{16} + \cfrac{\frac{x^2}{4\cdot 16^2}}{\frac{11}{16} + \cfrac{\frac{x^2}{4\cdot 16^2}}{\frac{13}{16}}}}}}}}$$

Program 8

(evaluation of $1/4e^{x}$ for $-1 < x < 1$)

```
e^x ...3202 06 3025 3044 0004        3211 05 3044 0004 3206
          3 01 0000 0004 0005           2 02 0005 0002 0005
          4 11 0002 3201 0002           3 11 0002 3042 0002
          5 11 0002 0002 0006           4 12 0002 0005 0002
          6 12 0006 0005 0005           5 01 0002 3043 0003
          7 02 0004 3044 0004           6 32 0000 0000 0000
       3210 01 0004 0005 0005
```

Argument '0002 = x; result '0003 = $1/4e^{x}$; initial instructions 3202.

To evaluate $\frac{1}{\pi}\arccos x$ and $\frac{1}{2}\arcsin x\,(-1 \leqslant x \leqslant 1)$ we use the "digit-by-digit" method. We use the formula

$$\varphi = \varphi_0 = \frac{\pi}{2} z_0,$$

to find the value of φ = arc cos x; here $z_0 = \alpha_0, \alpha_1, \ldots, \alpha_i \ldots$ $(0 \leqslant z_0 \leqslant 2;$ the α_i are binary digits).

If $x_0 = x > 0$, then $\cos\varphi > 0$, and φ is the angle in quadrant I, i.e., $0 \leq \varphi \leq \pi/2$; this means that $z < 1$, which is clear from the relationship $\varphi = \pi/2z_0$ and $\alpha_0 = 1$.

If $x < 0$, then $\cos \varphi < 0$ and φ is an angle in quadrant II, i.e., $\pi/2 \underline{\not\leq} x \underline{\not\leq} \pi$ and $z_0 > 1$. This means that $\alpha_0 = 1$.

We also compute x_{i+1} in order to determine the quadrant in which angle φ lies. If $x_i > 0$, then $\alpha_i = 0$ and $x_{i+1} = 2x_i^2 - 1$. If $x_i < 0$, then $\alpha_i = 1$ and $x_{i+1} = 1 - 2x_i^2$.

We compute $(1/\pi)$ arc sin x from the formula

$$\arcsin x = \frac{\pi}{2} - \arccos x.$$

Programs 9 and 10

1 [вычисление $\frac{1}{\pi}\arccos x$ и^2 $\frac{1}{\pi}\arcsin x\,(-1 \leqslant x \leqslant 1)$]

```
arcsin ... 3242 02 0000 0000 0004      3253 14 0003 0006 0003
              3 05 0000 0000 3245         4 02 3042 0005 0002
arccos ...    4 01 0000 0000 0004         5 12 0002 3263 0002
              5 01 0000 0000 0003         6 10 3042 0006 0006
              6 01 3042 0000 0006         7 05 3012 0006 3247
              7 11 0002 0002 0005      3260 31 0004 3262 3261
           3250 31 0002 3251 3253         1 02 3042 0003 0003
              1 02 0005 3042 0002         2 32 0000 0000 0000
              2 05 0000 0000 3255         3 10 0000 0000 0001
```

1) Evaluation of; 2) and.

Argument '0002 = x; result '0003 = $(1/\pi)$ arc cos x [$(1/\pi)$ arc sin x), accordingly]; the initial instruction for $(1/\pi)$ arc sin x is 3242 while for $(1/\pi)$ arc cos x it is 3244.

SUBROUTINES IN INTERCHANGEABLE BUILT-IN STORAGE (SSP)

The interchangeable built-in memory is realized in the form of blocks, each of which bears a number from 1 to 5 and contains 100 octal (64 decimal) codes.

Block programs are so grouped that those subroutine complexes that most probably will be required simultaneously in problem solution can be inserted as needed. Thus, for example, programs of linear algebra are located in blocks with different numbers so that they may be inserted at the same time. After inclusion of the relative-transfer operation in the operation set of the "Kiev" computer, we can insert built-in storage blocks in arbitrary sequence (here fixed addresses must be left for constants alone).

As an example, let us describe two blocks of the interchangeable built-in memory.

Block No. 1. Elementary Functions

One of the sets of block No. 1 (addresses 3300-3377) of the SSP contain the following routines:

1. Extraction of square root (instructions 3300-3317) for values of the argument in the range $0 \leq a < 1$. Initial instruction 3300. As usual, address 0002 contains the argument and address 0003 receives the results.

2. Scalar product of two vectors (instructions 3200-3333). Initial instruction 3320. Addresses 0002 and 0004 are vector specifiers; the result is placed into address 0003. The two vectors are given in the form of α-sequences. In this chapter, we speak in all cases of sequences with successor operation +1).

3. Multiplication of square matrix by vector (instructions 3334-3357). Initial address 3334. Location 0002 contains the specifier of the α-sequence used to give the matrix by rows; location 0004 contains the specifier of the sequence used to give the vector; location 0003 contains the specifier of the vector result.

4. Evaluation of $(1/4)\ a^4$ (instructions 3360-3374). Initial instruction 3360. Since this routine uses the routine for ln, we use '0007 = a; '0010 = x; '0003 = $(1/4)a^x$.

5. The last three addresses are used to store constants; 3375 = $= 2^{-14}$; 3376 = 0.4; 3377 = the constant for shifting by one address $(14 \cdot 2^{-40})$.

Interchangeable Built-In Block No. 1

```
√‾ ... 3300 01 0000 0000 0003
          1 05 0002 0000 3317
          2 01 3035 0000 0003
          3 16 0002 3035 3317
          4 01 3043 0000 0005
          5 05 3043 0002 3311
          6 12 0002 3043 0002
          7 11 0005 3042 0005
       3310 05 0000 0000 3305
          1 12 0002 0003 0004
          2 02 0004 0003 0004
          3 11 0004 3042 0004
          4 01 0003 0004 0003
          5 05 3021 0004 3311
          6 11 0003 0005 0003
          7 32 0000 0000 0000
(x, y) ... 3320 01 0000 0000 0003
          1 01 0002 3011 0005
          2 01 0004 3011 0006
          3 34 0002 0000 0007
          4 34 0006 0000 0010
          5 34 0005 0000 0000
          6 11 4000 0010 0011
          7 01 0003 0011 0003
       3330 01 0005 3011 0005
          1 01 0006 3011 0006
          2 02 0007 3011 0007
          3 31 0007 3324 3146
          4 34 0002 0000 0006
          5 01 0002 4000 0007
          6 01 0006 0003 0005
          7 01 0002 3011 0010
       3340 01 0003 3011 0012
          1 01 0004 3011 0011
          2 01 0000 0000 0015
          3 34 0010 0000 0013
          4 34 0011 0000 0000
          5 11 0013 4000 0014
          6 01 0014 0015 0015
          7 01 0010 3011 0010
       3350 01 0011 3011 0011
          1 05 0010 0007 3343
          2 34 0012 0000 0000
          3 01 0015 0000 4000
          4 01 0012 3011 0012
          5 01 0007 0006 0007
          6 05 0012 0005 3341
          7 32 0000 0000 0000
1/4 a^x ... 3360 01 0000 3035 0011
          1 01 0000 0007 0002
          2 30 3026 3363 3116
          3 11 0003 0010 0012
          4 05 0012 0004 3370
          5 01 0012 0004 0012
          6 11 0011 3065 0011
          7 05 0000 0000 3364
       3370 12 0012 0004 0012
          1 01 0000 0012 0002
          2 30 3026 3373 3202
          3 11 0003 0011 0003
          4 05 0000 0000 0001
          5 00 0004 0000 0000
          6 06 3146 3146 3146
          7 00 0000 0000 0014
```

Block No. 4. Runge-Kutta Method

One of the units of block No. 4 of the interchangeable built-in memory (addresses 3600-3677) is designed to contain routines for solving systems of ordinary differential equations by the Runge-Kutta method with constant integration interval.

The Runge-Kutta method, as applied to a system of n ordinary differential equations

$$y_i' = f_i(x, y_1, \ldots, y_n) \quad (i = 1, 2, \ldots, n) \tag{19}$$

consists in evaluating successive values of the functions y_{kl}, where k is the number of the point and i the number of the equation (function), by means of the formulas:

$$\left.\begin{aligned}
y_{k+1,i} &= y_{k,i} + \frac{hK_{1i}}{6} + \frac{hK_{2i}}{3} + \frac{hK_{3i}}{3} + \frac{kK_{4i}}{6};\\
K_{1i} &= f_i(x,\ y_{k1},\ \ldots,\ y_{kn});\\
K_{2i} &= f_i\left(x + \frac{h}{2},\ y_{k1} + \frac{h}{2}K_{11},\ \ldots,\ y_{kn} + \frac{h}{2}K_{1n}\right);\\
K_{3i} &= f_i\left(x + \frac{h}{2},\ y_{k1} + \frac{h}{2}K_{21},\ \ldots,\ y_{kn} + \frac{h}{2}K_{2n}\right);\\
K_{4i} &= f_i(x + h,\ y_{k1} + hK_{31},\ \ldots,\ y_{kn} + hK_{3n}),
\end{aligned}\right\} \qquad (20)$$

where h is the interval of integration.

At each step, as follows from Formulas (20), the values of the right sides are computed four times.

The initial information — the number n of equations and initial values of the functions y_i — are given as an α-sequence:

$$\begin{aligned}
{}'\alpha &= n;\\
{}'(\alpha+1) &= y_{01};\\
{}'(\alpha+2) &= y_{02};\\
&\ \ \cdot\ \cdot\ \cdot\ \cdot\ \cdot\\
{}'(\alpha+n) &= y_{0n},
\end{aligned}$$

whose governing address is stored in address II of an assembled word (NK)

'3005			α	

In addition, the following are specified in assembled words:

'3000 = 2h, twice the integration interval;

'3001 = x_0, the initial value of the independent variable;

'3002 = x_{max}, the final value of the independent variable;

'3004			m	

Here the number m indicates readout (printout) of the calculation results at each mth point of integration.

As working locations, we use addresses 0004-0043, as well as the 4n addresses representing the extent of the α-sequence, where the first n addresses are used to store the current values of the right sides of the equations, the next n addresses for the values of the arguments for them, and the last n addresses for sequential calculation of the quantities $2hK_{1j}$, i.e.,

$$\left.\begin{array}{l} '(\alpha+3n+1)=2h\cdot K_{j1}; \\ '(\alpha+3n+2)=2h\cdot K_{j2}; \\ \quad \cdots\cdots\cdots\cdots \\ '(\alpha+4n)=2h\cdot K_{jn}. \end{array}\right\} (j=1,2,3,4)$$

The initial address of the program is 3600. The current value of the independent variable for computation of the corresponding values of the right sides is given in address 0031. After calculation of the right sides f_1, linkage to the Runge-Kutta routine is accomplished at instruction 3634.

The initial address of the subroutine for calculating the right sides is given as the constant of the assembled word

3003	04			N

and '3677 equals the check sum for the given SSP block. On the basis of address 3200 (instruction 3615) control is transferred to the group-transfer and printout subroutine.

<u>Program for Runge-Kutta Method</u>

'3000 = $2h$
'3001 = x_0
'3002 = $x_{\max}$

'3003	04			N
'3004			m	
'3005			α	

```
RUNGE ... 3600 01 0000 3043 0040
             1 11 0040 3075 0036
             2 11 3042 3075 0037
             3 34 3005 0000 0043
             4 01 3005 4000 0033
             5 01 0033 4000 0016
             6 01 0016 4000 0035
             7 01 0036 0000 0017
          3610 01 0036 0000 0020
             1 01 3001 0000 0031
             2 01 3004 0000 0032
             3 01 0000 0000 0022
             4 01 0005 0000 0021

          3634 01 0035 3011 0027
             5 01 3005 3011 0024
             6 02 0036 0020 0020
             7 01 0020 0017 0017
          3640 04 0017 0037 3642
             1 02 0017 0037 0017
             2 34 0027 0000 0000
             3 11 4000 0017 0023
             4 34 0024 0000 0000
             5 01 4000 0023 4000
             6 01 0027 3011 0027
             7 01 0024 3011 0024
          3650 04 0024 0033 3642
             1 16 0022 3042 3673
             2 02 0040 0021 0021
             3 01 0021 0022 0022
             4 11 3000 0022 0041
             5 01 0041 0034 0031
             6 01 0035 3020 0042
             7 02 0026 3011 0026
          3660 02 0025 3011 0025
             1 02 0027 3011 0027
             2 34 0027 0000 0000
```

```
   5 04 3002 0031 3200
   6 01 0031 0000 0034
   7 01 3005 3011 0024
3620 01 0033 3011 0025
   1 01 0016 3011 0026
   2 34 0024 0000 0023
   3 34 0025 0000 0000
   4 01 0023 0000 4000
   5 34 0026 0000 0000
   6 01 0000 0023 4000
   7 01 0024 3011 0024
3630 01 0025 3011 0025
   1 01 0026 3011 0026
   2 04 0024 0033 3622
   3 05 0000 0000 3003
   3 11 0022 4000 0023
   4 34 0026 0000 0030
   5 34 0025 0000 0000
   6 01 0023 0030 4000
   7 04 0042 0027 3657
3670 01 0026 0043 0026
   1 01 0025 0043 0025
   2 04 0000 0000 3003
   3 02 0032 3011 0032
   4 01 3005 0000 0002
   5 30 0032 3612 3354
   6 05 0000 0000 3613
   7 Block check
     sum
```

STANDARD ROUTINES

Standard routines and subroutines are formed with allowance for a standard method of specifying the input and output information for them. As addresses for storing input and output information (arguments and results) we use addresses 0002, 0003, 0004. For routines processing complicated blocks (vector-sequences, etc.), the information is specified as α-sequences, for which addresses 0002 and 0004 serve as specifiers. The result is stored in address 0003. This is either the address of a variable if the result is a variable (for example, in a program for scalar multiplication of vectors), or a specifier, if the result is in turn a block (for example, in a routine for matrix multiplication).

Standard routines are formed in symbolic addresses; exceptions are formed by routines in interchangeable built-in storage, which are formed in absolute addresses. Calls for subroutines that do not in turn contain calls to subroutines (of lower level) are accomplished on the basis of a conditional-jump instruction transferring control to a subroutine of the form

K	30	3026	$K+1$	M

(with linkage based on the link register). Here M is the initial address of the subroutine; the first address specifies the address of the quantity whose sign governs the transfer of control to the subroutine (or the address of some negative constant, for example, 3026, if there is to be an unconditional jump to the subroutine). Here before the instruction calling for the subroutine the addresses of the arguments or their specifiers are transferred, respectively, into addresses 0002, 0003, 0004.

Let us list the routines developed as of 1 January 1962. In all cases, unless otherwise specified, linkage to subroutines is governed by the link register.

Subroutine for conversion from the binary system into the decimal system. In contrast to the corresponding subroutine stored in fixed built-in memory (see Table 2), this subroutine does not use group operations (which is necessary, for example, when the machine is operating in the simulated floating-point mode).

The program takes 9 instructions; argument '0002; converted number '0003; working addresses 0002-0005.

Subroutine for converting integers from the binary system into the decimal system. The binary integer, written into address II of location 0002 is converted into the decimal system and printed out. Group operatons are not used; linkage is governed by the link register; working addresses 0002, 0003; with the necessary constants, the program takes 23 instructions.

Subroutine for group conversion from binary system to decimal system. A group of numbers specified as an α-sequence with governing address 0002 is converted and printed out. The content of the α-sequence is stored. The subroutine uses group operations; working addresses 0003-0007; number of instructions, 18.

Subroutine for group conversion and printout using group-printout mode. As in the preceding subroutine, the converted numbers are transferred into an α-sequence which may also coincide with the initial α-sequence. The governing address of the second sequence is 0003. The working addresses are 0003-0011; group operations are used; the number of instructions is 26.

Subroutine for correcting program errors. Into some α-sequence, the correct codes are transferred, together with their addresses in the following sequence: first code, its address; second code, its address, etc.

In assembly location 3006, the governing address of the sequence is stored in address I; the last address is stored in the second (i.e., in NK 3006 there may be an instruction for loading an α-sequence — a zone of corrections.

Linkage is governed by the link register; number of instructions, 10; working addresses 0002-0005; F operation is used.

Subroutine for calculating sine for $-1 < x < 1$ using Taylor series. Accuracy, 9-10 digits; working addresses, 0004-0011; group operations are not used; '0002 is stored; number of instructions, 20.

Subroutine for computing tangent (cotangent). For $x < \pi/4$, the subroutine computes tan x, for $x \geq \pi/4$, it calculates the value cot x. The result is transferred on the basis of two addresses:

$$'0004 = \begin{cases} +0, & \text{if program yields tg } x; \\ -0, & \text{if program yields ctg } x. \end{cases}$$

$$'0003 = \begin{cases} \text{tg } x, & \text{if } '0004 = +0; \\ \text{ctg } x, & \text{if } '0004 = -0. \end{cases}$$

No group operations are used in the program; number of instructions, 21; '0002 is stored; working addresses, 0005-0007.

Subroutine for calculating arc tan x for $-1 < x < 1$. A polynomial of best approximation is used:

$$\operatorname{arctg} x = \sum_{i=0}^{7} C_{2i+1} x^{2i+1};$$

$$C_1 = 0{,}9999993329;\ C_7 = -0{,}1390853351;\ C_{13} = 0{,}0218612288;$$
$$C_3 = -0{,}3332985605;\ C_9 = 0{,}0964200441;\ C_{15} = -0{,}0040540580.$$
$$C_5 = 0{,}1994653599;\ C_{11} = -0{,}0559098861;$$

Number of instructions, 17; '0002 stored; group operations are used.

Subroutine for calculating ln x for 0 < x < 1. The program takes 34 instructions, and differs from the corresponding built-in program only in being more accurate.

Program for evaluating nth-order determinant. The determinant elements are given as an α-sequence with specifier 0002. The dimension of the determinant is stored in accordance with the governing address 'α = = n. The result is delivered in the form of the content of two addresses: for '0002 < 0, the value of the determinant equals '0003/'0004, while for '0004 > 0, it equals '0003·'0004.

The maximum determinant order is 25; number of instructions, 121; the F operation is used.

Subroutine for rectangular-matrix transposition. The information for the routine is specified as an α-sequence of its elements, by rows, and the size of the matrix is stored in accordance with the governing address of the sequence. Address 0002 serves as the specifier for this sequence. Address I of the sequence contains the number of lines, and address II the number of columns. Transposition consists in writing a column of the given matrix as a row of the transposed matrix in the form of an α sequence with specifier 0003. The elements are stored in the β-sequence in rows of the transposed matrix; the governing address for β contains the number of lines, while address β + 1 contains the number of columns of the transposed matrix.

The content of the α-sequence and the specifiers of the program are stored; the F operation is used; number of instructions, 18; working addresses, 0004-0011.

Linear combination of vectors. The information for the program is given as two sequences. The α-sequence gives the number of vectors, the dimension, the elements of the first, the elements of the first, the elements of the second, etc. (in the indicated sequence). The specifier for this sequence is address 0003. The β-sequence indicates the constants by which the vectors are to be multiplied. The specifier for this sequence is address 0003. The result is placed into an α-sequence for which the contents of addresses α and α + 1 are stored.

Number of instructions, 20; F operation is used; working addresses, 0002-0012.

Product of matrices and vector by matrix. The initial instruction of the program for multiplying a vector by a matrix is instruction K + + 13, the initial instruction for multiplication of matrices is instruction K + 0.

The sequence of elements of the vector or the first matrix are given as an α-sequence with specifier 0002. In the first case, the governing address of the sequence contains its dimension; in the second case it contains the number of lines, while the next address contains the number of columns, following which we have the matrix elements given by rows.

The second matrix is given in rows as a β-sequence with specifier 0004. The governing address indicates the number of rows and the next address the number of columns of the matrix.

The result is obtained as a γ-sequence with specifier 0003; the governing address indicates the number of lines of the resulting matrix, the next address the number of columns, followed by the elements of the matrix i rows.

A call to the routine is based on address 0001 (i.e., the link instruction is formed in location 0001). The input information to the program is stored. However '0002, '0003, and '0004 are not stored. Number of instructions, 35; working addresses, 0002-0014; the F operation is used.

Product of matrices and vector by matrix with scaling (variation I).

The initial instruction of the program for multiplying a vector by a matrix is K + 0, for multiplication of a matrix by a matrix, K + 2. The input information is given, as for the preceding program, while the output information is given as a γ-sequence with specifier 0003. Address I of the sequence contains the number of rows of the resultant matrix, while address II indicates the number of columns. There next follow the elements of the matrix and its scale factor:

'$\gamma = m$, number of matrix elements;

'$(\gamma + 1) = n$, number of matrix columns;

'$(\gamma + 2) = a_{11}$, first element of matrix;

'$(\gamma + 3) = \mu_{11}$, scale factor on first matrix element, etc.

The output information for the program is stored; '0002, '0003, '0004 are not stored; the F operation is used; number of instructions, 43; working addresses, 0002-0015.

Product of matrixes and vector by matrix with scaling (variation II). The initial instruction for the routine for multiplying a vector by a matrix is K + 0, for multiplication of a matrix by a matrix, K + 2. In contrast to the preceding program, a general scale factor is introduced for the resultant matrix applying to all of its elements; it is given in accordance with address 0016. The F operation is used; number of instructions, 48; working addresses, 0002-0017.

Scalar product of vectors. The vectors are given in the form of α-sequences with governing addresses with 0002 and 0004. The results are given in address 0003. The F operation is used; number of instructions, 12; working addresses, 0005-0007.

Multiplication of square matrix, transposed to given matrix, by vector. The information on the matrix is given by rows in the form of an α-sequence with governing address 0002; information on the vector is given on the basis of the specifier 0004. The governing address of the vector result is 0003. The F operation is used; number of instructions, 21; working addresses, 0005-0015.

Multiplication of square matrix by vector. Address 0002 is the specifier of the matrix, given by rows; '0002 equals the dimensions of the vector; 0004 is the specifier of the given vector; 0003 is the specifier of the vector result (blocks for the given vector and vector result will be different). The F operation is used; number of instructions, 20; working addresses, 0005-0015.

Evaluation of Bessel functions $I_n(x)$ for fractional argument and integral index

$$I_n(x) = \frac{x^n}{2^n n!}\left\{1 - \frac{x^2}{2(2n+2)} + \frac{x^4}{2 \cdot 4(2n+2)(2n+4)} - \dots\right\}.$$

here '0002 is the function argument; '0004 is the index of the Bessel function (given in address II); '0003 is the result. Working addresses, 0004-0013; group operations are not used; accuracy, 8 decimal digits; number of instructions, 41.

Evaluation of definite integrals by Simpson's rule for even number n of divisions (fixed-point mode). The input information takes the form of the following α-sequence with specifier 0002: 'α = n, number of intervals of division (even number); '(α + 1) = h, integration interval; '(α + 2) = y_0, ..., '(α + n + 1) = y_n, values of integrand function at points of division of interval of integration. The formation of the content of the α-sequence is accomplished by the main program. The result is the content of address 0003. Number of instructions, 39; working addresses, 0004-0014.

<u>Evaluation of definite integrals by Simpson's rule combined with trapezoidal rule for odd number of divisions</u> (fixed-point mode):

$$y = y_1 + y_2;$$
$$y_1 = \frac{h}{3}(y_0 + 4y_1 + 2y_2 + \ldots + 4y_{n-2} + y_{(n-1)});$$
$$y_2 = \frac{h}{2}(y_{n-1} + y_n).$$

The information is specified as in the preceding program. Number of instructions, 48; working addresses, 0004-0014; result, '0003.

<u>Transposition of square matrix</u>. Here 0002 is the specifier for the initial matrix, given by rows; 0003 is the specifier of the transposed matrix (also obtained by rows). Number of instructions, 17; working addresses, 0004-0012; '0002, '0003 are stored.

<u>Product of vector by rectangular matrix</u>. The vector elements are given in an α-sequence ('α = n) with specifier 0002. The matrix is given by rows in a β-sequence ('β, number of rows; '(β + 1), number of columns) with specifier 0004. Here 0003 is the specifier for the γ-sequence, which contains the elements of the resultant vector ('γ equals the dimension of the vector). Number of instructions, 22; F operation is used; contents of '0002, '0003, '0004 are stored, as is all of the input information; working addresses, 0005-0013.

<u>Matrix inversion by method of A.P. Yershov</u> (fixed-point mode). The initial information — the general scale factor and elements of the matrix by rows — is given in the form of a sequence with specifier 0002. Into assembled word 3000, the order of the matrix is introduced into address II. Several matrices of the same order may be inverted simultaneously. The number of matrices is specified in address III of assembled word 3001.

The inverted matrices are given (in the decimal system) by columns (with spaces between columns). The scale factor is printed out first.

Number of instructions, 48; the F operation is used.

Determination of the real roots of a polynomial of degree n. The region of definition of the roots is determined by the program in accordance with the formula

$$|x| < \frac{A}{|a_0|} + 1,$$

where $A = \max|a_i|$ $(i = 0, 1, \ldots, n)$; a_0 is the coefficient on the highest-order term.

The method of samples is used. A root-extraction indicator can be:

a) a change in sign of the function;

b) a change in sign of its derivative near the ox axis.

Calculations are carried out in the floating-point mode. Number of instructions, 319. The coefficients of the polynomial are first divided by K, where A < K < 10A, and they are given in a sequence of decreasing powers (those equal to zero are coded by zeros). The following are given in assembled words: '3000 = n, degree of polynomial (in address I); '3001 = m, number of polynomials (in address I).

The result is printed out in floating-point mode (exponent, mantissa). Accuracy, 4-5 digits.

Calculation of real characteristic roots and their corresponding vectors by iteration method. The characteristic root of greatest absolute value is found:

$$Ay_0 = y_1;\quad Ay_{i-1} = y_i;$$
$$A'y'_0 = y'_1;\quad A'y'_{i-1} = y'_i;$$
$$\lambda_i = \frac{(y'_i, y_i)}{(y'_{i-1}, y_i)},$$

where A is the initial matrix; A' is the transposed matrix; y_0, y'_0 are arbitrary zero approximations to the eigen vectors.

The characteristic root of matrix A is λ_1, for which

$$|\lambda_i - \lambda_{i-1}| < \varepsilon,$$

where ε is small.

The vectors y_1, y_1' corresponding to the given λ_1 will be the eigenvectors of matrices A and A', respectively.

Matrix $A_1 = A - \lambda[y, y']$ has the same characteristic root as matrix A, except for the first characteristic root λ, already computed.

The corresponding characteristic root of matrix A_1 will equal zero. The characteristic root of matrix A_1 having greatest absolute value, and equaling the second characteristic root of matrix A, is also computed by the iteration method. The same is true of the corresponding eingenvectors. Here [y, y'] represent the matrix product

$$\begin{pmatrix} a_1b_1 & a_1b_2 & a_1b_3 & \dots & a_1b_n \\ a_2b_1 & a_2b_2 & \dots & \dots & a_2b_n \\ \cdot & & & & \cdot \\ \cdot & & & & \cdot \\ \cdot & & & & \cdot \\ a_nb_1 & \dots & \dots & \dots & a_nb_n \end{pmatrix}$$

The characteristic roots of several matrices of the same order may be computed simultaneously with no halt. The results are printed out in the following order: $M\lambda_1$, the scale factor; λ_1, the first characteristic root; space, etc.

The program may also be used to compute multiple roots if the zero approximation can be given. It is possible to begin with one of the four zero approximations:

$$\begin{matrix} 1, 1, 1, \dots, 1 \\ 1, 0, 0, \dots, 0 \\ 0, 1, 0, \dots, 0 \\ 1, 0, 0, \dots, 0, 1 \end{matrix}$$

The choice of initial vector is determined by address I of NK 3001.

If none of these zero approximations leads to a result, the zero vectors must be given as an α-sequence, introduced after the initial matrix. Here it is necessary to omit construction of the zero approximations. The program uses subroutines for scalar multiplication, multiplication of a matrix by a vector, and multiplication of a transposed matrix by a vector.

The order of the matrix is given in address II of NK 3000 (in the octal system); '3001 = ε (accuracy of approximation); '3002 = m − 1, where m − 1, where m is the number of variations.

Number of instructions, 268; maximum order of matrix, 23. The program was written by V.S. Morornaya.

Finding minimum characteristic root of matrix by method of steepest descent. The calculation formulas are:

$$Ax = \lambda Bx;$$
$$\lambda_m = \frac{(x_m, Ax_m)}{(x_m, Bx_m)};$$
$$x_{m+1} = x_m - a_m r_m;$$
$$r_m = Ax_m - \lambda_m Bx_m;$$
$$a_m = \frac{(r_m, r_m)}{(r_m, A_m r_m)}.$$

Information on the matrices is given in the form of an α-sequence which contains: the order of the matrix A; its elements in rows; the order of matrix B; its elements in rows. As the zero approximation, we use a vector with unit components. The order of the matrix and number of variations is given in assembled words.

The program yields a characteristic root with scale factor and the components of the corresponding eigenvector. If the programmer so wishes he can arrange to have all intermediate iterations printed out (by setting a switch at the console).

The program can be used for the case of a real minimum root. The maximum order of the matrices is 24 (unless external storage is used). The program was written by V.S. Motornaya.

Solution of a system of linear algebraic equations by the square-root method. The information on the problem − order of the matrix, the matrix in rows, the right sides − is given in the form of an α-sequence with specifier 0002. The result is obtained in the form of an α-sequence with specifier 0003. The program runs with floating scale factor. Number of instructions, 282.

Solution of system of linear algebraic equations by Seidel method. The information is given as in the preceding program. The program is realized for the interchangeable built-in memory (64 instructions).

Solution of systems of linear algebraic equations for matrices with incomplete filling by Lopschits method. The initial information on the matrix of coefficients is given in the form of two sequences: an α-sequence of scale factors and a β-sequence of coefficients. The scale locations contain: in address I, the number of the line of the nonzero matrix element, in address III, the number of its column, and in address II the address of the element itself. In the β-sequence of coefficients, there are coded only different and nonzero matrix elements.

The governing elements of the α- and β-sequences are given in address II of the assembled words 3000 and 3001, respectively.

Number of instructions, 83; working locations, 0001-0012; F operation is used. The program was written by Ye.I. Mikhaylova.

Mean-level wind prediction. The initial information is the matrix of initial data for the baric topography. The output information is the wind-speed vector matrix. To solve the corresponding system of partial differential equation, a special method of numerical solution has been developed. Number of instructions, 587. Input matrix, 23 × 19. Group operations and standard routines are used together with constants of the built-in memory. The computational method and the program were developed by Yu.S. Fishman.

Solution of Poisson equation for Dirichlet problem for a rectangular region by the method of block iteration using the driving method. Assembled words are used for the information: '3000 = n, number of points on x axis; '3001 = m, number of points on y axis; '3001 = m, number of points on y axis; '3003 = h^2, where h is the grid spacing on the x axis; '3004 = ℓ^2, where ℓ is the grid spacing on the y axis; '3005 = ε is the prescribed accuracy. When only high-speed storage is used, $mn \leq 360$; the program may be used to solve the Laplace equation for a grid $mn \leq 720$. Number of instructions, 81. The program was written by G.I. Visnyuk.

Solution of Laplace equation for rectangular region by method of block iteration using driving method (with memory-location reduction using a method proposed by I.N. Molchanov; see DAN UkSSR, No. 3, 1962). The following information is given by means of assembled words: '3000 = = h^2, where h is the grid spacing on the x axis; '3001 = ℓ^2, where ℓ is the grid spacing on the y axis; '3002 = ε, the prescribed accuracy; '3003 = m, number of mesh points on x axis; '3004 = n, number of mesh points on x axis. Memory space is conserved owing to the fact that the values of the function are given only for even lines of the grid. High-speed memory permits solution of equations for a grid $mn \leq 35^2$.

Number of instructions, 305. The program was written by G.I. Viznyuk.

Algorithm for obtaining pseudorandom numbers with uniform distribution law. The method of residues is used. Number of instructions, 132.

Algorithm for laying out rectangular parts on rectangular sheets. The algorithm developed yields (as compared with manual part lay-out) a savings of 2% in metal. The over-all number of parts may reach 100; up to 12 parts may be laid out on one sheet. Number of instructions, 300. The algorithm and program were developed by G.Ya. Mashbits.

Calculation of correlation functions for large numerical blocks (using magnetic drum). Number of instructions, 150. The program was written by E.K. Yadrenko.

Realization of random process with correlation function $e^{-a|\tau|} \cos b\tau$. Number of instructions, 250. The program was written by E.K. Yadrenko.

Solution of transportation problem by Lur'ye-Oleynik method. The high-speed memory and one magnetic drum of external storage are used completely. The cost matrix may have dimension m × n up to 3000. The program was written by S.B. Branovitskaya.

Design of highway and railway profiles with fixed abscissas of points of discontinuity – stakes (with allowance for enginerring requirements). Information on up to 20 km of route is loaded at one time into computer storage. The method of successive analysis of variations, developed by V.S. Mikhalevich and N.Z. Shor is used. The program was written by A.N. Sibirko.

Program for reading out alphanumeric information on screen of cathode-ray tube. The program was written by V.K. Yeliseyev.

Translator program for general-purpose control computer (UMShN) using the address language as source language; the program is based on the principle of element-by-element decoding of the input information. The PP compiles the working programs only for fixed-point operation. The PP has a length of about 290 address lines or 600 "Kiev" computer instructions. The PP processes the following address formulas: transfers, predicate formulas, entrance formulas, unconditional-jump labels, nonstandard formulas written in UMShN codes, printout formulas, and halt formulas.

After the translator program processes the initial information in each zone, the problem program is printed out in UMShN computer language; the instructions of the working program are printed out with absolute numbers. The program was written by R.A. Godz', G.A. Polishchuk, A.I. Stiranka, under the guidance of Ye.L. Yushchenko.

Hydraulic design of water-and gas-supply systems. The computer solves a nonlinear system of hydraulic computations for water- and gas-supply networks. It is possible to design networks consisting of k sections; at each end of the system there are a maximum of m sections. If the data is to be stored in high-speed memory, it is necessary that $2k + 3m \leq 808$. Number of instructions, 261. Time required for calculations for a system of 250 sections, 30 min. The program was written by V.S. Kvakush in accordance with a method suggested by B.N. Pshenichnyy.

Realization of algorithm for determining pouring time for a Bessemer converter for a given metal carbon content. Current information on the course of the converter Blow is furnished by sensors installed at the Bessemer mill of the Dzerzhinskiy Plant (in Dneprodzerzhinsk); the information is converted to digital form by a special device, averaged, and transmitted over a telegraph channel to the receiving apparatus, installed in the computer room for the "Kiev" computer. A special decoder converts the telegraph signals into binary code and introduces them into a storage unit of the "Kiev" computer.

The program determines the instant of pouring, and a control signal is transmitted by telegraph channel to the mill.

The work was carried out by A.I. Nikitin and others led by V.M. Glushkov, K.S. Garger, and L.N. Dashevskiy.

Determination of optimum process regime in a carbonization column at a soda combine. The solution is similar to that for the preceding problem for a controlled object — a soda combine (at the city of Slavyansk). The work was guided by B.N. Malinovskiy and A.B. Tyutinnikov; I.A. Yanovich and R.S. Tsvigun were project executives.

Traction calculations for a locomotive. Information is specified for a rectified path profile in the form of lengths of elements and reduced grades. The program gives the running time and speed of the locomotive over elements and over runs, the electric-power consumption, engine overheating, mechanical work, and other intermediate data. Calculation time is 5 min per 100 km of route. Number of instructions, 400.

The method and algorithm were developed by N.Z. Shor and A.A. Alekseyev.

Solution of systems of linear algebraic equations by the method of conjugate gradients with minimization of the norm for departure of the approximate solution from the exact solution (see V.Ye. Shamanskiy, "Ukrainian Mathematical Journal," No. 1, 1962). For matrices with incomplete filling, scales are used, and only the nonzero elements are coded.

If $\underline{n}$ is the order of the system and $\underline{m}$ the number of nonzero elements, then for storage of the information in high-speed memory, the condition $6n + m \leq 673$ must be satisfied. For matrices with complete filling, $n \leq 23$. Number of instructions, 319. The program was written by G.P. Pravotorova and M.F. Yakovlev.

Solution of systems of linear algebraic equations by the method of conjugate gradients with minimization of the discrepancy norm. As in the preceding program, scales are used for matrices with incomplete filling. For storage of the information in high-speed memory for a matrix with incomplete filling, the condition $6n + m \leq 608$ must be satisfied, where $n \leq 41$ and $\underline{m}$ is the number of nonzero elements; for matrices with complete filling, $n \leq 21$. Number of instructions, 370. The program was written by G.P. Pravotorova and M.F. Yakovlev.

Solution of systems of linear algebraic equations with symmetric matrix using the improved Gauss method (see the sample program compiled by PP-AK).

Solution of symstems of differential equations by the Runge-Kutte (see the sample block of the interchangeable built-in memory).

Solution of a parabolic partial differential equation with boundary conditions of the first kind. The heat equation and Boussinesq equation are solved for regions of arbitrary configuration (the form of the equation is specified by a special characteristic). Information on the type of region is specified with the aid of special logical scales. The grid method is used with an explicit difference scheme.

The maximum number of grid points is 750. Number of instructions, 120. The program was written by V.F. Temchenko.

Approximation of functions of two variables by the method of least squares (see A.S. Novik and V.Ye. Shamanskiy, DAN UkSSR, 1962, No. 3). The input information is a table of values for a function of two variables f(x, y) at the mesh points of a rectangle. The program gives the coefficients of an approximating polynomial of the form

$$P_{nm}(x, y) = \sum_{r=0}^{n} x^r \sum_{s=0}^{m} c_{sr} y^s.$$

Number of instructions, 408. High-speed memory permits construction of polynomials for which the following relationship holds:

$$\frac{\alpha\beta}{2} + \alpha + \beta + 3\max(\alpha, \beta) + 2\max(m, n) + \beta(m + 1) + \\ + \frac{m^2 + 3m}{2} < 609,$$

where α is the number of mesh points along $\underline{x}$; β is the number of mesh points along $\underline{y}$; $\underline{m}$ and $\underline{n}$ are the degrees of the polynomials in $\underline{x}$ and in $\underline{y}$, respectively.

The program was written by A.S. Novik.

Algorithm for obtaining pseudorandom numbers with uniform distribution, based on the shift (13) and equality (17) operations. The numbers have a period in excess of 500,000. Number of instructions (with check on randomness and uniformity conditions), 105. The program was written by V.S. Kvakush.

Diameter and pressure calculations for gas- and water-supply networks. Let m be the number of elements, n the number of sections, and k the maximum number of sections in one element. High-speed memory permits calculations to be carried out when the condition $m + 2n + 4k \leq \leq 706$ is satisfied. Number of instructions, 288. The program was written by V.S. Kvakush in accordance with a method proposed by B.N. Pshenichnyy.

We shall now give an incomplete list of algorithms realized on the "Kiev" computer for simulating elementary processes of thought, creation, etc.:

Algorithm for morphological analysis of the Russian language proposed by I.A. Mel'chuk. Number of instructions, 360. (See N.M. Grishchenko, "Problems of Cybernetics," No. 6, 1961).

Algorithm for syntactic analysis of the Russian language proposed by I.A. Mel'chuk. All of the high-speed memory and one magnetic drum are used. The program was written by S N. Yakimenko.

Algorithm for learning to recognize the meaning of simple sentences (see V.M. Glushkov, N.M. Grischenko, and A.A. Stogniy, Principles of Construction of Learning Systems, Gostekhizdat UkSSR, 1962). An idea of V.M. Glushkov was used as the basis for construction of a self-perfecting system of algorithms which after communication of a certain number of N of randomly chosen phrases of given structure will correctly recognize the meaning of any phrase of the same structure. The algorithm and program were developed by A.A. Strogniy and N.M Gritsenko.

Simulation of biological evolutionary processes. (See A.A. Letichevskiy, Principles of Construction of Learning Systems, Gostekhizdat UkSSR, 1962). The program algorithms were developed by A.A. Letichevskiy and A.A. Dorodnitsyna under the guidance of V.M. Glushkov.

Algorithm for learning to recognize elementary geometric figures (see V.M. Glushkov, V.A. Kovalevskiy, and V.I. Rybak, Principles of Construction of Self-Perfecting Systems, Gostekhizdat UkSSR, 1962).

Correlation method of pattern recognition. The authors of the program were V.A. Kovalevskiy, V.K. Yeliseyev.

Glushkov algorithm for synthesis of automatic device on the basis of event presented to it (See V.M. Glushkov and A.A. Storgniy, "Computer Mathematics and Mathematical Physics," 1961, No. 3).

Minimization of automatic devices using Aufenkamp-Khon method (See A.A. Stogniy, "Computer Mathematics and Mathematical Physics," 1961, No. 3). The program was written by V.P. Klimenko.

PROGRAM REALIZATION OF FLOATING-POINT MODE ON "KIEV" COMPUTER

General Characterization of Simulating Routine (MP)*

The determination and matching of scale factors in the preparation of problems for fixed-point computers frequently involves difficulties, since in the realization of various programs sequences of operations are performed for which the maximum values of input and output quantities cannot be subjected to any restrictions. In many cases, each new problem gives rise to its own method of scale-factor calculation and as a result scaling becomes an art, unamenable on the whole to standardization. Thus in order to increase the effective speed of a fixed-point general-purpose computer we should include in its operation set special operations simplifying the program implementation of the floating-point mode. Of these operations, together with logical operations and a comprehensive set of control-transfer operations, an important role is played in the "Kiev" machine by the normalization operation.

In the development of the algorithms for program realization of the floating-point mode in the "Kiev" computer, the aim was to create conditions such that it would be possible to write a program in fixed-point mode while its actual realization could take place in the floating-point mode. Such algorithms came to be called simulation algorithms, since they simulate by program means the operation of the missing electronic circuits which the computer would need to realize floating-point operations. The admissibility of constructing simulating algorithms follows from the existence of an accurate description of the rules for transforming codes into floating-point mode (for the selected coding method) on the one hand, and from the universality of the computer itself, on the other.

The fundamental tendency of the present version of the MP consists in minimizing storage space, in isolated cases at the expense of computational time.

The initial data for the calculation is given in the form of a sequence of numbers with a single scale factor, which is stored at the end of the sequence. The numbers occupy one storage location each. The seven lowest-order digit positions are used for exponents. When the results are read out, one number is printed out on two lines (mantissa and exponent).

Programs for fixed-point calculations are written in "Kiev" machine language, which is subject to several restrictions. This means that the programmer, writing the program with the indicated restrictions for fixed-point calculations may in the event of an unhappy choice of scale factors go over to calculations with the aid of the MP, which will solve the same problem in the floating-point mode.

Let us call computer fixed-point operation mode I and, correspondingly, floating-point operation mode II. Then if in writing a program we use the set of "Kiev" machine operations given below, to solve a problem in mode II it is sufficient to set up in address II of NK 3000 the number of the first instruction of the working program and transfer control to location 3300 (to the beginning of the MP).

A practical check on the simulation system, using several problems including specially selected problems, with allowance for statistical data on the frequency with which various operations are employed in the solution of problems of computational nature, carried out in both modes, showed that the rate of program execution decreased by roughly a factor of 10 in mode II.

The program was implemented in the form of an interchangeable built-in memory (256 codes). The high-speed memory remained completely free except for the first 24 locations (0001-0032).

OPERATION SET FOR MP

The developers of the MP kept in mind that in the floating-point mode it would be necessary to solve problems of peculiarly computational nature. Thus fundamental attention was devoted to achieving minimum use of the high-speed storage by the simulating program together with a high rate of execution of its arithmetic operations. In this connection, the logical possibilities of the computer were somewhat reduced, which is reflected in the fact that several operations may not be used. Still we should point out that the logical operations admitted for mode II are executed fairly rapidly and the rate of program implementation for the solution of problems of computational nature rises sharply with an increase in the number of logical operations used.

For mode II, the operation set of the "Kiev" computer includes the following operation: addition (01), subtraction (02), instruction addition (03), subtraction of absolute values (06), cyclic addition (07), multiplication with rounding (11), division (12), logical shift (13), logical addition (14), logical multiplication (15), transfer of control on equality (16), addition modulo 2 (17), load numbers (20), load instructions (21), readout (printout) (22), call to external memory (23, 24, 25), transfer control on sign of number (31), halt (33), read on basis of specifier (34), normalization (35).

The simulating program does not accept indicators of address modification. Thus, program segments making use of modification must not be included in the simulation mode. This does not, however, mean that use of the A-register is completely excluded in mode II. The A-register may be used in connection with the F operation (34) to reduce the rank of an address and to form control by iterative-type loops. In view of this, in programming schemes for block scanning, it is necessary to make use of the F operation.

Structure of Simulation Program. The simulation program (MP) is similar in structure to the algorithm for element-by-element decoding of address functions given in Chapter 3. Using an algorithm called the operation commutator, the simulating program extracts the code for the next instruction to be simulated, and uses this as the basis for transferring control to the corresponding subroutine. Here nonsimulated operations are executed directly (in fixed-point mode).

Instruction Commutator (instructions 3300-3316, 3557-3573)

```
3300 01 0000 3621 0026          1 16 0013 3044 3333
   1 01 0000 3622 0027          2 16 0013 3050 3461
   2 02 3000 3011 0014          3 16 0013 3601 3333
   3 14 3050 3026 0012          4 16 0013 3602 3323
   4 01 0014 3011 0014          5 16 0013 3603 3317
   5 34 0014 0000 0011          6 31 0000 3557 0000
   6 15 0011 3025 0013          7 01 0011 0000 0010
   7 16 0013 3600 3430       3320 01 0014 3011 0014
3310 16 0013 3045 3333          1 34 0014 0000 0011

   2 31 0000 0010 0000          1 30 3026 3562 0011
   3 13 3604 0011 0014          2 16 3026 0013 3630
   4 34 0014 0000 0014          3 31 0000 3304 0000
   5 31 0014 3326 3330          4 13 3624 0011 0017
   6 15 0011 3023 0014          5 15 0011 3032 0021
   7 31 0000 3305 0000          6 14 0021 3574 0021
3330 13 3624 0011 0011          7 01 0021 0000 0011
   1 15 0011 3023 0014       3570 31 0000 3561 0000
   2 31 0000 3305 0000          1 15 0017 3023 0014
3557 16 0013 0426 3510          2 02 0014 3011 0014
3560 16 0013 0427 3564          3 32 0000 0000 0000
```

Simulation of Addition-Type Operations

The first group of operations consists of addition (01), subtraction (02), and subtraction of absolute values (06), which are simulated by a single subroutine. To execute these operations, the machine transfers the operation arguments into locations 0020 (first component) and 0015 (second component). The subroutine separates the mantissas from the exponents: 0021 — mantissa of term I; 0022 — exponent of I; 0023 — mantissa of term II; 0024 — exponent of term II; it then performs the operation. Here the number exponents are equalized, and the operation is performed on the mantissas. The result of the operations is normalized, the mantissa is combined with the exponent in a single location, and transferred in accordance with address III of the simulated instruction. If the exponents are equal or their difference exceeds the number of additional shifts, i.e., the operation result, to within the sign, equals one of the components, exponent equalization is skipped in order to save time, and the result is transferred directly according to address III of the simulated instruction. Running of the program is affected by the setting of the overflow switch. This makes it possible to make the computational process automatic in case of possible overflows of the digit format.

Subroutine for Addition-Type Operations [addition (01), subtraction (02), modulo subtraction (06)]

```
3333 14 0013 3613 0025        5 31 0016 3346 3410
   4 13 3624 0011 0017        6 05 0016 3607 3352
   5 13 3604 0011 0016        7 34 0017 0000 0000
   6 34 0011 0000 0015     3350 14 0000 0020 4000
   7 34 0016 0000 0020        1 31 0000 3304 0000
3340 15 0020 3605 0021        2 02 0000 0016 0016
   1 15 0020 3606 0022        3 31 0023 3354 3356
   2 15 0015 3605 0023        4 13 0016 0023 0023
   3 15 0015 3606 0024        5 31 0000 0025 0000
   4 02 0022 0024 0016        6 15 0023 3035 0023
   7 13 0016 0023 0023
3360 02 0000 0023 0023
   1 31 0000 0025 0000
   2 05 0022 3610 3365
   3 33 0000 0000 0000
   4 31 0000 3363 0000
3365 01 3012 0022 0022
   6 11 0021 3042 0021
   7 01 3042 0021 0021
3370 15 0021 3611 0016
   1 16 0000 0016 3375
   2 01 0021 3612 0021
   3 31 0000 3375 0000
   4 31 0000 3362 0000
   5 15 0021 3605 0021
   6 16 0021 0000 3446
   7 34 0017 0000 0000
3400 14 0021 0022 4000
   1 31 0000 3304 0000
   2 35 0021 0016 0021
```

```
   3 01 0016 0022 0022
   4 04 3012 0022 3370
   5 34 0017 0000 0000
   6 15 3026 0021 4000
   7 31 0000 3304 0000
3410 16 0000 0016 0025
   1 05 0016 3607 3415
   2 14 0000 0000 0021
   3 14 0024 0000 0022
   4 31 0000 0025 0000
   5 31 0021 3416 3421
   6 13 0016 0021 0021
   7 14 0024 0000 0022
3420 31 0000 0025 0000
   1 15 0021 3035 0021
   2 13 0016 0021 0021
   3 02 0000 0021 0021
   4 14 0024 0000 0022
   5 31 0000 0025 0000
```

Simulation of Multiplication-Type Operations

The second group — the multiplication operation (11) and division operation (12) — are also simulated by a single subroutine.

In multiplication, the exponents are added, and the result of multiplication of the mantissas is normalized. In division, the machine checks to see that the denominator is not equal to zero and that division can be performed. Where the first condition is not satisfied, an automatic halt occurs. Here address 0023 will contain a zero. If the mantissa of the divisor exceeds the mantissa of the dividend, the exponents are subtracted and the mantissas divided (the operation is possible); in the opposite case, the mantissa of the numerater is first shifted and the exponent increased accordingly. As in the case of an addition-type operation, when this routine is run, the mantissas are separated from the exponents; the normalized result is stored according to address III of the simulated instruction.

Running of the subroutine is affected by the setting of the overflow switch.

Program for Multiplication-Type Operations (instructions 3430-3460, 3461-3505)

```
3426 22 0000 0000 0000
   7 16 0000 0000 0000
3430 34 0011 0000 0015
   1 13 3624 0011 0017
   2 13 3604 0011 0016
   3 34 0016 0000 0020
   4 15 0020 3605 0021
   5 15 0020 3606 0022
   6 15 0015 3605 0023
   7 15 0015 3606 0024
3440 01 0022 0024 0022
   1 02 0022 3611 0022
   2 05 0022 3610 3445
   3 33 0000 0000 0000
   4 31 0000 3443 0000
   5 04 3012 0022 3451
   6 34 0017 0000 0000
   7 14 0000 0000 4000
3450 31 0000 3304 0000
   1 11 0021 0023 0021
   2 05 0021 0000 3456
   3 35 0021 0016 0021
   4 01 0016 0022 0022
   5 04 0000 0022 3376
   6 34 0017 0000 0000
   7 14 0000 0000 4000
3460 31 0000 3304 0000
   1 34 0011 0000 0015
   2 16 0015 0000 3363
   3 13 3624 0011 0017
   4 13 3604 0011 0016
   5 34 0016 0000 0020
   6 16 0020 0000 3477
   7 15 0020 3607 0021
3470 15 0020 3607 0022
   1 15 0015 3605 0023
   2 15 0015 3606 0024
   3 02 0022 0024 0022
   4 10 0022 3611 0022
   5 05 3610 0022 3363
   6 04 0000 0022 3502
   7 34 0017 0000 0000
3500 14 0000 0000 4000
   1 31 0000 3304 0000
   2 11 0021 3042 0021
   3 01 0022 3012 0022
   4 12 0021 0023 0021
   5 31 0000 3370 0000
```

Number Loading and Readout in Mode II

For floating-point-mode operation, numbers are written in the following form: the lowest-order digit positions from the first through the seventh are set aside for the exponent, arbitrarily represented as $a = p + 64$, where p is the actual exponent of the number. Since $-64 \leq$ ≤ 63, the conventional exponents form a range ([0-127], while the numbers range correspondingly from 2^{-64} to 2^{+63}. An empty location is taken as zero. The 8-th through 40-th digit positions are occupied by the mantissa. The 41st digit position is the mantissa sign.

For loading into the computer, numbers are represented in nonnormalized form in the range $-1 < x < 1$, i.e., in the form normally used for the "Kiev" computer. Numbers not falling within the range (-1, 1) are introduced in the following manner. All n numbers are divided by a suitable number M and given in the form of an α-sequence whose last position contains the number M^{-1}. All-in-all, the sequence will contain (n + 1) numbers. If M = 1, a zero is written in place of M^{-1}. The number

M may be chosen arbitrarily. It may be a power of 2, 10, etc.

The subroutine for loading numbers at the instruction "load numbers" (20) introduces the α-sequence of numbers in fixed-point form with the chosen scale factor M, converts them into floating-point form with allowance for the scale factor, and stores the converted numbers in the same addresses. The address that had contained the factor M may be used for other purposes.

<u>Subroutine for Loading Numbers</u> (Instructions 3630-3705)

```
3630 35 0000 0006 0000
   1 14 0000 0000 0017
   2 15 0011 3023 0005
   3 15 0011 3023 0005
   4 11 0004 3040 0003
   5 02 0005 0003 0007
   6 11 0003 3040 0005
   7 14 0004 0005 0005
3640 14 3630 0005 0004
   1 14 3050 3026 0005
   2 30 3026 3643 0004
   3 34 0003 0000 0000
   4 16 0000 4000 3650
   5 15 4000 3605 4000
   6 01 0006 3611 0006
   7 14 0006 4000 4000
3650 05 0007 0017 3655
   1 01 3011 0017 0017
   2 03 3016 0004 0004
   3 01 3011 0003 0003
   4 31 0000 3642 0000
   5 14 3011 0000 0017
   6 34 0011 0000 0000
   7 16 4000 0000 3304
3660 15 3605 4000 0004
   1 15 3606 4000 0005
   2 11 0011 3040 0006
   3 34 0006 0000 0000
   4 15 3605 4000 0022
   5 15 3606 4000 0010
   6 16 4000 0000 3702
   7 12 0022 0000 4000
3670 31 0000 3674 0000
   1 11 0022 3042 0022
   2 01 0010 3012 0010
   3 30 0000 3667 0000
   4 02 0010 0005 0010
   5 01 0010 3611 0010
   6 35 4000 0003 4000
   7 01 0003 0010 0010
3700 15 4000 3605 4000
   1 14 4000 0010 4000
   2 16 0017 0007 3304
   3 01 3011 0017 0017
   4 03 3011 0006 0006
   5 31 0000 3663 0000
```

The printout subroutine converts the mantissas and exponents into the decimal system and prints out the results in two-line form: the mantissa and exponent of a number appear on different lines. The argument of the subroutine is stored in address 0004.

<u>Printout Subroutine</u>

```
3510 14 0000 0000 0007
   1 15 0004 3606 0005
   2 15 0004 3605 0006
   3 16 0005 0000 3551
   4 05 3625 0005 3524
   5 05 3623 0005 3532
   6 11 0006 3626 0006
   7 02 0007 3012 0007
3520 35 0006 0004 0006
   1 31 0000 3514 0000
   2 05 3611 0005 3536
   3 11 3042 0006 0006
   4 01 3012 0005 0005
   5 31 0000 3532 0000
   6 01 0000 0006 0002
   7 30 3026 3540 3100
3540 05 0007 3616 3547
   1 31 0007 3542 3544
   1 01 0004 0005 0006         2 02 0007 3617 0004
   2 01 0005 3615 0005         3 31 0000 3545 0000
   3 31 0000 3515 0000         4 01 0007 3617 0004
   4 11 0006 3614 0006         5 14 0004 3620 0004
   5 35 0006 0004 0006         6 22 0003 0004 3304
   6 01 3012 0007 0007         7 14 0007 0000 0004
   7 01 0004 0005 0005      3550 22 0003 0004 3304
3530 01 0005 3627 0005         1 22 0006 0007 3304
```

Calculation of Elementary Functions in Mode II

Special floating-point subroutines have been developed to calculate the values of elementary functions. The argument of the elementary function is stored, as is usual, in location 002, while the result appears in locations 0003 and 0004. After the subroutine has been executed, control is transferred to the next instruction.

The subroutines that have been developed include: $\sqrt{x}$, $\sin x$, $\cos x$, e^x, $\ln x$, $\arcsin x$, $\arccos x$.

The call to a subroutine for computing a given elementary function, realized in the simulated program with the aid of an instruction for a conditional jump to the subroutine, is converted by the simulating routine to the form

V (14)		γ	0001
УПЧ (31)		N	

Here γ is the address of the instruction to which the computer is to go after the call to the subroutine: N is the initial address of the subroutine.

Since in simulation, execution of working-program instructions is carried out with the aid of the F operation, it is not permissible to use subroutines from fixed storage in mode II, since the A-register in the "Kiev" computer has only 10 digit positions.

Special Constants Used in Running of All MP Programs

```
3600 11 0000 0000 0000        3614 14 6314 6314 6320
   1 06 0000 0000 0000           5 00 0000 0000 0004
   2 31 0000 0000 0000           6 00 0000 0000 0011
   3 25 0000 0000 0000           7 00 0000 0000 0012
   4 20 0000 0000 0014        3620 00 0000 0000 0020
   5 37 7777 7777 7600           1 05 0000 0000 3402
   6 00 0000 0000 0177           2 05 0000 0000 3362
   7 00 0000 0000 0041           3 00 0000 0000 0075
3610 00 0000 0000 0176           4 00 0000 0000 0014
   1 00 0000 0000 0100           5 00 0000 0000 0101
   2 00 0000 0000 0200           6 12 0000 0000 0000
   3 00 0021 0023 0021           7 20 0000 0000 0003
```

Rounding of Numbers

In the simulation of arithmetic operations, as well as in the formation of numbers whose mantissa and exponent have been separated in a single location, numbers are rounded in accordance with the following rule: if the highest-order digit of the discarded part of the number equals unity, a one is added in the lowest-order digit of the remaining part of the number; if, however, the highest-order digit of the discarded part of the number equals zero, the rest of the number is not changed. When numbers are read out with the aid of the printout subroutine, no provision is made for rounding in order to save time. Rounding of numbers when arithmetic operations are executed in mode II permits an increase in the accuracy with which problems are solved.

TEST ROUTINES FOR THE "KIEV" COMPUTER

The aim of programmed computer checkout consists, on the one hand, in discovering operating malfunctions and, on the other hand, to obtain the maximum possible amount of information on the location of the malfunction. Accordingly, the test routines designed to check out the computer are divided into two categories. Programs of the first type are designed to discover that a given device is not operating properly. They are characterized by the presence of a large number of diverse examples for which the answers are not known in advance. In order to check on the correctness with which a problem is solved, the same answer is obtained by different means. Test routines of the second type are designed to pinpoint the location of a malfunction in an improperly operating device, for example, in a malfunctioning digit position of an arithmetic-unit adder. In such programs, a fairly small number of check problems are solved, and the results are known in advance. Comparison of incorrect results with known results makes it possible to localize the malfunction. At this time, however, there exist no sufficiently perfected methods of analyzing circuits and constructing systems of examples that would bring to light any malfunction.

The methods developed by S.V. Yablonskiy [23] for relay-contact circuits are ill-suited for the case under consideration, since they require a complete list of all possible malfunctions. In this connection, certain simplifying assumptions were made in the development of test routines for the "Kiev" computer. Thus, in particular, it was assumed that for a complete check on each device, it would be sufficient to supply combinations of codes to it such that under proper operation each of its elements would receive all possible combinations of inputs (for example, to each digit position of an arithmetic-unit adder).

The second difficulty consists in the following. In checking out a given device it is necessary to use other devices, which in turn may not be operating properly. Generally speaking, in certain (fairly infrequent) cases malfunctions might appear such that in their presence program checkout methods could not determine which element was operating improperly. To reduce the effect of this circumstance, a large number of similar examples are run with variation in the data affecting the operating regime only of the device being checked out.

In accordance with what we have said, the following test programs have been developed and debugged for programmed checkout of the "Kiev" computer: arithmetic unit; high-speed and passive storage units; control unit; magnetic drums; printout unit.

CONTROL-UNIT (UU) TESTS

Operation of the UU is checked by means of three test routines. The first test routine is designed to check correctness of control transfers. The program provides for transfers of control by all possible methods and, in particular, using the link register, to different memory locations. Here 0 and 1 in all digit positions are transferred to the instruction counter and the link register. For each correct transfer of control, indicators (—0) are transferred into a specially allocated block; these indicators are checked after all transfers and printed out if even one indicator is absent. An incorrect transfer may cause a halt before the indicators are printed out. In this case, they may be printed out by transferring control to the appropriate instruction.

The second test routine checks out the A-register, the address adder (SmA), and loop register Ts, i.e., the execution of group operations. The address counter is checked by calling the contents of several locations with the aid of the F operation. The loop register is checked with the aid of the NGO operation whose execution causes a transfer of control depending on whether or not the contents of the A and Ts registers are the same. Thus the check makes use of the coincidence element, but we need no guarantee that the latter operates correctly, since a malfunction in the coincidence element will be turned up by the codes used.

When the OGO operation is executed, there is an addition in the address adder. The correctness of this addition is checked by the transfer of control after execution of the instruction, i.e., with the aid of the comparison element. An incorrect result causes a halt.

The third UU test routine checks the A and S registers, the address adder, and S counter more completely: proper operation of the S counter as the OZU and PZU address counter; correctness of the transfer of codes consisting of all units and one zero into the A and Ts registers of the S counter, as well as transfers into SmA. Here a check is also made on the operation of digit position 11 of the link register R, which is not checked by the first test. A halt occurs should these devices not operate properly.

<u>The fourth UU test routine</u> checks operation of the SmA using words of the type 0...01...1 + 0...01...1; it checks the indicator of a group operation (GO) in address III of control-transfer operations, as well as in all addresses of the normalization operation, which on the basis of control is the most complex of all the arithmetic operations.

A malfunction causes a halt. Running of UU tests is controlled by the sign of assembly location 3000:

Тесты 1	I	II	III	IV	4 Все
Знак 3000 2	—	—	—	—	+
Начало 3	0007	0125	0240	0566	0007

1) Tests; 2) sign of 3000; 3) beginning; 4) all.

<u>Tests for High-Speed Storage Unit (OZU)</u>

Each OZU location has a magnetic switch for reading. The magnetic switches are located at nodes of the matrix, which is called the coordinate grid. This grid consists of 32 rows (<u>x</u> coordinate) and 32 columns (<u>y</u> coordinate). In writing, 5 binary digits are needed for each coordinate. As an example, for the location with address 1043 (1 000 100 011 in the binary system), x = 17; y = 3; thus the 10-digit address of the number is divided into two five-digit numbers, each of which is applied to the address formers Φ_{ax} and Φ_{ay}.

Operation of the OZU is checked by means of four tests, each of which looks for one type of malfunction:

1) improper reading or writing of codes into or out of OZU location;

2) unstable storage of words in OZU under multiple reading from one location and partial selection;

3) improper reading under large load of Φ_a;

4) improper selection of word on basis of address.

Running of these tests is controlled by the sign of assembly location 3000. If the sign is plus, all four tests run in order (the first follows the fourth), otherwise only the test to which control is transferred from the console will run:

Тесты 1	I	II	III	IV	3 Все
3000	–	–	–	–	+
Начало 2	0022	0045	0100	0117	0022

1) Tests; 2) beginning; 3) all.

With a malfunction of any OZU location, i.e., if an incorrect number is read out, any of the four tests will cause a halt. At that time, the instruction register K will contain the halt instruction

33		k	n

Here the number <u>k</u> indicates the test number; <u>n</u> is the number of readings that have already taken place.

The address of the location whose reading caused the halt is contained in the A-register; the word that it should have contained appears in location 0020.

OZU tests are programmed in two versions; at the beginning of storage (locations 0200-1777 are checked) and end of storage (the location block from 0001 to 1600 is checked).

<u>The first OZU test</u> performs one-shot writing and multiple reading on locations in the tested block, using specially selected words:

00 ... 0; 11 ... 1; 01 ... 0; 10 ... 1

and ones in addresses 0, I, II, and III. In addition, it is possible to write the word set up in NK No. 3007. To do this, it is necessary to set a one into the 41st digit position in NK No. 3001 (a minus). If this is not the case, the words listed above will be written.

<u>Second OZU test</u>. "Tickling" and partial selection. Into locations of the tested block that are arranged on storage-matrix diagonals (Nos. 0000, 0041, 0102, 0143, ..., 1736, 1777), the program writes zeros in all digits, while ones are written into the remaining locations. The diagonal locations are subjected to multiple reading. After this, a check is made on the content of the other locations. This test will turn up one of the two following malfunctions:

1) appearance of a zero in place of a one under multiple reading of zeros or vice versa;

2) partial selection; in this case, the addresses of the malfunctioning locations can be used to determine which of the diagonal locations was being read when the partial selection occurred.

<u>Third OZU test</u> — frequency check of Φ_a. Multiple reading of the word 0101...10 takes place from a group of locations having the common code <u>x</u>:

0000—0037
0040—0077
. . . . .
1740—1777

Fourth OZU test. The memory is filled with the words

```
00 ... 001
00 ... 010
. . . . .
10 ... 000
00 ... 001
. . . . .
```

Next they are read in turn and compared with words formed by the same rule as was used in writing. After this, the test is repeated with the ones complement.

The PZU is checked out by calling its entire contents into the OZU and comparing all codes with those loaded into the computer from punched tape.

Arithmetic-Unit (AU) Tests

The arithmetic unit of the "Kiev" computer includes:

1) receiving registers R_1 and R_2, used to store the multiplicand and multiplier in multiplication and the divisor and quotient in division. Word shifts occur in them in the operations of multiplication, division, normalization, and shifting. The R_1 register is also used as the end-of-division counter, etc.;

2) an adder, in which the operation of inequality is performed as well as transfers from digit to digit;

3) operation register, operation decoder, operation control element, and time control element.

The arithmetic unit is checked by two programs. The first program checks execution of all arithmetic and logical operations. Each operation is executed on a certain group of words, and the results are compared with known results. Here the words are so chosen that all possible combinations of inputs are applied to each of the digit positions of the two registers and the adder.

Certain operations permit partial localization of malfunctioning adder circuits. The operation of logical addition checks the first register and the setting input of the flipflop for each adder digit position; the operation of logical multiplication tests the second register and the counter inputs of the adder flipflops for the initial zero state; the "inequality" operation checks operation of the adder in the absence of carries, while the carry circuits are checked by the operations of addition and cyclic addition; shifts in R_1 and R_2 are checked by the operations of multiplication, division, normalization, and shifting.

After the arithmetic unit has been checked out with the aid of the first program, doubt still remains as to its proper operation since malfunctions might exist that would not be turned up by this test. A more complete check is accomplished by the test of multiplication and division of pseudorandom numbers. This routine is designed to turn up malfunctions in the arithmetic unit and is less convenient from the viewpoint of localization of the improperly operating circuits.

Multiplication is checked in the following manner. A sequence of pseudorandom numbers is generated. Each new number is multiplied twice by the preceding number, with the locations of the factors being interchanged. Since the multiplicand and multiplier play an essentially different role in the execution of the multiplication operation, a malfunction of the adder or one of the registers should cause disagreement in the results of the multiplications. In this routine, multiplication is carried out with rounding.

Division is also checked with the aid of a sequence of pseudorandom numbers. The operation is performed twice on two successive numbers (the smaller is divided by the larger). Here the second division is carried out by a special subroutine using an iteration formula which should yield a quotient having an error not exceeding $\varepsilon = 2^{-36}$. The results should coincide to within ε.

Both tests are punched into a single tape. The first test is divided into three parts: testing of logical operations, the addition operation, and remaining operations. Setting up the different signs in assembly location 3000, the test may be run in parts or all parts may be run in sequence:

Тесты 1	I	II	III	IV	4 Все
Знак НК 3000 . . 2	+	+	+	+	−
Начало 3	0043	0110	0155	0214	0042

1) Tests; 2) sign of NK 3000; 3) beginning;
4) all.

The routine is run repeatedly until the computer is halted from the control console or until an incorrect result appears. If an incorrect result is obtained for one of the first three test segments, a halt will occur and the result of the disagreement between the result obtained and the previously known correct result will appear at the control console (adder signal).

If test IV causes a disagreement in the results of two multiplications or two divisions of exactly the same numbers, the machine will print out both factors (or dividend and divisor) and both results, and will halt. The halt instruction in instruction register K will contain in address III the code for the operation executed (multiplication 11 or division 12).

Printout Test

The number-printing device is tested by printout in octal or decimal of four groups of numbers (see Appendix 2). The order and number of printouts is regulated by the content of location 3000 in the following manner. For 3000 = +0, the first group of codes is printed out multiply.

When 3000 is switched to –0, printing out of the second group commences. Then +0 gives the third group and –0 the fourth group.

It should be recalled that when we go to print out the third group it is necessary to ensure decimal printout. The first two groups are printed out in the octal system. Programs are loaded from the control console by the instruction

21	0001	0655	0001

Magnetic-Drum (MB) Test

The magnetic-drum test verifies that words are written properly onto MB and read properly from MB into OZU.

The test runs in the following sequence:

1) words are written into the first half of the OZU location;
2) words are written into the first drum areas;
3) reading from MB into the second half of the OZU location;
4) word-by-word comparison of codes written onto and read from MB;
5) writing onto next MB segment;
6) go to No. 3.

The sections of the MB test may be of different length, but cannot occupy more than half the OZU, i.e., can contain no more than 500 words. The length of a segment is given in address I of location NK 3000 (n). Address II of NK 3001 specifies the number of segments into which the drum is divided. The test is designed to check two drums. The drum number is given in location 3002

		0k00	b

Here k = 1, 2; b is the initial drum location.

Should words not agree, the computer will halt.

Provision is made for two cases of formation of codes to be written:

a) the code to be written is set up manually in NK 3007, and exactly the same word is written into all drum locations;

b) the code to be written is formed on the basis of a pseudoword program, and a new word is transferred into each OZU location for writing.

Manuscript Page No.	[Footnotes]
140	The MP algorithms were developed by I.V. Sergiyenko and L.N. Ivanenko under the guidance of Ye.L. Yushchenko.

TRANSLATOR PROGRAMS FOR THE "KIEV" COMPUTER

FORMULATION OF PROBLEM

In this chapter, we describe two translator programs (PP) for the "Kiev" computer — PP-AK and PP-2.

The PP-2 is the first of the automatic-programming systems developed for the "Kiev" computer; fundamentally, it is intended for the programming of arithmetic problems. The PP-2 language takes into account several special features of "Kiev" machine language, and is not a general-purpose language such as the address language or Algol [13], i.e., it is intended only for translation for the "Kiev" computer. In this lies the defect of PP-2.

The PP-AK language uses as a source language the address language; very minor restrictions are imposed on the style of the latter [19]. The use of the address language as a source language for all other computers of the VTs AN UkSSR accounts for the extensive success gained in the practical introduction of this PP.

In the development of both types of PP, the following was the basic requirement: the PP should be acceptable in practice and convenient to use. As a result, both PP are intermediate in position between large translator programs and compiler programs. They resemble the latter in compactness and high running speed. In a certain sense, both PP may be considered as loading routines, since the translation of the initial information into "Kiev" machine language is accomplished at the same time as loading, delaying the latter process almost not at all. But the use of the PP makes it possible to load a program into any storage area, reduces the number of working locations used, makes it possible to write the program onto drum, as well as to monitor loading and magnetic writing.

We also note that with the aid of the PP-AK and PP-2 it is possible to obtain programs intended to perform floating-point calculations. One of the following methods may be used for this purpose.

The execution of floating-point calculations may be provided for directly in the writing of an address program which is used in conjunction with the PP to write floating-point programs.

The insertion of the floating-point mode directly into address programs, however, frequently envolves great amounts of effort and in this connection it is possible to use the method given in Chapter IV for simulation of the floating-point mode. Since the use of this method imposes certain restrictions on the set of admissible elementary operations, and also leads to slowing down of the calculations by roughly a factor of 10, it may be employed for isolated program segments; the other segments; the other segments may be executed in fixed-point mode or with floating scale factors.

The utilization of parentheses in formula representations is a special problem. As the Polish mathematician Jan Lukasiewicz has shown, the order of execution of operations may be specified uniquely without the use of parentheses; when we do this, the operation symbol must come first, with the arguments following. As an example, the expression

$$(a+b)\times c=f$$

in the Lukasiewicz representation will have the form

$$=\times + abcf,$$

while the expression

$$\frac{\ln(a+b)}{a-b}\times c=f$$

will have the form

$$= \times : \ln + ab - abcf.$$

The Lukasiewicz representation is no less convenient than the generally-accepted representation of formulas with initial and terminal parentheses, brackets, braces, etc. It is especially convenient in the representation of conversion operators for automatic programming. Here we speak not only of eliminating one class of initial information — the parentheses — which enables us to simplify coding. Parenthesisless representation of formulas possesses one extremely interesting property: the sign on the first operation symbol on the right belongs to the operation being executed. Thus, the parenthesisless representation of formulas permits a substantial simplification in the choice of information for instruction synthesis.

It is simple to go from the generally-accepted notation to the parenthesisless form; moreover, this may easily be done by the computer itself, as in the automation systems given here. The Lukasiewicz representation is used directly by the PP-AA and PP-2 translator programs in a somewhat modified form — it is used from right to left. Our examples would be rewritten as follows:

$$fcba + \times =$$
$$fcba - ba + \ln : \times =$$

This type of representation makes it easy to select information for instruction synthesis. Thus in the first example, the first operation to be performed is the addition a + b or ba +. To establish this, the algorithm scans three elements at a time from left to right for a two-term operation and two elements at a time for a one-term operation. In the second example, the first operation executed will be a — b or ba —, but to find it, the algorithm must scan and analyze one information element (EI) at a time, and find the operation symbol. In virtue of the representation rules, the two EI on the left (one for a one-term operation) will be arguments.

In programming formulas given in parenthesis representation, we first apply the algorithm for translating the parenthesis representation into the parenthesisless form (see [21]) with a simultaneous (formal) check on these representations [22].

TRANSLATOR PROGRAM FOR "KIEV" COMPUTER USING ADDRESS ALGORITHM AS INPUT INFORMATION (PP-AK)*

The advantage of the PP-AK as compared with the translator programs known to us consists primarily in the generality of the algorithmic source language. This feature of the source language makes the PP-AK extremely promising as the number of different types of computer increases.

The tendency to create PP that run quickly and are convenient for practical utilization has led to the need for imposition of certain restrictions. In particular, in PP for the "Kiev" computer there so far is no automated call to external memory.

Despite the fact that PP-AK (without the block for conversion of formulas into parenthesisless form) consists of a total of only 576 instructions (in the decimal system) it uses the universal address language as a source language with very slight restrictions on the style of the language, includes a formula-reduction block, a working-cell reduction block, and a block for evaluation of predicate functions.

The working program compiled by the PP-AK may be punched out onto cards or paper tape.

The average running speed of the PP-AK is 100 instructions per minute (allowing for loading and readout). Input information is supplied by zones. There is provision for automatic monitoring of working-program allocation over the working field.

The optimum length of working programs (most suitable for the PP-AK) runs up to 600 octal codes; to create longer working programs, it is necessary to divide the address algorithm into closed blocks, i.e., into segments with one entrance and one exit.

PP-AK SOURCE LANGUAGE

For this variation of translator program is the address program for the problem. In accordance with this situation, the following address formulas are admissible for the information: transfers (arithmetic operators); program entrances (operators calling up subroutines); predicate formulas (logical operators), and unconditional-Ump labels; halts; loops; printout. In isolated cases, it is permissible to supply information in the form of prepared instructions which the translator program shifts as a whole into the working program. Such information elements are called nonstandard statements.

<u>Transfer formulas</u> form a portion of an address program that realizes one-shot conversion of information in accordance with a certain sequence of address formulas; the result is written into an address specified, in the general case, as the value of some address function. The radix-point mode should be considered when an address algorithm is written.

The transfer of the content of address <u>a</u> into location <u>b</u> is represented as $'a \Rightarrow\rangle 'b$ (as a rule, we write $'a \Rightarrow\rangle b$). in address language). Thus in the address style of PP-AK, the rank of a formula located to the right of the transfer symbol $\Rightarrow\rangle$, is increased by unity as compared with the generally employed method of representation in the address language (we may assume that the symbol $\Rightarrow\rangle$ is replaced by the symbol $\Rightarrow\rangle'$). Addresses of rank zero are used only as address components and as counters in loop statements. The use of addresses of rank zero is forbidden for other cases. An address of rank zero must be the second argument of a statement.

An address-program segment is called a <u>subroutine</u> if it has a single initial entrance label K_v and an exit lable 'ᗺ' (subroutines with <u>n</u> entrances are considered as <u>n</u> subroutines). Address formulas representing the realization of a subroutine with entrance K_v at a given point in the address program are called subroutine entrance formulas; such formulas are represented by ΠK_v.

Subroutine entrance formulas are coded as one-term operations in internal storage.

The entire input information for a subroutine (information on the blocks to be processed and obtained, labels of subroutines to be used, exit label, etc.) are specified in the form '0002 = sequence. For individual subroutines with a small number of parameters, it is necessary to perform parameter transfers directly ahead of the entrance formulas, with the parameters being placed into subroutine working locations (0004, 0003, ...), except for 0002. The parameter that should appear in location 0002 is written as the argument of a single-term operation coding the entrance formula for the given subroutine.

<u>Example</u>. In an address program, the calculation 'a + ln'x with scale factor N on ln should be represented in the form $N \Rightarrow 0004$

$$'a + \ln 'x \Rightarrow 'c.$$

Each subroutine should terminate in the ᗺ.

All lines of the program to which control must <u>necessarily</u> be transferred (and only such lines) are <u>marked off by labels</u> in order of their appearance in the initial information; the labels take the form of a sequence of positive integers (they are numbered starting with 1), and after each label there is the "beginning of operator" sign (designated i..., where <u>i</u> is the operator number). The same labels are used in the representation of values of corresponding predicate formulas.

In addition to operations available in the operation set of the "Kiev" computer, the following operations, which can be simulated by the PP-AK, are admissible:

1. No more than "$\leq$":

$$a < b \Longrightarrow {}'c.$$

The operation means

$$'c = \begin{cases} \text{false when} & a > b; \\ \text{true when} & a < b. \end{cases}$$

2. No more than in absolute value "$| \leq |$":

$$|a| < |b| \longrightarrow {}'c.$$

The operation means

$$'c = \begin{cases} \text{false when} & |a| > |b|; \\ \text{true when} & |a| < |b|. \end{cases}$$

3. Equals "=":

$$(a = b) \Longrightarrow {}'c.$$

Operation means

$$'c = \begin{cases} \text{false when} & a \neq b; \\ \text{true when} & a = b. \end{cases}$$

4. Equal in absolute value:

$$|a| = |b| \Longrightarrow c.$$

The operation means

$$'c = \begin{cases} \text{false when} & |a| \neq |b|; \\ \text{true when} & |a| = |b|. \end{cases}$$

Here a and b are addresses of rank zero, one, or two; the value true is represented by the code −0, and the value false by the code +0.

5. Inversion: not f, where f is any address formula. The logical operations and, or, not are represented by the corresponding words.

A halt formula is represented by the word "halt."

Loop formulas of the following type are admissible:

$$\text{Ц}i\ \{a_0\,(\Delta a)\,a_n \Longrightarrow {}'\pi\}\ K_1,$$

where a_0, Δa, a_n are addresses of rank zero or one.

The formula Ц is placed at the beginning of its domain of influence (at the beginning of the operator), simultaneously replaces the label symbol $\underline{i}$ with the ellipsis, and causes the program segment on the basis of line K_1 exclusively (!) to repeat, assigning successive values to address π ranging from a_0 to a_n with interval $\Delta\alpha$.

Thus, it is possible to specify both numerical values of loop parameters and the addresses containing them; here the numerical values can only be positive octal integers, while the addresses can contain any values. It is necessary to remember that:

a) a_n is given in the form $a_n = \bar{a}_n - \Delta a$, where $\bar{a}_n$ is the last value of the parameter $'\pi$ for which the loop should be executed;

b) if the lower limits of inner and outer loops coincide, two labels must accordingly be placed at the end of the loop.

Example. Let the operator Σ_1 be used in a double loop. The corresponding address program looks like this:

$$\begin{array}{c} \text{Ц}i\,(\ldots)\,i+3 \\ \text{Ц}i+1\,(\ldots)\,i+2 \\ \Sigma_1 \\ i+2\ \ldots \\ i+3\ \ldots \end{array}$$

The operation of reading out (printing out) the content of address $'a$ – an octal or decimal word – is represented in the form

$$\text{Печ}_8{}'a \text{ or } \text{Печ}_{10}{}'a.$$

To read out a sequence of codes, a loop formula is employed.

Example. Print out the number $'(\alpha + 1)$, ..., $'(\alpha + n)$.

$$\begin{array}{c} \text{числа } '(\alpha+1),\ \ldots,\ '(\alpha+n). \\ \text{Ц}K\ \{1\,(1)\,n \Longrightarrow {}'\varphi\}\ K+1 \\ \text{Печ}_{10}{}'('\varphi+\alpha) \\ K+1\ \ldots \end{array}$$

Storage Allocation

Machine memory is allocated in the following manner. Initial data for a problem and fixed addresses are stored at the end of machine memory from location 1777 on; the working program (RP) compiled by the PP-AK is placed from location 0020 downward. The PP-AK selects working locations, distributing them between the program and initial data.

For coding, the address program is represented in symbolic addresses (in letter expressions).

After processing the information, the PP-AK delivers the working program for the problem and a table showing the distribution of labeled lines in computer memory. Without the block for converting formulas to parenthesisless form, the PP-AK takes up 1100 OZU locations (octal), i.e., 576_{10} addresses. Including loading and readout, it runs at a rate of 100 instructions per minute.

<u>Coding Information for PP-AK</u>

The following dimensional data is punched into the fourth tape zone in the indicated order ahead of the basic information:

1) maximum number of labels (with ellipsis) in the problem – n_3;

2) number of locations occupied by numerical information of problem – n_4;

3) length of longest segment of information – n_5;

4) length of first segment of information and number of zone on punched tape into which this segment is coded.

All information on the program is divided into segments not exceeding n_5 elements in length (n_5 is fixed for a given problem). This division is arbitrary, except that the integrity of individual formulas and nonstandard statements should be preserved. Each segment is punched in to a separate tape zone; its information terminates with the "end-of-loaded-segment" symbol, after which there is an indication of the length of the next segment, given in the form

		m	k

Here *m* is the length of the segment; *k* is the number of the punched-tape zone in which this segment appears.

Each individual information element is stored in a single OZU location. The first five digit positions of the location (operation-code bits) are set aside for the indicators that classify the element.

Below we give the rules for coding various classes of information elements.

1. Before a nonstandard statement, there should be its indicator – the governing element

36		*n*	

Here *n* is the number of elements in the nonstandard statement. Following this code there are *n* information elements, which should be placed as a whole into the working program.

2. The beginning of the operator is

15			

3. The end of the information on the problem is

14			

4. The end of the loaded segment is

17			

Coding of numbers. Information on a number should contain the number indicator, the absolute address of the number, and the rank of its address. The number indicator is coded in the operation-code bits of the instruction, address *a* goes into the address-I bits, the rank *r* of the address goes into the address-III bits:

00	*a*		*r*

The rank of a number is arbitrary and may range from 0 upward;

addresses of rank 0 are treated by the PP-AK as corresponding numbers of units in address II. The use of addresses of rank zero is admissible only for address modification and the operation of counters.

Coding of operations. The PP-AK distinguishes among one-term, two-term, and prime operations. The operation of transfer based on an address is treated as a two-term operation.

In addition to the elementary operations available in the operation set of the "Kiev" computer (+, —, |−|, ×, ⊠, :, Л→, ∨, ∧, +Ц, СлК, ≃), certain generalized operations are also admissible; they are simulated by the translator program through the writing of routines for their realization or calling up a standard subroutine where one is available in the OZU. Among the operations simulated by the PP-AK we might note mod a, a^n, as well as certain special operations frequently encountered in programs. This expands the set of computer operations, as it were. As we have already indicated, the PP-AK also simulates certain logical operations.

The simulated operations are coded as:

<	02	1350		r
|=|	02	1366		r
=	02	1404		r
|<|	02	1410		r
1 Отрицание	22	1620		r
2 Степень	22	1626		r
3 Модуль	22	1635		r

1) Negation; 2) degree; 3) absolute value.

Here r is the rank of the result of the operation.

The following codes are introduced for coding of formulas in parenthesis representation:

initial-parenthesis code

04			

terminal-parenthesis code

05			

Formulas are separated from one another by a separating symbol, whose code takes the form

			90

The halt, load, and readout operators are coded as nonstandard statements.

Elementary operations are coded in the form

01	00*k*		*r*

Here <u>k</u> is the octal operation code; <u>r</u> is the rank of the result.

The operation code for $\Longrightarrow$ has the form

03			

The evaluation of functions by a standard subroutine is coded as a one-term operation, whose code corresponds to the address of the first subroutine instruction in ZU or to the subroutine label

21	k_1		*r*

Here k_1 is the subroutine entrance; <u>r</u> is the rank of the result of the operation.

A prime operation is coded in the following form:

a) rank of address if this operation occurs directly before an address;

b) rank of result if it occurs before an operation symbol.

Coding of predicate formulas. The realization of an arbitrary predicate formula

$$P\{F(x_1, \dots, x_s)\} K_1 \downarrow K_2 \qquad (21)$$

may be reduced to calculation of the values of the corresponding predicate function $F(x_1, \dots, x_s)$ and to realization of a predicate formula having the form

$$P\{{}^r a < 0\} K_1 \downarrow K_2, \qquad (22)$$

where ${}^r a$ is an address of rank r used as the basis for storing the computed value of the predicate function.

Predicate formulas of form (22) are said to be elementary. A predicate formula of the general form (21) is coded as

$$F(x_1, \dots, x_s) P K_1 \downarrow K_2.$$

The translator routine programs evaluation of the predicate function F by an arithmetic operator. Two locations are set aside for the coding of elementary predicate functions; data is stored in them in the following manner:

16	a		r
		K_2	K_1

If the predicate formula (22) occurs in the information directly after evaluation of a predicate function, and the value of the latter has been obtained in a working location, then zeros are used in place of a and r when the predicate is coded.

If K_1 (or K_2) is the label for a statement occurring directly after a predicate formula, then this statement need not be marked off specially, and in coding, K_1 (or K_2, accordingly) is replaced by a zero.

Information on loops is coded as follows: the loop formula

$$Ц\,({}^{r_1}a_0\,({}^{r_2}\Delta a_0)\,{}^{r_3}a_{\text{конечн}} \Longrightarrow)\,'\pi)\,K$$

is coded in two OZU locations in the form

$K+1$	15	π	$a_{\text{конечн}}$	K_1
$K+2$	r_3	a_0'	Δa_0	$r_1 + r_2$

Here r_1, r_2 and r_3 equal 0 or 1. When $r_1 = 1$, a 1 is placed into the 2nd bit of location K + 1, while when r = 2, a 1 is placed into the 1st bit of location K + 2.

Programming Algorithms

The translator program consists of the following blocks: governing algorithm; arithmetic; processing of nonstandard statements; storage of statement addresses; assignment of absolute addresses; processing of loop formulas.

The governing algorithm of the PP-AK translator program loads into OZU the next segment of the information on the program, analyzes the next governing element of the information, and depending on its classification transfers control the appropriate PP-AK block.

The arithmetic-statement programming block programs formulas while simultaneously reducing the number of formulas and working locations employed.

The arithmetic block transfers control to the governing statement; the signal for this is the appearance during scanning of an element that is not characteristic, i.e., an element that is neither a variable nor an operation.

The arithmetic-block algorithm fundamentally duplicates the algorithms used by humans in manual programming. Reduction in the number of working locations is achieved within a given statement; reductions in the number of instructions are gained within a given formula. Expansion of the algorithm to reduce the number of instructions outside a given formula is an extremely difficult program in view of the presence of addresses of higher rank in the information.

The order of execution of the arithmetic-block algorithm is as follows:

Search (from left to right) for the first operation to be executed. The information is scanned until the first operation to be executed is found.

Construction of an instruction (or several instructions) realizing the operation that has been found, and transfer of the result into the next free working location.

Transfer of address of result of programmed operation with indicator of working location into initial information at location of code for this operation.

Compression of information: the information from the beginning of the operator up to the location storing address I, on which the operation is to be performed, is tightened up all the way to the location that now contains the result of the programmed operation.

Example. Let the following arithmetic operator be programmed:

$$\frac{('a + 'b)}{'c} \times \sin 'a \Longrightarrow 'd.$$

In parenthesisless notation, the formula will look like this:

$$dcba + : a \sin \times \Longrightarrow.$$

Let the information on the formula occupy locations from α to $\alpha +$ + 9

...	d	c	b	a	$+$	$:$	a	sin	$\times$	$\Longrightarrow$	...	...
	α	$\alpha+1$	$\alpha+2$	$\alpha+3$	$\alpha+4$	$\alpha+5$	$\alpha+6$	$\alpha+7$	$\alpha+8$	$\alpha+9$	$\alpha+10$	$\alpha+11$

The first operation to be performed will be the addition a + b. After this operation has been programmed and the result transferred to some working location r, as a result of compression the information will have the form

...	...	...	d	c	r	:	a	sin	×	$\Longrightarrow$	...
	α	$\alpha+1$	$\alpha+2$	$\alpha+3$	$\alpha+4$	$\alpha+5$	$\alpha+6$	$\alpha+7$	$\alpha+8$	$\alpha+9$	

Location $\alpha + 2$ becomes the beginning of the statement.

Reduction in number of instructions at the initial-information level. If within a given formula we find an operation to be executed for which the information coincides with the information on an executable operation that has already been found, the address of the result of the first of these operations is assigned the indicator of a standard location and this together with the indicator of a working location is transferred to the location of the operation code of the analogous formula, after which the information is again compressed. The search for the next analogous operation continues until the transfer sign $\Longrightarrow$, is encountered, indicating the end of the formula.

Thus, if a formula contains several analogous operations, after the first such operation has been programmed, in place of information on each of them, there will be information on the address of the result and for all formulas except the last the address of the result will be provided with the standard-location indicator, while the last will be provided with the working-location indicator.

Search for next operation to be executed. Searching continues until the entire information on the arithmetic operator has been processed into the program for the problem.

If the addresses contained in a considered operation to be executed are addresses of higher rank, the special address-rank reduction routine will run before the operation is programmed. This routine forms the address-rank reduction instruction if a is an address of rank two,

Φ	a		—

while if the rank of the address a > 2, the following instructions are formed:

Φ	a	—	—	
Φ	4000	—	—	
. . .	. . .	. . .	. . .	(n — 2) times
Φ	4000			

Before these instructions are written into the program, a check is made to ensure that the A-modification register does not contain the required address and that the content of the given address of higher rank has not changed in the A-register at the instant of calling if the call has taken place (i.e., that the given address of higher rank is not contained in address III of the instructions of the working program that follow the instruction calling it to the A-register). Where conditions are favorable, the F instructions can be eliminated.

If some operation K is performed on addresses of which only one is an address of rank two, then the following instructions are written:

Φ	b	—	—
K	4000	a	r
. . .	. . .	. . .	. . .

If both addresses are of rank two, then the operation on them is programmed in the form

Φ	b	—	r
Φ	a	—	—
K	r	4000	r

Instructions are programmed similarly for ranks exceeding two. Operatons with addresses of rank zero are programmed in a special manner.

If in the initial information an operation is preceded by the code for a transfer on the basis of an address $\Rightarrow$, this means that a change is made in the content of the address for rank one by <u>n</u> units. In this case, the translator program writes the instruction

+	a	r_n	b

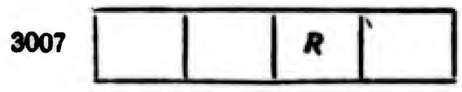
Here <u>a</u> is the modified address; r_n is the address into which the PP-AK places the shift constant <u>n</u> after first varifying that it is not contained among the PZU numbers or among constants formed earlier.

If the result of an operation is an address for another operation, programming of an operation of rank 0 must be performed directly before programming of the operation for which it forms the address. Here the address of rank 0 is stored with the modification indicator in the instruction, while the address of higher rank is placed into the A-register.

<u>Reduction in number of working locations</u>. The translator program sets aside a special block for working locations; only the lower limit is specified (the upper limit may also be specified since the maximum number of working locations equals the number of operations to be executed simultaneously that enter into a single formula).

Each working location into which a result is placed is marked off in the block by transfer into it of a special code (—0). If a location with the indicator rab then participates in the operation as performed, according to the instruction-reduction algorithm, it will be taken to be free from this time on, and the code +0 will be placed into it. Thus, all working locations that are free at a given instant will contain +0, while those that are bound will contain —0. When it is necessary to store the result of an operation in a working location, the PP-AK looks through the block of working locations for the first location from the bottom that is free, i.e., the first location from the bottom containing +0.

Assignment of absolute addresses. In allocating machine memory, it is necessary to take into account the fact that the construction of algorithms by the PP-AK requires that the initial data for the program be stored at the end of memory; the program for the problem will be written from the OZU location specified in assembled word 3007:



The PP-AK obtains information on the absolute addresses of numbers and itself selects working locations, so that all instructions that it compiles are written in absolute addresses. Instructions realizing conditional jumps and operations on codes contain in place of the instruction addresses the labels for lines — the numbers of statements to which control is to be transferred. An exception is represented by instructions calling up standard routines, which are also written in absolute addresses. In the course of programming problems, the PP-AK compiles a table of statement storage allocation, in which the ith line (location) contains the absolute address of the first instruction for the ith statement of the problem.

After programming, a special PP-AK block begins to run, the so-called absolute-address assignment block. In all instructions containing numbers of statements, these numbers are replaced by the addresses of the first instructions of the statements in accordance with the statement-allocation table:

EXAMPLES OF PROGRAMS COMPILED BY THE PP-AK

I. Let us consider an algorithm for the method of conjugate gradients following the first Gauss transformation used to solve systems of linear algebraic equations.

Description of the Method

Let A be a nonsingular matrix. From the system $AX = F$, the first

Gauss transformation produces the system

$$A'AX = A'F.$$

Application of the method of conjugate gradients to this system yields

$$X = X_0 + \sum_{i=1}^{n} a_i s_i;$$
$$a_i = \frac{(\bar{r}_{i-1}, \bar{r}_{i-1})}{(s_i, A'As_i)};$$
$$s_1 = \bar{r}_0;$$
$$\bar{r}_i = \bar{r}_{i-1} - a_i A'As;$$
$$s_{i+1} = \bar{r}_i + b_i s_i;$$
$$b_i = \frac{(\bar{r}_i, \bar{r}_i)}{(\bar{r}_{i-1}, \bar{r}_{i-1})}.$$

Here $\bar{r}_i$ is the discrepancy for the transformed system. It is clear that $\bar{r}_i = A'r_i$, where r_i is the discrepancy for the initial system. Taking this into account and transforming the scalar product, we arrive at the calculation formulas:

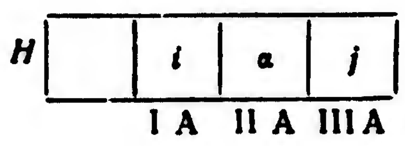
$$X = X_0 + \sum_{i=1}^{n} a_i s_i;$$
$$a_i = \frac{(A'r_{i-1}, A'r_{i-1})}{(As_i, As_i)};$$
$$s_1 = A'r_0;$$

$$r_i = r_{i-1} - a_i As_i;$$
$$s_{i-1} = A'r_i + b_i s_i;$$
$$b = \frac{(A'r_i, A'r_i)}{(A'r_{i-1}, A'r_{i-1})}.$$

Address Program

A vector s_1 is given in the form of an A-sequence, a vector As_1 as a B-sequence, a vector r_1 as a C-sequence. The solution of system X will be obtained in the form of a D-sequence.

In connection with the fact that the initial matrix may contain many zeros, the information is specified in the form of a block of numbers and scales



Here $\underline{i}$ is the number of the line; $\underline{j}$ is the number of the column; and α is the address of the nonzero element.

The number of scales equals the number of nonzero matrix elements. To two equal matrix elements there corresponds a single number in the block of numbers and two scales with different second addresses.

The block of scales is specified in the form of an H-sequence, and n_1 is the length of this block. The beginning of the block of free terms of the system is given, while all other sequences are obtained in accordance with the program.

The program includes three subroutines: 1) scalar product of vector; 2) product of transposed matrix by a vector; 3) product of matrix by a vector.

After [n/2] loops the computer should check to see whether the elements of the C-sequence are less than the given ε. If any of the elements is not less than ε, then [n/2] more loops should be executed. If all elements of the C-sequence are less than ε (in absolute value), a check is made, i.e., the C-sequence is calculated again. If the elements of the newly computed C-sequence are less than ε, the D-sequence is generated — this is the system solution; in the opposite case, computations begin again with the newly obtained C-sequence.

<u>Address Algorithm in PP-AK Style</u>

```
1 ...                                   11 4001 0006 0006
     HC  —   0012  —                    01 0006 0003 0003
     01  0000 0000 0003                 01 0002 3011 0002
     34  0002 0000 0005                 01 0004 3011 0004
     34  0004 0000 0006                 02 0005 3011 0005
     34  0002 0000 0000                 31 0005      3146

                  2 ...
                       '3012 ⟹ 'j
                  3 ...
                       '3010 ⟹ 'i
                       '3011 ⟹ 'h
                           0 ⟹ 's
```

```
4 ...
    $'('H + 'h) \wedge '3024 \Longrightarrow 'r_1$
    $P\{'r_1 = 'j\}\ 5 \downarrow 0$
    $'h + '3011 \Longrightarrow 'h$
    $P\{'h < n_1\}\ 4 \downarrow 7$
5 ...
    $'('H + 'h) \wedge '3022 \Longrightarrow 'r_1$
    $P\{'r_1 = 'i\}\ 0 \downarrow 6$
    $'i \times '3040 \Longrightarrow 'r_1$
    $^2('H + 'h) \times '('Cr + 'r_1) + 's \Longrightarrow 's$
    $'h + '3011 \Longrightarrow 'h$
6 ...
    $'i + '3010 \Longrightarrow 'i$
    $P\{'h < n_1\}\ 0 \downarrow 7$
    $P\{'i < 'r_3\}\ 4 \downarrow 0$
7 ...
    $'j : '3040 \Longrightarrow 'r_1$
    $'s \Longrightarrow '('Ar + 'r_1)$
    $'j + '3012 \Longrightarrow 'j$
    $P\{'j < 'r_4\}\ 3 \downarrow 8$
10 ...
    $'3010 \Longrightarrow 'i$
    $'3011 \Longrightarrow 'h$
11 ...
    $'3012 \Longrightarrow 'j$
    $0 \Longrightarrow 's$
12 ...
    $'('H + 'h) \wedge '3024 \Longrightarrow 'r_1$
    $P\{'r_1 = 'j\}\ 0 \downarrow 13$
    $'j : '3040 \Longrightarrow 'r_1$
    $^2('H + 'h) \times '('Ar + 'r_1) + 's \Longrightarrow 's$
    $'h + '3011 \Longrightarrow 'h$
13 ...
    $'j + '3012 \Longrightarrow 'j$
    $P\{'j < 'r_4\}\ 12 \downarrow 0$
    $'i \times '3040 \Longrightarrow 'r_1$
    $'s \Longrightarrow '('Br + 'r_1)$
    $'i + '3010 \Longrightarrow 'i$
    $P\{'i < 'r_3\}\ 11 \downarrow 8$

Beginning of program ...
    $n : '3040 \Longrightarrow 'r_3$
    $n \times '2040 \Longrightarrow 'r_4$
    $n - '3011 \Longrightarrow 'r_2$
    $n + '3011 \Longrightarrow 'r_1$
    $'C + 'r_1 \Longrightarrow 'R$
    $'R + 'r_1 \Longrightarrow 'A$
    $'A + 'r_1 \Longrightarrow 'B$
    $'B + 'r_1 \Longrightarrow 'D$
14 ...
    $'C \Longrightarrow 'Cr$
    $'A \Longrightarrow 'Ar$
    $\pi 2$
    $n \Longrightarrow {}^2A$
    $'A + '3011 \Longrightarrow '0004$
    $\pi 1\,(A) \Longrightarrow '\beta_1$
15 ...
    $Ц16\left\{0\,(1)\left(\left[\frac{n}{2}\right] - '3011\right) \Longrightarrow 's_1\right\}25$
    $Ц17\,\{1\,(1)\,'r_2 \Longrightarrow 'k\}\,20$
    $0 \Longrightarrow '('D + 'k)$
```

```
20 ...
        $'A \Rightarrow 'Ar$
        $'B \Rightarrow 'Br$
        π10
        $n \Rightarrow {}^{2}B$
        $'B + '3011 \Rightarrow '0004$
        $\pi 1\,(B) \Rightarrow '\alpha$
        Ц21 $\{1\,(1)\,'r_2 \Rightarrow 'k\}$ 22
        $'('C + 'k) - \frac{'\beta_1 \times '('B + 'k)}{'\alpha} \Rightarrow '('C + 'k)$
22 ...
        $'C \Rightarrow 'Cr$
        $'B \Rightarrow 'Br$
        π2
        $n \Rightarrow {}^{2}B$
        $'B + '3011 \Rightarrow '0004$
        $\pi 1\,(B) \Rightarrow '\beta_2$
        Ц23 $\{1\,(1)\,'r_2 \Rightarrow 'k\}$ 24
        $'('D + 'k) + \frac{'\beta_1 \times '('A + 'k)}{'\alpha} \Rightarrow '('D + 'k)$
        $'('B + 'k) + \frac{'\beta_2 \times '('A + 'k)}{'\beta_1} \Rightarrow '('A + 'k)$
24 ...
        $'\beta_2 \Rightarrow '\beta_1$
25 ...
        Ц26 $\{1\,(1)'\,r_3 \Rightarrow 'i\}$
        P $\{|'('C + 'i)| < \varepsilon\}$ 26 ↓ 15
27 ...
        $'D \Rightarrow 'Ar$
        $'C \Rightarrow 'Br$
        π10
        Ц30 $\{1\,(1)'\,r_2 \Rightarrow 'k\}$ 31
        $'('R + 'k) - '('C + 'k) \Rightarrow '('C + 'k)$
31 ...
        Ц32 $\{1\,(1)'\,r_3 \Rightarrow 'k\}$ 33
        P $\{|'('C + 'k)| < \varepsilon\}$ 32 ↓ 14
33 ...
        Ц34 $\{1\,(1)'\,r_2 \Rightarrow 'k\}$ 35
        $'('D + 'k) \Rightarrow '0002$
```

HC	—	0002	—
30	3026	0000	3100
22	0003	0003	0000

```
35 ...
        Halt
```

The lines from label 1 to label 2 are the subroutine for the scalar product of vectors (given as a nonstandard statement); the lines from marker 2 to 10 are the subroutine for multiplication of a transposed matrix by a vector; the lines from label 10 to 13 inclusive are the subroutine for the product of a matrix by a vector.

Storage Allocation

0001	— j	1774	— r_3
0002	— Cr, Br	1775	— r_4
0003	— i	1776	— k
0004	— Ar	1777	— s_1
0005	— h	3000	— H
0006	— r_1	3001	— C
0007	— r_2	'3002	— n
0010	— s	'3003	— n_1
0011	— R	'3004	— $\left[\frac{n}{2}\right]$
0012	— A	'3005	— ε
0013	— B		
0014	— D		
0015	— β_1		
0016	— α		
0017	— β_2		

Working Program Produced by PP-AK

```
0020 01 0000 0000 0003
   1 34 0002 0000 0005
   2 34 0004 0000 0006
   3 34 0002 0000 0000
   4 11 4001 0006 0006
   5 01 0006 0003 0003
   6 01 0002 3011 0002
   7 01 0004 3011 0004
0030 02 0005 3011 0005
   1 31 0005 0022 3146
   2 01 3012 0000 0001
   3 01 3010 0000 0003
   4 01 3011 0000 0005
   5 01 0000 0000 0010
   6 01 3000 0005 0566
   7 34 0566 0000 0000
0040 15 4000 3024 0006
   1 16 0006 0001 0045
   2 01 0005 3011 0005
   3 02 0005 3003 0566
   4 31 0566 0071 0036
   5 01 3000 0005 0566
   6 34 0566 0000 0000
   7 15 4000 3022 0006
0050 02 0006 0003 0566
   1 02 0003 0006 0565
   2 15 0565 0566 0566
   3 31 0566 0065 0054
   4 11 0003 3040 0006
   5 01 0002 0006 0566
   6 01 3000 0005 0565
   7 34 0565 0000 0000
0060 34 4000 0000 0565
   1 34 0566 0000 0000
   2 11 0565 4000 0565
   3 01 0565 0010 0010
   4 01 0005 3011 0005
   5 01 0003 3010 0003
   6 02 0005 3003 0566
   7 31 0566 0071 0070

0070 04 0003 1774 0036
   1 12 0001 3040 0006
   2 01 0004 0006 0566
   3 34 0566 0000 0000
   4 01 0010 0000 4000
   5 01 0001 3012 0001
   6 04 0001 1775 0033
   7 32 0000 0000 0000
0100 01 3010 0000 0003
   1 01 3011 0000 0005
   2 01 3012 0000 0001
   3 01 0000 0000 0010
   4 01 3000 0005 0566
   5 34 0566 0000 0000
   6 15 4000 3024 0006
   7 02 0006 0001 0566
0110 02 0001 0006 0565
   1 15 0565 0566 0566
   2 31 0566 0124 0113
   3 12 0001 3040 0006
   4 01 0004 0006 0566
   5 01 3000 0005 0565
   6 34 0565 0000 0000
   7 34 4000 0000 0565
0120 34 0566 0000 0000
   1 11 0565 4000 0565
   2 01 0565 0010 0010
   3 01 0005 3011 0005
   4 01 0001 3012 0001
   5 04 0001 1775 0104
   6 11 0003 3040 0006
   7 01 0002 0006 0566
0130 34 0566 0000 0000
   1 01 0010 0000 4000
   2 01 0003 3010 0003
   3 04 0003 1774 0102
   4 32 0000 0000 0000
   5 12 3002 3040 1774
   6 11 3002 3040 1775
   7 02 3002 3011 0007

0140 01 3002 3011 0006
   1 01 3001 0006 0011
   2 01 0011 0006 0012
   3 01 0012 0006 0013
   4 01 0013 0006 0014
   5 01 0012 0000 0004
   6 01 3001 0000 0002
   7 30 3026 0150 0032
0150 34 0012 0000 0000
   1 01 3002 0000 4000
   2 01 0012 3011 0004
   3 01 0012 0000 0002
   4 30 3026 0155 0020
   5 01 0003 0000 0015
   6 02 3004 3011 0006
   7 02 0000 3011 1777
0160 01 1777 3011 1777
   1 02 3011 3011 1776
   2 01 1776 3011 1776
   3 01 0014 1776 0566
   4 34 0566 0000 0000
   5 01 0000 0000 4000
   6 04 1776 0007 0162
   7 01 0012 0000 0004
0170 01 0013 0000 0002
   1 30 3026 0172 0100
   2 34 0013 0000 0000
   3 01 3002 0000 4000
   4 01 0013 3011 0004
   5 01 0013 0000 0002
   6 30 3026 0177 0020
   7 01 0003 0000 0016
0200 02 3011 3011 1776
   1 01 1776 3011 1776
   2 01 3001 1776 0566
   3 01 0013 1776 0565
   4 34 0565 0000 0000
   5 11 0015 4000 0564
   6 12 0564 0016 0564
   7 34 0566 0000 0000
```

```
0210 02 4000 0564 4000
   1 04 1776 0007 0201
   2 01 0012 0000 0004
   3 01 3001 0000 0002
   4 30 3026 0215 0032
   5 34 0013 0000 0000
   6 01 3002 0000 4000
   7 01 0013 3011 0004
0220 01 0013 0000 0002
   1 30 3026 0222 0020
   2 01 0003 0000 0017
   3 02 3011 3011 1776
   4 01 1776 3011 1776
   5 01 0014 1776 0566
   6 01 0012 1776 0565
   7 34 0565 0000 0000
0230 11 0015 4000 0564
   1 12 0564 0016 0564
   2 34 0566 0000 0000
   3 01 4000 0564 4000
   4 01 0012 1776 0566
   5 34 0566 0000 0000
   6 11 0017 4000 0565
   7 12 0565 0015 0565
0240 01 0013 1776 0564
   1 34 0564 0000 0000
   2 01 4000 0565 0564
   3 34 0566 0000 0000
   4 01 0564 0000 4000
   5 04 1776 0007 0224
   6 01 0017 0000 0015
   7 04 1777 0006 0160
```

```
0250 02 3011 3011 0003
   1 01 0003 3011 0003
   2 01 3001 0003 0566
   3 34 0566 0000 0000
   4 06 4000 3005 0566
   5 31 0566 0156 0256
   6 04 0003 0007 0251
   7 01 0014 0000 0004
0260 01 3001 0000 0002
   1 30 3026 0262 0100
   2 02 3011 3011 1776
   3 01 1776 3011 1776
   4 01 3001 1776 0566
   5 01 0011 1776 0565
   6 34 0565 0000 0565
   7 34 0566 0000 0000
0270 02 0565 4000 4000
   1 04 1776 0007 0263
   2 02 3011 3011 1776
   3 01 1776 3011 1776
   4 01 3001 1776 0566
   5 34 0566 0000 0000
   6 06 4000 3005 0566
   7 31 0566 0145 0300
0300 04 1776 0007 0273
   1 02 3011 3011 1776
   2 01 1776 3011 1776
   3 01 0014 1776 0566
   4 34 0566 0000 0002
   5 30 3026 0312 3100
   6 22 0003 0003 0313
   7 04 1776 0007 0302
0310 33 0000 0000 0000
```

II. Let us give programs for the direct and inverse directions of solution for a system of linear algebraic equations with symmetric matrix, making use of the improved Gauss method (see Chapter III).

<u>Address Algorith in PP-AK Style</u>

Forward direction ...

```
$'\psi - 1 \Rightarrow\rangle\ 'r$
$0 \Rightarrow\rangle\ 's$
$1 \Rightarrow\rangle\ '\delta$
1...
$'s + '\delta \Rightarrow\rangle\ 's$
$'s \Rightarrow\rangle\ 's_1$
$'\delta \Rightarrow\rangle\ 'q$
$1 \Rightarrow\rangle\ '\delta_1$
2...
$'\varphi \Rightarrow\rangle\ '\beta$
$1 \Rightarrow\rangle\ '\pi$
3...
$'\beta + '\pi \Rightarrow\rangle\ '\beta$
$'('\varphi + 's_1 + '\delta + 1) - '('\varphi +$
$+ 's + '\pi) \times '('\varphi + 's_1 + '\pi) :$
$: {}^2\beta \Rightarrow\rangle\ '('\varphi + 's_1 + '\delta + 1)$
$'\pi + 1 \Rightarrow\rangle\ '\pi$
$P\{'\pi \leqslant '\delta\}\ 3 \downarrow 0$
$'q + 1 \Rightarrow\rangle\ 'q$
$'s_1 + 'q \Rightarrow\rangle\ 's_1$
$'\delta_1 + 1 \Rightarrow\rangle\ '\delta_1$
$P\{'\delta_1 \leqslant ('\psi - '\delta + 1)\}\ 2 \downarrow 0$
$'\delta + 1 \Rightarrow\rangle\ '\delta$
$P\{'\delta \leqslant 'r\}\ 1 \downarrow 0$
B
```

Initial address mapping

β	0001	q	0006
r	0002	δ_1	0007
s	0003	π	0010
δ	0004	r_1	0011
s_1	0005	φ	3000
		ψ	3001

Working Program Generated by PP-AK

(forward direction)

```
0020 02 3001 3011 0002        0040 34 0001 0000 0000
   1 01 0000 0000 0003           1 12 0075 4000 0075
   2 01 3011 0000 0004           2 01 0077 0010 0077
   3 01 0003 0004 0003           3 34 0077 0000 0000
   4 01 0003 0000 0005           4 11 0075 4000 0075
   5 01 0004 0000 0006           5 34 0076 0000 0000
   6 01 3011 0000 0007           6 02 4000 0075 4000
   7 01 3000 0000 0001           7 01 0010 3011 0010
0030 01 3011 0000 0010        0050 04 0010 0004 0031
   1 01 0001 0010 0001           1 01 0006 3011 0006
   2 01 3000 0005 0077           2 01 0005 0006 0005
   3 01 0077 0004 0076           3 01 0007 3011 0007
   4 01 0076 3011 0076           4 01 3001 3011 0077
   5 01 3000 0003 0075           5 02 0077 0004 0077
   6 01 0075 0010 0075           6 04 0007 0077 0027
   7 34 0075 0000 0075           7 01 0007 3011 0007
                              0060 04 0004 0002 0023
                                 1 33 0000 0000 0000
```

Address Algorithm in PP-AK Style

Inverse direction ...

```
'n ⟹ 'i
0 ⟹ 'j
'φ − 'n ⟹ 'D
1...
'φ − 'n − j ⟹ 'B
'n − 'i ⟹ 'r
0 ⟹ m'; 0 ⟹ 'k; 0 ⟹ 's
2...
P {'k = 'r} 3 ↓ 0
'B − 'm ⟹ 'B
'('A − 'k) × ²B + 's ⟹ 's
'k + 1 ⟹ 'k
'n − 'k ⟹ 'm
P₀ → 2
3...
's + '('φ − 'j) ⟹ 'r
0 − 'r ⟹ 'r
'r : ²D ⟹ '('A − 'j)
'D − 'i ⟹ 'D
'j + 1 ⟹ 'j
'i − 1 ⟹ 'i
P {'i = 0} 0 ↓ 1
'n − 1 ⟹ 'r
4...
'A − 'r ⟹ '0002
УПП 3026   ↓   3100
22     0003 0003   ↓
'r − 1 ⟹ 'r
P {0 < 'r} 4 ↓ 0
Я
```

Initial address mapping

φ	3000	r	0005
n	3001	m	0006
D	0001	k	0007
B	0002	s	0010
j	0003	A	0011
i	0004		

Working Program Generated by PP-AK

(inverse direction)

```
0020 01 3001 0000 0004
   1 01 0000 0000 0003
   2 02 3000 3001 0001
   3 02 3000 3001 0077
   4 02 0077 0003 0002
   5 02 3001 0004 0005
   6 01 0000 0000 0006
   7 01 0000 0000 0007
0030 01 0000 0000 0010
   1 16 0007 0005 0043
   2 02 0002 0006 0002
   3 02 0011 0007 0077
   4 34 0077 0000 0077
   5 34 0002 0000 0000
   6 11 0077 4000 0077
   7 01 0077 0010 0010
0040 01 0007 3011 0007
   1 02 3001 0007 0006
   2 31 0000 0031 0031
   3 02 3000 0003 0077
   4 34 0077 0000 0000
```

```
   5 01 0010 4000 0005
   6 02 0000 0005 0005
   7 34 0001 0000 0000
0050 12 0005 4000 0077
   1 02 0011 0003 0076
   2 34 0076 0000 0000
   3 01 0077 0000 4000
   4 02 0001 0004 0001
   5 01 0003 3011 0003
   6 02 0004 3011 0004
   7 02 0000 0004 0077
0060 02 0004 0000 0076
   1 15 0076 0077 0077
   2 31 0077 0023 0063
   3 02 3001 3011 0005
   4 02 0011 0005 0002
   5 30 3026 0066 3100
   6 22 0003 0003 0067
   7 02 0005 3011 0005
0070 31 0005 0064 0071
   1 33 0000 0000 0000
```

THE PP-2 TRANSLATOR PROGRAM*

The following requirements formed the basis for development of the PP-2:

1. The PP-2 language must be similar to the "Kiev" machine language.

2. The PP-2 should not encumber internal storage.

3. The PP-2 should be convenient to use.

Here cognizance was taken of the fact that the more the source language for the translator program differed from machine language, the more difficult it would be to find ones way around in the working program (RP). In this connection we have extremely accute problem pertaining to debugging of the working programs: where do we look for errors — at the source-language level or in the RP?

Program Operators

In working programs for earlier-model computers, it was necessary to distinguish between readdressing, initialization, readdressing-constant-generation, and other types of operators. For computers having an address-modification register, a simpler classification of program operators can be used: four types of operators are distinguished:

1. Transformation operators of the arithmetic-operator type, which process information exterior to the RP (initial data and data obtained in the course of calculations).

2. Logical operators determining the order in which information transformations are performed and controlling the computational process.

3. Address-modification operators controlling the selection of information from storage while it is being processed in loops.

4. Technical operators controlling the processes of reading information into and out of the computer and exchanging codes with external storage devices and which vary the computer operating regime and cause the computer to halt.

Operator Representation

Let us consider methods of representing operators for the sake of a translator program, without considering their representation in terms of operations of actual computers.

Transformation operators are usually represented by algebraic formulas and formulas of Boolean algebra with addresses of various ranks. We call these input addresses. The set of algebraic operations may include the operations $|-|, \vee, \wedge, E(x)$, etc. A calculation using a formula assumes that a unique result will be obtained. The address into which the result is to be written is called the output address and is indicated by a special symbol.* The general form of the formula will be

$$F(A_1, \ldots, A_n, D) \Rightarrow A^*,$$

where D is the operation set and A_i are the addresses.

One of the methods for representing logical operators is to construct predicate formulas which, depending on the truth value of a given statement will yield a two-direction branch in the computational process. We may also take the path of constructing the set of elementary logical operations (standard predicate formulas) used as the basis for construction of operators for many classes of problems. It is especially convenient to have sets of operators for narrow classes of problems such as, for example, machine translation.

The problem of choosing a set of large-scale operators for a fairly wide class of problems remains as yet unsolved. We might note a solution in which the transfer to a transformation operator is determined by a set of values for logical variables. To represent an operator in this form it is necessary to extract all admissible sets of values of the logical variables and compare them with the numbers of the transformation operators. This type of representation is convenient for logical problems, although difficulties arise during translation into TsAM programs.

We attempted to make the PP-2 as short as possible and, to some degree, slanted it toward arithmetic-problem programming applications; we selected a very simple but not optimum solution, and used as the elementary logical operators the predicate formulas that could be realized by the circuitry of the "Kiev" computer, namely:

$$P\{a < b\}\, x \downarrow n+1;$$
$$P\{|a| < b\}\, x \downarrow n+1;$$
$$P\{|a| = |b|\}\, x \downarrow n+1;$$
$$P\{a < -0\}\, x \downarrow y,$$

where <u>x</u>, <u>y</u> are arbitrary numbers; n + 1 is the number of the instruction following the instruction containing the predicate formula.

In the latter case, the control transfer is written on the basis of the sign of the number. This transfer may be preceded by calculation of the truth value of a certain statement for which the values of the variables are written into the sign digit positions of the storage locations. The calculations themselves are performed by the transformation operator.

Thus, predicate formulas are written in the form of finished instructions for the "Kiev" computer, but in symbolic addresses (x, y). The work of the logical block of the PP-2 consists in assigning absolute addresses.

As for the representation of modification operators for the PP-2, here also we were able to avoid special constructions owing to the direct use in the language, on the one hand, of "Kiev" computer operations and, on the other, of nonstandard operators.

The fundamental role of the technical operators is to control the exchange of codes with external memory units. Some of the operations may be written beforehand, since the volume of numerical material and its storage allocation are known. As for subroutines, the instructions used to write them onto drum are formed by the PP-2 itself at the end of programming. Those technical operators that may be written out prior to programming are represented in "Kiev" machine instructions, and are loaded as nonstandard operators.

Storage Allocation and Coding

In view of the limited capacity of computer high-speed storage, it becomes necessary to break programs down into subroutines and to load them sequentially from drum. The PP-2 makes provision for realizing this process. All of the initial material for programming is divided into segments from which the subroutines are later compiled. At the beginning of each segment there is the NP sign (beginning of subroutine), and at the end, KP (end of subroutine). The last subroutine carries a central-control function (TsU) and has special beginning (NTs) and end (KTs) signs. The subroutines are not joined directly together. Each has its own set of standard locations for storage of initial data and results. Thus at any given moment, high-speed memory should contain a single subroutine. Calling up of subroutines from drum, the required transfers of initial data and the transfer of control to subroutines is realized by the TsU, which is in memory at all times. This yields the storage allocation shown schematically in Table 4. The numbers of the first subroutine instructions are the same. The TsU begins after the longest subroutine.

Since a limited amount of PP-2 initial data is stored in high-speed memory, it is necessary to divided it into zones of k elements each, where k equals half of the corresponding block of OZU positions.

TABLE 4
Memory Allocation

1 Распределение памяти машины «Киев»	2 Оперативная память 1024						3 Пассивная память 512 ячеек
	Γ_4	Γ_x	Γ_2	Γ_3	Γ_1	Γ_0	
4 Для ПП-2	5 Рабочие ячейки ПП-2	7 Массив мест для записи РП	9 Массив рабочих ячеек РП	10 Внешняя информация ПП-2	Словарь ОЗУ 11	Словарь ВЗУ 13	ПП-2
Для РП 4	6 Рабочие ячейки стандартных подпрограмм	Массив подпрограмм Массив ЦУ 8		Внешняя информация РП 12			Стандартные подпрограммы 14

1) Allocation of "Kiev" machine memory; 2) high-speed memory, 1024; 3) passive memory, 512 location; 4) for; 5) PP-2 working locations; 6) working locations for standard routines; 7) block of positions for writing RP; 8) block of subroutines, TsU block; 9) block of RP working locations; 10) external information for PP-2; 11) OZU dictionary; 12) external information for RP; 13) VZU dictionary; 14) standard routines.

Two zones of initial data are initially loaded into the computer; in the course of programming, the information elements (EI) that are used are erased, and the representation is compressed. When there are less than k EI in the block, the next zone is loaded from punched tape. The division into zones is performed mechanically, and has nothing whatsoever to do with the division into subroutines.

Stages and Cycles of PP-2 Operation. The working programs compiled by the PP-2 consist of subroutines and TsU. All subroutines are programmed in the same way. For the TsU, the difference lies in the address of the first instruction. Thus an outer loop consists in the programming of subroutines. Here three stages must be distinguished: the first stage, programming of arithmetic formulas and rewriting of the elements of the remaining operator; the second stage, assignment of absolute addresses; and the third stage, taking of the check sum and writing onto drum.

We have already mentioned the simplicity of the search for the first operation to be executed when the formulas are represented from left to right in the Lukasiewicz form. As soon as an operation is found, an instruction is formed. A special investigation is carried out before address III is filled. If there is an output address in the initial information following an operation, it is written into address III of the instruction. The appearance of an output address indicates the end of formula programming. In the opposite case, a working-location address appears in address III. The number of working locations is reduced while the block is in use. The block is scanned in the direction of decreasing addresses, and the address of the first location containing +0 is inserted in the instruction, while —0 is written into this same address. If the numbers of the working locations appear in addresses I or II of the instruction, the corresponding address is "freed" i.e., a +0 is written into this address. The number of the greatest working location (having the smallest address) is noted and compared with the number of the next instruction. An overlapping of blocks causes an emergency halt of the PP-2.

A single-term operation is programmed in the form of a call to standard subroutines. The number of working locations is reduced as in the preceding case. After a finished instruction has been written into the working program, the initial information of the PP-2 is compressed, i.e., the used elements in the information are erased and the remainder of the block is pulled together; at particular times, the next zone is read. A +0 acts to indicate the end of writing. Nonstandard operators are written after the sign W(n) and are copied into the working program. The process continues until appearance of the "end-of-subroutine" signal. Then the second stage of programming commences. Now the subroutine is scanned and all instructions with symbolic addresses are extracted in turn. The symbolic address is used as a basis for finding the absolute address in the OZU dictionary; this is placed into the instruction. The information element T (transfer in symbolic addresses) is processed specially; it is needed for the organization of control transfers from TsU to subroutines and within subroutines linking back to a definite place in the program. In the instruction $Tx_1 - x_2$, both addresses are symbolic. Here x_1 is the place containing the constant governing return from the subroutine; x_2 is the address of some standard location used as a link register. Subroutines conclude with a transfer of control to this standard location, and since it in turn contains an unconditional-jump instruction, control finally is transferred to the required point in the program. Using the EI F and W, we can construct an arbitrary number of link registers. In a T instruction, in addition to the assignment of absolute addresses, there is replacement of the operation code by a plus sign (transfer).

The third stage is an auxiliary step. The check sum for the subroutine is calculated and written before the first instruction. Next the instructions for writing the subroutine onto drum are generated together with the check sum, and writing is implemented. The instructions for reading from drum are also generated and stored in the VZU dictionary at the N- and (N + 1)-position locations from the beginning. After this, the subroutine is again read into the OZU, summed, and the check sums compared. When they do not agree, an emergency halt of the PP-2 occurs.

Clearly, the element following the KP in the initial information will be the NP or NTs. For the NP, the loop is repeated. For the NTs, the address of the first TsU instruction is established. On the basis of the KTs signal, the computer not only sums and checks, but also reads the entire program out in a form suitable for loading into the computer (it is punched out).

Thus, we have been able to make the PP-2 and "Kiev" languages quite close to one another. To make the representation more compact, two arithmetic-operator information elements are written into a single location although it is permissible to code one element into each location for these operators; it is also permissible to cancel elements erroneously coded twice. The words for all the other operators occupy complete locations. The end-of-block indicator is a one at the end of the representation. As a result, when the PP-2 is arranged in blocks of the interchangeable built-in memory, the PP occupies a total of 300 octal (192 decimal) high-speed storage locations (without the block for conversion to parenthesisless form).

The PP-2 is realized in the form of two blocks — an arithmetic operator that generates and writes arithmetic instructions and reduces the number of working locations needed, and a block for processing the remaining words of the PP-2 language, loading from punched cards, writing onto drum, etc.

Storage allocation. When the PP-2 is used, provision is made for a typical memory allocation (see Table 4). It is first necessary to estimate the length of the working program (RP) to be developed and, if it is found that it will not fit in its entirety into high-speed storage, the best division into subroutines must be found.

Since RP subroutines are stored in external memory (drum) and are read out one above the other, the RP must contain a block that is located at all times in OZU in order to control the process of reading and executing the various manipulations of the information. This block is called the central control (TsU). If the program as a whole will fit into the OZU, then it is the TsU.

Information on memory allocation is stored in assembled words (NK) of storage numbered 3000-3004:

3000			Γ_4	
3001			Γ_3	
3002			k	
3003			$2m$	

Here m is the number of the subroutines used to determine the limit of the VZU dictionary; G_4 is the address of the beginning of the RP representation in the OZU. The drum number and location is coded into NK 3004.

We next determine the length of the external information (VI) for the RP, which includes the initial data and intermediate values of variables. We shall assume that the VI contains n words; then $G_2 = G_1 - n = G_0 - 2m - n$. After this, we find N — the number of operators in subroutines or in the TsU, which gives the length of the OZU dictionary. Then $G_3 = G_1 - N = G_0 - 2m - N$. Now we can compute the length of the zone for the external information of the PP-2. Two zones should fit in the interval (G_2, G_3). Thus taking the difference $G_3 - G_2$ and reducing it to the nearest even [$G_3 - G_2$], we obtain

$$k = \frac{[\Gamma_3 - \Gamma_2]}{2}.$$

The minimum value of G_4 = 31, since locations 0001-0030 are used by the PP-2 itself.

PP-2 Source Language

The types of words used to represent algorithms in PP-2 language are shown in Table 5.

TABLE 5
Types of Words for PP-2 Source Language

24 № типа слова	25 Элемент информации (слово)	26 Символ	27 Кодировка			
			0-8 A	IA	IIA	IIIA
1	Входной адрес	*a*	01	*a*		
2	Выходной »	*a*°	21	*a*		
4	Входной модифицируемый адрес	$\bar{a}$	01	4000 + *a*		
3	Выходной » »	$\bar{a}$°	21	4000 + *a*		
5	Одноместная операция	*s*	23	28 № входной команды подпрограммы		
6	Двухместная »	*P*	03	Код в языке 29 «Киев»		
7	Сравнение со знаком	*Ср*1	04	*a*	*b*	*x*
8	» по модулю	*Ср*2	.05	*a*	*b*	*x*
9	» на совпадение . . .	*Ср*3	16	*a*	*b*	*x*
10	Передача управления по знаку числа	*УПЧ*	31	*a*	*x*	*y*
11	Метка оператора	F_x	11		*x*	
12	» подпрограммы	M_i	10			*i*
13	» нестандартного оператора	▼	17		*n*	
14	Печать	*Печ*	22	*a*	*b*	*x*
15	Пересылка в истинных адресах *И* → *И*	═)	37	*a*		*b*
16	Пересылка в условных адресах *У* → *У*	T_s	36		*x*	*y*
17	Пересылка *И* → *У*	T_s	36	*x*	*a*	
18	Пересылка *У* → *И*	T_s	36	*a*		*x*
19	Начало подпрограммы	НП	12			
20	Конец »	КП	13			*i*
21	Начало центрального управления	НЦ	14			
22	Конец центрального управления	КЦ	15			
23	Конец зоны входной информации	!	07			

<u>Note</u>. a, b) absolute addresses; x, y) operator numbers; i) subroutine number; n) number of EI in nonstandard operator.

1) Entrance address; 2) exit address; 3) modified exit address; 4) modified entrance address; 5) one-term operation; 6) two-term operation; 7) comparison with sign; 8) comparison in absolute value; 9) comparison for coincidence; 10) control transfer on sign of number; 11) operator

label; 12) subroutine label; 13) nonstandard-operator label; 14) print out; 15) transfer in relative addresses I → I; 16) transfer in symbolic addresses U → U; 17) transfer I → U; 18) transfer U → I; 19) beginning of subroutine; 20) end of subroutine; 21) beginning of central control; 22) end of central control; 23) end of input-information zone; 24) number of type of word; 25) information element (word); 26) symbol; 27) coding; 28) No. of entrance instruction of subroutine; 19) code in "Kiev" language.

Types 1-6 are used to represent arithmetic operators.

Type 7-10 are used to represent logical operators. They differ from their corresponding analogs in "Kiev" machine language only in that the corresponding addresses contain the number of the operator to which control is transferred rather than the number of an instruction.

An operator label is placed before the operator to which control is transferred and indicates its number.

The label for a subroutine is placed in the TsU and serves as a signal for loading the subroutine whose number is specified in the label into OZU.

The label for a nonstandard operator is placed ahead of the group of codes introduced into the RP directly in "Kiev" machine language (they are not processed by the PP-2).

It is permissible to use words of types 7-10 in nonstandard operators only if they serve not to transfer control within the nonstandard operator but to transfer control outside it. In this latter case, the numbers of the operators are placed into the addresses. It is permissible to use machine operations carrying numbers of 20 or higher and this in turn expands the PP-2 source language.

If type 16 permits replacement by an arithmetic operator a0 + b*, then types 17-18 play an important role in the language, since they permit operation with instructions and, in particular, their transfer or readdressing (the latter process is absolutely necessary for the floating-point mode used, where the employment of group operations is forbidden).

There are three words or three types of transfer associated with T itself:

$$У \to У;\ И \to У;\ У \to И,$$

where U is a symbolic address and I an absolute address.

Example. We are required to readdress the instruction cba × in addresses I and III (multiplication of a vector by a constant). In the PP-2 language, the representation has the form

$$F(x)\, ab \times c^{*} T_3 (x,\ f)\, f \delta_{101} + f' T_3 (f,\ x).$$

This representation corresponds to the program

$$\begin{array}{rcccc} n & \times & a & b & c \\ n+1 & + & n & - & f \\ n+2 & + & f & \delta_{101} & f \\ n+3 & + & f & - & n \end{array}$$

Here δ_{101} is a constant of the form 00 0001 0000 0001. Instructions n + + 1 and n + 3 are superfluous, but it must be remembered that direct readdressing is seldom required in the "Kiev" computer. The transfer T_3 may be used for initialization of instruction n.

The functions of types 20-23 are specified by their names. Since type 23 indicates the end of writing into a zone, it also occurs in the final position.

A program may be written in symbolic form on one line. Here in place of the address symbol α it is necessary to write the actual value of the address a, i.e., instead of α* we write 21a; in place of $\bar{\alpha}$* we write 214,000 + a; one-term and two-term operations are represented by their symbols ln a, sin $\bar{a}$, +, ×, etc. For control transfers, it is necessary to indicate in parentheses the arguments of the predicate formula Sr2(abx). The same procedure is used when other words are written.

We can write a program directly in PP-2 language on blanks for punching. Here it is necessary to remember that information elements for arithmetic operators are placed two in a location, while the indicator for a second word in a location is written into address II. We write a as follows:

03	0001	0001	a

If the representation of an arithmetic operator ends with the first half of the location, the second half is left empty. The other words of the PP-2 language occupy complete locations.

The representation of the initial information is mechanically divided into zones of equal length and the symbol "!" (code 07) is placed at the end of each zone. The zones are numbered from 0001.

The last zone may not be full, so that after the "!" symbol, it should be filled out with markers (zeros) to the prescribed number of k markers.

Arithmetic Block of the PP-2

The arithmetic block of the PP-2 programs arithmetic formulas of any level consisting of variables, two-term operations, and one-term operations, written in parenthesisless Lukasiewicz form, from right to left.

Preparation of external information for arithmetic block of PP-2. The arithmetic block forms instructions for routines running in fixed-point mode, so that scale factors must be provided by the programmer in advance and introduced into the arithmetic formula.

Each variable of an arithmetic formula must be associated with a storage location in the "Kiev" computer, which we shall call the entrance location, and a location must be selected in advance to receive the result of the calculation using the given formula, and we shall call this the exit location.

As an example, let us look at the formula

$$f = (b-a)\frac{d}{a\ln(b+c)}, \qquad (23)$$

where a, b, c and d are certain numbers, and f the result of the calculation by the formula. We store these numbers, respectively, in locations H_1, H_2, H_3, and H_4, while we place the result f into exit location H*:

$$H^* = \frac{H_4}{H_1 \ln(H_2 + H_3)}(H_2 - H_1). \qquad (24)$$

The representation of Formula (24) in parenthesisless Lukasiewicz form will look like this:

$$H_1H_2 - H_1H_2H_3 + \ln \times H_4 : \times H^*. \qquad (25)$$

Coding elements of arithmetic formulas.

In order to load arithmetic formulas into the computer, each element must be coded into 17 digit positions of a "Kiev" machine location. Thus, two elements will fit into each location of the "Kiev" computer.

The indicators permitting differentiation between variables and operations are coded into the first five digit positions of each group. The next 12 bits contain:

a) the address containing the variable if the coded element is a variable;

b) the number of the first instruction of the subroutine if the coded element is a one-term operation;

c) the operation code if the coded element is a two-term operation.

Exit locations are coded in the same manner as variables, except that a special indicator is contained in the first five bit positions (see Table 5). If H_1, H_2, H_3 and H_4 correspond to locations 0007, 0010, 0011, 0012, and H* corresponds to 0014, then the elements of Formula (24) are coded separately in the following manner:

H_1	01 0007	ln	23 3116
H_2	01 0010	$\times$	03 0011
$-$	03 0002	H_4	01 0012
H_1	01 0007	:	03 0012
H_2	01 0010	$\times$	03 0011
H_3	01 0011	H^*	21 0014
$+$	03 0001		

Storing two elements in each location, we obtain the following octal numbers which together with the elements of the logical operators of the problem form the external information for the PP-2.

```
K+1 01 0007 0001 0010
K+2 03 0002 0001 0007
K+3 01 0010 0001 0011
K+4 03 0001 0023 3116
K+5 03 0011 0001 0012
K+6 03 0012 0003 0011
K+7 21 0014 0000 0000
```

TABLE 6

Operation of Arithmetic Block

№ выполняемой операции[1]	H_1	H_2	$-$	H_1	H_2	H_3	$+$	ln	$\times$	H_4	:	$\times$	H^*	
	U													
		U												
1			U											$- H_2 H_1 r_1$
	r_1	$\times$	$\times$	H_1	H_2	H_3	$+$	ln	$\times$	H_4	:	$\times$	H^*	
				U										
					U									
						U								
2							U							$+ H_3 H_2 r_2$
	r_1	$\times$	$\times$	H_1	r_2	$\times$	$\times$	ln	$\times$	H_4	:	$\times$	H^*	
3								U						$\overset{+}{УПП}$ $\overset{r_2}{3026}$ к. $\overset{-}{5}$ к. $\overset{0002}{3116}$
	r_1	$\times$	$\times$	H_1	r_2	$\times$	$\times$	$\times$	$\times$	H_4	:	$\times$	H^*	$+ 0003 - r_2$
4									U					$\times r_2 H_1$ [illegible]
	r_1	$\times$	$\times$	r_3	$\times$	$\times$	$\times$	$\times$	$\times$	H_4	:	$\times$	H^*	
									.	U				
5											U			$: H_4 r_2 r_3$
	r_1	$\times$	$\times$	r_3	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	H^*	
6												U		$\times r_3 r_1 H^1$
	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	

1) Number of operation to be executed.

The sequence of processing of the initial information and the representation of results for Formula (25) is shown in Table 6, where U is the specifier of the input-information elements. Where boxes have been struck out in Table 6, it indicates erasure of information.

EXAMPLE OF PROGRAM COMPILED BY PP-2

We are required to compute the values of an improper integral using the formula

$$I_n = \int_0^\infty e^{-x} f(x)\,dx \approx \sum_{k=1}^{n} A_k f(x_k)$$

for specified values of the parameters, where the A_k are certain numbers and the function f(x) is given by the relationship

$$f(x_k) = (x_k + 4\lambda)^{2\left(\frac{R}{\lambda} - 1\right)} \psi_0(x_k).$$

With the aid of the arithmetic block of the PP-2, a formula was programmed for I_n at n = 0 and $x_k = x_1$ for k = 1:

$$I_0 = A_1 f(x_1),$$

where

$$f(x_1) = (x_1 + 4\lambda)^{2\left(\frac{R}{\lambda} - 1\right)} \psi_0(x_1);$$
$$\psi_0(x_1) = D_0 B_0(\xi)[b_0(x_1)];$$
$$B_0(\xi) = 2\xi - \gamma;$$
$$b_0(x_1) = (x_1 + 2\lambda)\left[\frac{2\left(\frac{R}{\lambda} - 1\right)}{x + 2\lambda} - 1\right].$$

Here D_0 is a certain constant.

We rewite I_0 in the form

$$I_0 = \exp\left\{\ln A_1 + \ln D_0 + 2\left(\frac{R}{\lambda} - 1\right)\ln(x + 4\lambda) + \ln[12\xi^3 - 6\xi^2\gamma - 8\xi] + \right.$$
$$\left. + \ln\left[2\left(\frac{R}{\lambda} - 1\right) + (x + 2\lambda)\right]\right.$$

After picking scale factors, storing the quantities in addresses H_i ($1 \leq i \leq 16$), and reducing the numberof formulas, we ultimately obtain

$$H'' = \frac{H_8}{H_{13}} - H_{11}; \qquad H' = \frac{H_1}{H_{13}H_8} + H_{11}:$$

$$I_0 = \exp\{\ln H_3 + \ln H_4 + H_{13}H''\ln(H_1 + H_{13}H_2) + \ln[H_{16}H'H'H' - H_{15}H'H'H_6 - H_{16}H']\}\exp\{\ln[H_{13}H'' + H_1 + H_{13}H_2]\}.$$

We use the Lukasiewicz representation for these formulas from left to right:

$$H_{11}H_8H_5 : - H''H_{11}H_8H_{13} \times H_1 : + H'H_3\ln H_4 \ln + + H''H_{13} \times H_2H_{13} \times H_1 + \ln \times + H'H_{16} \times H_6H' \times H' \times H_{15} \times - - H'H' \times H' \times H_{16} \times - \ln + H''H_{13}H_2H_{13} \times H_1 + + \ln + eH'.$$

We note that the formulas are coded in the sequence in which they must be programmed (H_1, ..., H_{16}, respectively, 0001, ..., 0016):

```
K+10 00 0000 0000 0000          3 12 0001 0177 0177
   1 01 0011 0001 0002          4 01 0177 0011 0020
   2 01 0005 0003 0012          5 01 0003 0000 0002
   3 03 0002 0021 0021          6 30 3026 0057 3116
   4 01 0011 0001 0002          7 01 0003 0000 0177
   5 01 0012 0003 0011       0060 01 0004 0000 0002
   6 01 0001 0003 0012          1 30 3026 0062 3116
   7 03 0001 0021 0020          2 01 0003 0000 0176
K+20 01 0003 0023 3116          3 01 0176 0177 0177
   1 01 0004 0023 3116          4 11 0012 0021 0176
   2 03 0001 0001 0021          5 11 0013 0002 0175
   3 01 0012 0003 0011          6 01 0001 0175 0175
   4 01 0002 0001 0013          7 01 0175 0000 0002
   5 03 0011 0001 0001       0070 30 3026 0071 3116
   6 03 0001 0023 3116          1 01 0003 0000 0175
   7 03 0011 0003 0001          2 11 0175 0176 0176
K+30 01 0020 0001 0016          3 01 0176 0177 0177
   1 03 0011 0001 0006          4 11 0016 0020 0176
   2 01 0020 0003 0011          5 11 0020 0006 0175
   3 01 0020 0003 0011          6 11 0020 0175 0175
   4 01 0015 0003 0011          7 11 0015 0175 0175
   5 03 0002 0001 0020       0100 02 0175 0176 0176
   6 01 0020 0003 0011          1 11 0020 0020 0175
   7 01 0020 0003 0011          2 11 0020 0175 0175
K+40 01 0014 0003 0011          3 11 0014 0175 0175
   1 03 0002 0023 3116          4 02 0175 0176 0176
   2 03 0001 0001 0021          5 01 0176 0000 0002
   3 01 0012 0003 0011          6 30 3026 0107 3116
   4 01 0002 0001 0012          7 01 0003 0000 0176
   5 03 0011 0001 0001       0110 01 0176 0177 0177
   6 03 0001 0003 0001          1 11 0012 0021 0176
   7 23 3116 0000 0000          2 11 0012 0002 0175
K+50 03 0001 0000 0000          3 01 0001 0175 0175
K+51 00 0000 0000 0000          4 01 0175 0176 0176
   2 23 3220 0021 0020          5 01 0076 0000 0002
   3 07 0000 0000 0000          6 30 3026 0117 3116
 The following working          7 01 0003 0000 0176
program was obtained:        0120 01 0176 0177 0177
0050 12 0005 0002 0177          1 01 0179 0000 0002
   1 02 0177 0011 0021          2 30 3026 0123 3220
   2 11 0012 0002 0177          3 01 0003 0000 0020
```

Manuscript Page No.	[Footnotes]
163	The PP-AK was developed at the VTs AN UkSSR by L.P. Bystrova under the guidance of Ye.L. Yushchenko [19].
186	The PP-2 was developed by L.N. Ivanenko under the guidance of Ye.L. Yushchenko [8]. N.M. Grishchenko also participated in programming of the arithmetic operator.

APPENDICES

Appendix 1

CONTROL CONSOLE (PU)

In this appendix, we describe only those control and signal elements with which it is necessary to deal in debugging programs and solving problems. Elements involved in checking on the correctness of computer operation are excluded from the description.

The control console (PU) consists of an oblique signal panel (Fig. 3) and a horizontal control panel (Fig. 4).

On the signal panel there are monitoring lights associated with various elements of the control unit (UU), arithmetic unit (AU), and storage unit (ZU). These lights enable us to follow the operations occuring at a given instant in the various machine elements.

The set of signal lights in the top row Sm are connected to the adder and indicate its operation. In reading the content of the adder, we should remember that the numbers in the adder are represented in a modified ones complement. Lights Zn2S and Zn1S which represent the sign of the number in the adder serve simultaneously to signal an overflow of the digit format: if the number in the adder is greater than or equal to +1, the lamps indicate 10, while if it is less than or equal to −1, they indicate 01. In the absence of an overflow in these digit positions, 00 appears for a positive number and 11 for a negative number. Light ZnR2 is connected to the sign bit of the second-number register and indicates the sign of the number selected in address II.

On the control console there are three switchable locations (assembled words NK) 3000, 3001, 3002 connected in parallel with the same locations at the PZU cabinet. Each console location is controlled by a switch. When the switch is on (in the upper position) the console location operates; when it is off, the corresponding PZU location is operative. Where all NK are used in a problem, it is necessary to set up the words for the problem on the NK panels, and to set up at the console the instructions for loading, correcting locations, etc. After loading, the NK switch is turned off. If the program makes use of NK that must change in the course of calculations, they should be set up at the PU.

The set of 32 lights labeled KOp transmit signals on the operation being performed at the given instant. All lights of this set are arranged in order of coding of the operations, and are provided with the operation symbols.

The TrVyb light goes on when the computer is exchanging information with external storage devices, namely: punched tape (punched cards), magnetic tape, or magnetic drum.

The address involved in execution of the given cycle of an instruction is transmitted to the A-register (RA) indicator. When this is done, the A-register is set to zero, so that its content is not used in debugging.

The link-register (RV) indicator block characterizes the state of the link register. The RV is set to 0 when the PRV instruction is executed.

The TrPr light (conditional-jump indication flipflop) goes on when an instruction generating the conditional-jump indication is executed if this indication is generated. The Avt light goes on when the computer is running automatically. Light Zn goes on when an overflow occurs.

The RA signal-light blocks and RTs loop register characterize the state of the A and Ts registers at the time of machine operation.

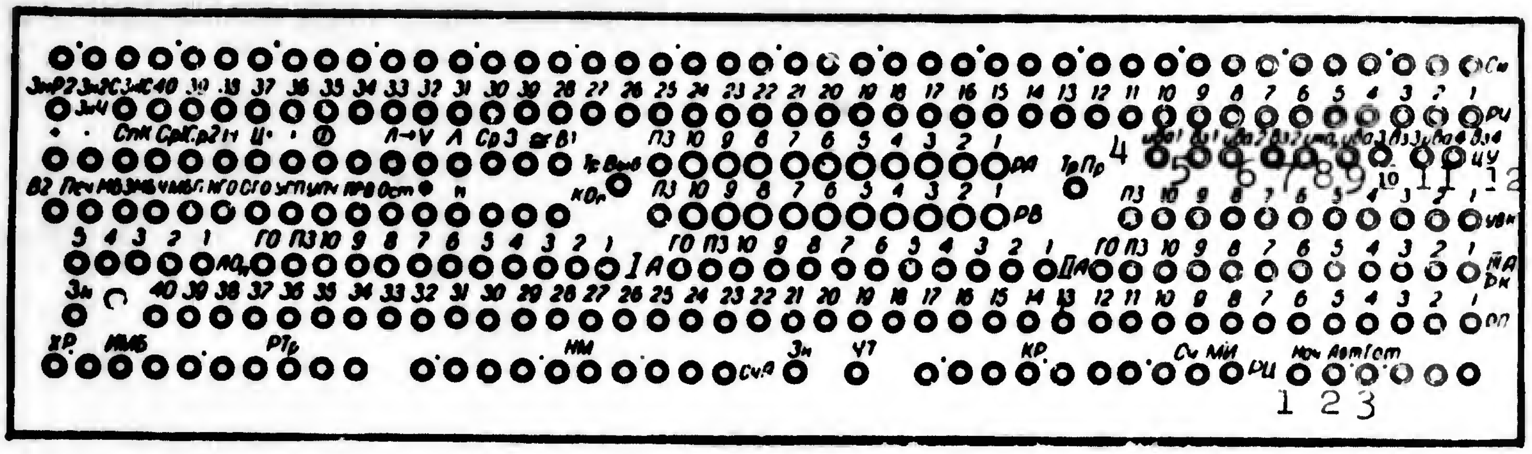


Fig. 3. Signal panel. 1) Nach; 2) Avt; 3) Got; 4) uval; 5) vzl; 6) uva2; 7) vz2; 8) IKO_p; 9) uva3; 10) vz3; 11) uva4; 12) vz4; 13) Ost.

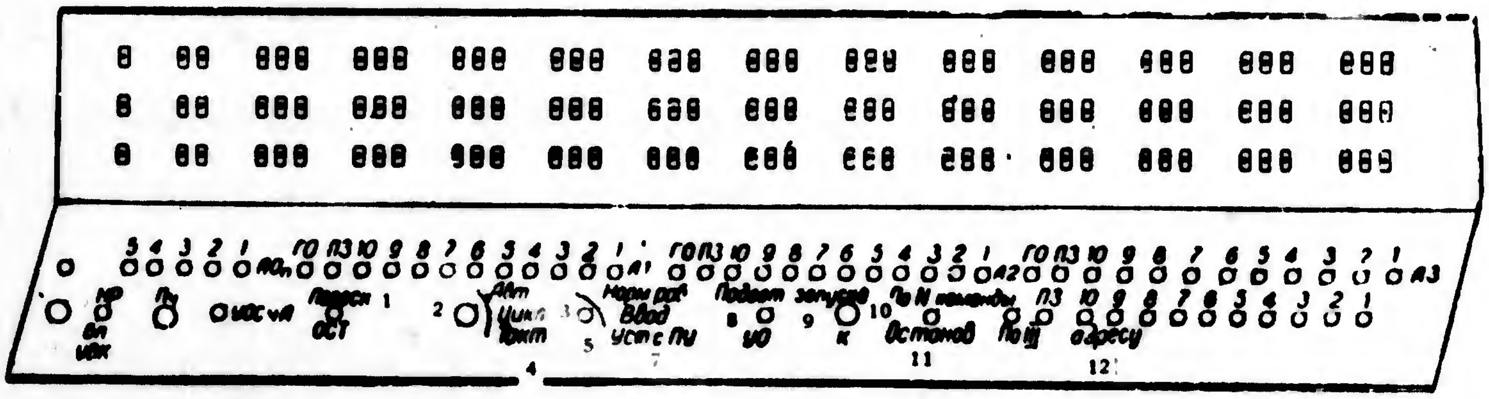


Fig. 4. Control panel. 1) Skip; 2) Avt; 3) cycle; 4) pulse; 5) normal operation; 6) load; 7) set at PU; 8) prepare; 9) start; 10) N instruction; 11) halt; 12) address.

The TsU signal-light block is designed for pulse-by-pulse monitoring of computer operation. Lights ival, iva2, and iva3 signal the transmission to the storage unit of selection pulses for the contents of addresses I, II, and III, respectively, of an instruction, lights vzl and vz2 indicate the end of entry of the contents of the corresponding addresses into the arithmetic unit, light vz3 indicates end of writing of a number into address III, lights iva4 and vz4 indicate transfer of the pulse for selecting the executed instruction and reception of this instruction at the control unit, while light IKOp indicates termination of execution of an arithmetic-unit operation.

The UVK (instruction-transfer device) signal-light block shows the content of instruction counter S — the number of the instruction being executed. When reading the number of an instruction, we must remember that it will change when the arithmetic unit completes execution of the preceding instruction (signal IKOp goes on).

RK signaling indicates the instruction register K has received an instruction to be executed. The instant of arrival of the instruction at the register coincides with the instant of operation of vz4. For reading convenience, the RK indicator is grouped in accordance with the instruction format. At the instant a halt occurs, the RK will contain the executed instruction while UVK will indicate the number of the next instruction and for UPCh, UPP, and OGO instructions the number of the executed instruction, and for V1, V2, Pech, MBZ, and MBCh instructions, a zero. Thus for these last instructions it is not possible to order a halt basedon the instruction number.

Selection and execution of an instruction occurs in the following order:

a) in accordance with the number written into the UVK, an instruction is taken from storage and placed into the instruction register of the control unit. This instruction is written into the RK. The beginning and end of selection is indicated by lights iva4 and vz4;

b) the content of address I of the instruction written into the RK is taken. The address number is written into RA. The beginning and end of selection are marked by lights ival and vzl. The RA is set to zero;

c) process b) is repeated for address II;

d) the operation called for by the instruction selected is executed. Conclusion of execution is marked by lighting of IKOp;

e) the UVK is changed for selection of the next instruction. At the same time, the result of operation execution is transferred to storage. The beginning and end of the transfer are marked by lights iva3 and vz3;

f) the machine proceeds to execute the operations specified in process a).

Lights Zn and Cht indicate the operating mode: write or read. The signals for KR (rotary switch), SchMI (marker-pulse counter), Nach and Got (begin and prepare) monitor code traffic.

The control panel (Fig. 4) has a switch instruction register which may be used to assembly an arbitrary instruction. This instruction register makes it possible to execute the desired instruction from the console.

Button UO (zero set) is designed to set all unit registers to zero when the machine is first started and when various operations are executed from the control console.

The NR-Bl UVK switch is used to block changes in the content of the instruction counter, which permits multiple execution of the same instruction. The UO SchA blocks operation of SchA, RTs and RV. This makes it possible when a program is being debugged to repeat segments within loops and standard routines, since when the UO button is pushed and switch UO SchA is on, the RV and RTs will not be set to zero.

After control has jumped to the desired program segment, the switch is turned off for further operation with the address counter or link register.

Switch Pch connects the readout devices for high-speed printout, slow printout, or automatic reading.

The Avt-Cycle-Pulse switch is designed to switch the machine into different operating modes. The Avt position of this switch corresponds to automatic operation, i.e., automatic execution of the program that has been loaded into the computer; the Cycle position causes execution of the program in cycles, i.e., instruction-by-instruction; the Pulse position causes execution of the program by instruction-cycle components. As we have already mentioned, execution of an instruction occurs in four pulses: selection of instruction, selection of address I, selection of address II, and transfer to address with simultaneous setting of control unit to execute the next instruction. While when a program is executed by cycles, an entire instruction is executed when switch K is operated once, when instructions are executed by pulses, it is necessary to operate the switch four times in accordance with the listed cycle pulse components.

The normal operation – set from PU switch is used to switch the machine so that it will execute the instruction set up at the control console (set from PU position) or will operate normally (normal operation position), i.e., so that it will execute the program written into machine internal storage.

The Halt button is used to stop the computer. When this button is pushed, the machine stops after execution of the instruction contained in the instruction register at the instant the button is pushed.

The halt switch has three positions: neutral, halt on instruction number and halt on address III. Together with the switch halt register, this switch is designed to set up the "halt on address III" and "halt on instruction number" operations.

The instant the machine halts, the RK contains the code for the operation just executed, and Sm the result of this operation; UVK contains the number of the next instruction, while for control-transfer-type operations (UPP, UPCh) it shows the number of the instruction executed.

During a halt operation, the results of the preceding instruction will be left in Sm. At the PU there is a switch that blocks an overflow "with skip," and this permits us, skipping one instruction to continue machine operation when an overflow occurs. When the switch is on, a special button will erase the OZU core array.

<u>Order of Execution of Operations Associated with Program Debugging and Problem Solution</u>

1. Initial program loading:

a) place punched tape (or punched-card deck);

b) set Avt-Cycle-Pulse switch to Avt position, and normal operation — set from PU switch into set from PU position;

c) set up load instruction 20 αβ (for number loading) or 21 αβ (for instruction loading) on instruction switch register; α and β are the initial and final program locations;

d) turn on loading device;

e) push UO button;

f) operate switch K to load program.

The tape should be run through completely. The remainder is a characteristic of the given tape, and it should be stored.

With the method considered, the computer will halt after loading is concluded. With this method, following loading it is necessary to make a special transition to program execution. It is possible to set up the load instruction in assembled storage, however, together with a control-transfer instruction, to transfer control to the address at which the required instruction has been set up and thus proceed to execute the loaded program with no further switching operations.

To check the content of a storage location:

a) set the Avt-cycle-pulse switch into the Pulse position.

b) set the normal operation-set from PU switch into the normal operation position;

c) in address III of the switch instruction register set up the number of the required location;

d) push button UO;

e) operate switch K and read the content of the location called from the Sm signal block at the control console.

3. To execute an instruction written into storage:

a) depending on the mode in which the instruction is to be executed, set the Avt-Cycle-Pulse switch. The Avt position corresponds to automatic execution of the program, the Cycle position to execution of the program instruction-by-instruction, and the Pulse position to execution of the program by instruction-cycle components;

b) perform the operation specified in points b, c, d under the description of initial loading;

c) operate switch K with the Avt-Cycle-Pulse switch in the Avt position, thus starting the computer on automatic operation; in the Cycle position, operation of switch K causes execution of one instruction, and to execute the next instruction we must again operate the switch; in the Pulse position, we must operate K four times to execute a single instruction.

4. To execute an instruction from the control console:

a) set up the required instruction on the control-console instruction switch register;

b) set the Avt-Cycle-Pulse switch to the Cycle position;

c) set the normal operation — set from PU switch into the set from PU position;

d) push UO button;

e) operate switch K to execute the instruction that has been set up on the instruction switch register.

5. To correct the content of a location:

a) set up the required location content in assembly storage;

b) at the control console, set up on the instruction switch register the instruction to transfer the code set up in assembly memory to the required location;

c) perform the operation specified in points b-e under the description of execution of an instruction from the control console.

6. Halt on address III and halt on instruction number:

a) set the halt switch to the position corresponding to the required halt mode;

b) set up the required address or instruction number on the halt location switch register;

c) when the computer is started in the automatic mode, a halt will occur in accordance with the specified regime.

The correctness of the loading of a program or separate segments of a program may be checked either by sequential checking of the segment or by a octal printout of this segment by a readout instruction executed from the console.

When the computer is running in the automatic mode, it may be halted by pushing the Halt button. By operating switch K, it is possible to start the computer to continue the calculation.

Appendix 2

TESTS

Control Unit (UU) Test

Instructions 0001-0006 – check summation of program.
Instructions 0163-0173 – check multiplicaton operation with indication of group operation (GO) in all addresses.
Instructions 0241-0250 – check operation of instruction loading device (UVK) as counter (from 1st to 10th digit).
Instructions 0251-0275 – check operation of UVK for transfer to it of word consisting of all ones and a single zero and a word consisting of all zeros and a single one (from 1st to 10th digit).
Instructions 0276-0277 – check 11th bit of UVK.
Instructions 0300-0303 – check 11th bit of link register (RV).
Instructions 0304-0425 – check A-register and address adder (SmA) using words consisting of all ones and a single zero.
Instructions 0426-0510 – check operation of loop register (RTs).
Instructions 0511-0565 – check transfers to SmA.
Instructions 0566-0617 – check operation of SmA.
Instructions 0620-0641 – check operation of GO indication in control-transfer operations.
Instructions 0643-0662 – check operation of normalization operation with GO indication in all addresses.
Instructions 0700-0706 – check operation of GO indication in control-transfer operations with one present in 11th bit.

```
0001 01 0000 0000 1220
   2 26 0706 0000 0000
   3 07 4001 1220 1220
   4 27 4001 0063 0005
   5 22 1220 1220 0006
   6 33 0000 0000 0000
   7 01 0075 0000 1524
0010 01 0076 0000 1525
   1 01 0077 0000 1526
   2 01 0100 0000 1527
   3 01 0101 0000 1530
   4 01 0102 0000 1247
   5 01 0103 0000 1250
   6 01 0104 0000 1251
   7 01 0105 0000 1252
0020 01 0106 0000 1253
   1 01 3035 0000 0107
   2 01 3035 0000 0110
   3 01 3035 0000 0111
   4 01 3035 0000 0112
   5 01 3035 0000 0113
   6 01 3035 0000 0114
   7 01 3035 0000 0115
0030 01 3035 0000 0116
   1 01 3035 0000 0117
   2 01 3035 0000 0120
   3 01 3035 0000 0121
   4 01 3035 0000 0122
   5 01 3035 0000 0123
   6 01 3035 0000 0124
   7 04 0000 0002 0052
0040 02 0000 0000 0107
   1 05 0002 0000 0052
   2 02 0000 0000 0110
```

```
   3 16 0002 0000 0052
   4 02 0000 0000 0111
   5 31 0000 0046 0052
   6 02 0000 0000 0112
   7 30 0000 0000 0052
0050 02 0000 0000 0113
   1 04 0000 0000 0053
   2 22 0107 0124 0006
   3 02 0000 0000 0114
   4 05 0000 0000 0056
   5 22 0107 0124 0006
   6 02 0000 0000 0115
   7 16 0000 0000 0061
0060 22 0107 0124 0006
   1 02 0000 0000 0116
   2 31 0002 0060 0063
   3 02 0000 0000 0117
   4 30 0002 1527 0066
   5 22 0107 0124 0006
   6 02 0000 0000 0120
   7 32 0000 0000 0000
0070 26 0124 0107 0000
   1 31 4000 0074 0072
   2 27 4001 0071 0073
   3 31 3000 0125 0007
   4 22 0107 0124 0006
   5 33 0000 0000 0000
   6 02 0000 0000 0123
   7 04 0000 0000 1252
0100 02 0000 0000 0121
   1 30 0002 1250 0067
   2 33 0000 0000 0000
   3 02 0000 0000 0122
   4 04 0000 0000 1525
   5 02 0000 0000 0124
   6 04 0000 0000 0070
   7 00 0000 0000 0000
0110 00 0000 0000 0000
   1 00 0000 0000 0000
   2 00 0000 0000 0000
   3 00 0000 0000 0000
   4 00 0000 0000 0000
   5 00 0000 0000 0000
   6 00 0000 0000 0000
   7 00 0000 0000 0000
0120 00 0000 0000 0000
   1 00 0000 0000 0000
   2 00 0000 0000 0000
   3 00 0000 0000 0000
   4 00 0000 0000 0000
   5 01 0207 0000 1525
   6 01 0210 0000 1252
   7 34 0212 0000 0215
0130 16 0215 0207 0132
   1 33 0000 0000 0000
   2 34 0213 0000 0215
   3 16 0215 0210 0135
   4 33 0000 0000 0000
   5 34 0214 0000 0215
   6 16 0215 0211 0140
   7 33 0000 0000 0000

0140 26 1525 1525 0142
   1 33 0000 0000 0000
   2 26 1252 1252 0144
   3 33 0000 0000 0000
   4 01 0000 0000 0216
   5 26 0005 0000 0000
   6 01 4207 0216 0216
   7 27 4001 0146 0150
0150 16 0216 0217 0152
   1 33 0000 0000 0000
   2 26 1777 0525 0000
   3 27 5252 0154 0155
   4 33 0000 0000 0000
   5 26 1525 0252 0000
   6 27 5253 0157 0160
   7 33 0000 0000 0000
0160 26 1000 0777 0000
   1 27 4001 0162 0163
   2 33 0000 0000 0000
   3 26 1775 1253 0000
   4 01 3042 0000 4000
   5 01 3042 0000 4001
   6 10 4000 4001 4002
   7 16 4002 3043 0171
0170 33 0000 0000 0000
   1 01 0000 0000 4000
   2 01 0000 0000 4001
   3 27 4002 0164 0174
   4 31 3000 0240 0125
   5 00 0000 0000 0000
   6 00 0000 0000 0000
   7 00 0000 0000 0000
0200 00 0000 0000 0000
   1 00 0000 0000 0000
   2 00 0000 0000 0000
   3 00 0000 0000 0000
   4 00 0000 0000 0000
   5 00 0000 0000 0000
   6 00 0000 0000 0000
   7 00 0000 0000 0001
0210 00 0000 0000 0002
   1 00 0000 0000 0003
   2 00 0000 1525 0000
   3 00 0000 1252 0000
   4 00 0000 0211 0000
   5 00 0000 0000 0000
```

```
   6 00 0000 0000 0000
   7 00 0000 2777 0006
0220 01 3043 0000 0200
   1 16 3043 0200 0245
   2 01 3044 0000 0200
   3 16 3044 0200 0251
   4 16 0000 0000 0254
   5 16 0000 0000 0256
   6 16 0000 0000 0260
   7 16 0000 0000 0262
0230 16 0000 0000 0264
   1 16 0000 0000 0266
   2 16 0000 0000 0270

0233 16 0000 0000 0272
   4 16 0000 0000 0274
   5 16 0000 0000 0276
   6 00 0000 0000 0025
   7 32 0000 0000 0000
0240 01 0170 0000 0001
   1 01 0220 0000 0777
   2 01 0221 0000 1000
   3 01 0170 0000 1001
   4 04 0000 0000 0777
   5 01 0222 0000 1777
   6 01 0223 0000 0001
   7 01 0170 0000 0002
0250 04 0000 0000 1777
   1 01 0170 0000 1777
   2 01 0224 0000 1776
   3 04 0000 0000 1776
   4 01 0225 0000 1775
   5 04 0000 0000 1775
   6 01 0226 0000 1773
   7 04 0000 0000 1773
0260 01 0227 0000 1767
   1 04 0000 0000 1767
   2 01 0230 0000 1757
   3 04 0000 0000 1757
   4 01 0231 0000 1737
   5 04 0000 0000 1737
   6 01 0232 0000 1677
   7 04 0000 0000 1677
0270 01 0233 0000 1577
   1 04 0000 0000 1577
   2 01 0234 0000 1377
   3 04 0000 0000 1377
   4 01 0235 0000 0777
   5 04 0000 0000 0777
   6 01 0170 0000 1777
   7 03 0236 0224 1776
0300 30 3026 1776 0237
   1 03 0236 0225 1775
   2 30 3026 1775 0237
   3 03 0236 0226 1773
   4 30 3026 1773 0237
   5 03 0236 0227 1767
   6 30 3026 1767 0237
   7 03 0236 0230 1757
0310 30 3026 1757 0237
   1 03 0236 0231 1737
   2 30 3026 1737 0237
   3 03 0236 0232 1677
   4 30 3026 1677 0237
   5 03 0236 0233 1577
   6 30 3026 1577 0237
   7 03 0236 0234 1377
0320 30 3026 1377 0237
   1 03 0236 0235 0777
   2 30 3026 0777 0237
   3 01 0170 0000 1147
   4 30 3026 0325 3146
   5 01 0170 0000 1043
```

```
0326 01 0330 0000 0001
   7 30 3026 3043 0237
0330 04 0000 0000 0331
   1 01 0000 0000 1777
   2 01 3042 0000 1776
   3 26 0000 1776 0000
   4 16 3042 4000 0336
   5 33 0000 0000 0000
   6 16 4000 3042 0340
   7 33 0000 0000 0000
0340 01 3042 0000 1775
   1 26 0000 1775 0000
   2 16 3042 4000 0344
   3 33 0000 0000 0000
   4 16 4000 3042 0346
   5 33 0000 0000 0000
   6 01 3042 0000 1773
   7 26 0000 1773 0000
0350 16 3042 4000 0352
   1 33 0000 0000 0000
   2 16 4000 3042 0354
   3 33 0000 0000 0000
   4 01 3042 0000 1767
   5 26 0000 1767 0000
   6 16 3042 4000 0360
   7 33 0000 0000 0000
0360 16 4000 3042 0362
   1 33 0000 0000 0000
   2 01 3042 0000 1757
   3 26 0000 1757 0000
   4 16 3042 4000 0366
   5 33 0000 0000 0000
   6 16 4000 3042 0370
   7 33 0000 0000 0000
0370 01 3042 0000 1737
   1 26 0000 1737 0000
   2 16 3042 4000 0374
   3 33 0000 0000 0000
   4 16 4000 3042 0376
   5 33 0000 0000 0000
   6 01 3042 0000 1677
   7 26 0000 1677 0000
0400 16 3042 4000 0402
   1 33 0000 0000 0000
   2 16 4000 3042 0404
   3 33 0000 0000 0000
   4 01 3042 0000 1577
   5 26 0000 1577 0000
   6 16 3042 4000 0410
   7 33 0000 0000 0000
0410 16 4000 3042 0412
   1 33 0000 0000 0000
   2 01 3042 0000 1377
   3 26 0000 1377 0000
   4 16 3042 4000 0416
   5 33 0000 0000 0000
   6 16 4000 3042 0420
   7 33 0000 0000 0000
0420 01 3042 0000 0777
0421 26 0000 0777 0000
   2 16 3042 4000 0424
   3 33 0000 0000 0000
   4 16 4000 3042 0426
   5 33 0000 0000 0000
   6 01 0000 0000 1777
   7 01 3035 0000 1776
0430 26 1776 1770 0000
   1 27 4001 0431 0432
   2 16 4000 3035 0434
   3 33 0000 0000 0000
   4 01 3035 0000 1775
   5 26 1775 1770 0000
   6 27 4001 0436 0437
   7 16 3035 4000 0441
```

```
0440 33 0000 0000 0000
   1 01 3035 0000 1773
   2 26 1773 1770 0000
   3 27 4001 0443 0444
   4 16 3035 4000 0446
   5 33 0000 0000 0000
   6 01 3035 0000 1767
   7 26 1767 1763 0000
0450 27 4001 0455 0451
   1 16 4000 3035 0453
   2 33 0000 0000 0000
   3 01 3035 0000 1757
   4 26 1757 1753 0000
   5 27 4001 0455 0456
   6 16 4000 3035 0460
   7 33 0000 0000 0000
0460 01 3035 0000 1737
   1 26 1737 1733 0000
   2 27 4001 0462 0463
   3 16 4000 3035 0465
   4 33 0000 0000 0000
   5 01 3035 0000 1677
   6 26 1677 1673 0000
   7 27 4001 0467 0470
0470 16 4000 3035 0472
   1 33 0000 0000 0000
   2 01 3035 0000 1577
   3 26 1577 1573 0000
   4 27 4001 0474 0475
   5 16 4000 3035 0477
   6 33 0000 0000 0000
   7 01 3035 0000 1377
0500 26 1377 1374 0000
   1 27 4001 0501 0502
   2 16 4000 3035 0504
   3 33 0000 0000 0000
   4 01 3035 0000 0777
   5 26 0777 0770 0000
   6 27 4001 0506 0507
   7 16 4000 3035 0511
0510 33 0000 0000 0000
   1 26 0000 0001 0000
   2 16 0002 4001 0514
   3 33 0000 0000 0000

0514 16 0004 4003 0516
   5 33 0000 0000 0000
   6 16 0010 4007 0520
   7 33 0000 0000 0000
0520 16 0020 4017 0522
   1 33 0000 0000 0000
   2 16 0040 4037 0524
   3 33 0000 0000 0000
   4 16 0100 4077 0526
   5 33 0000 0000 0000
   6 16 0200 4177 0530
   7 33 0000 0000 0000
0530 16 0400 4377 0532
   1 33 0000 0000 0000
   2 01 3043 0000 1000
   3 16 3043 4777 0535
   4 33 0000 0000 0000
   5 26 0000 0003 0000
   6 16 4001 0004 0540
   7 33 0000 0000 0000
0540 26 0000 0007 0000
   1 16 4001 0010 0543
   2 33 0000 0000 0000
   3 26 0000 0017 0000
   4 16 4001 0020 0546
   5 33 0000 0000 0000
   6 26 0000 0037 0000
   7 16 4001 0040 0551
0550 33 0000 0000 0000
   1 26 0000 0077 0000
   2 16 4001 0100 0554
   3 33 0000 0000 0000
   4 26 0000 0177 0000
   5 16 4001 0200 0557
   6 33 0000 0000 0000
   7 26 0000 0377 0000
0560 16 4001 0400 0562
   1 33 0000 0000 0000
   2 26 0000 0777 0000
   3 16 4001 1000 0565
   4 33 0000 0000 0000
   5 31 3000 0566 0240
   6 26 0000 0003 0000
   7 16 4003 0006 0571
0570 33 0000 0000 0000
   1 26 0000 0007 0000
   2 16 0016 4007 0574
   3 33 0000 0000 0000
   4 26 0000 0017 0000
   5 16 4017 0036 0577
   6 33 0000 0000 0000
   7 26 0000 0037 0000
0600 16 0076 4037 0602
   1 33 0000 0000 0000
   2 26 0000 0077 0000
   3 16 4077 0176 0605
   4 33 0000 0000 0000
   5 26 0000 0177 0000
   6 16 0376 4177 0610

0607 33 0000 0000 0000
0610 01 3035 0000 0776
   1 26 0000 0377 0000
   2 16 4377 3035 0614
   3 33 0000 0000 0000
   4 01 3035 0000 1776
   5 26 0000 0777 0000
   6 16 3035 4777 0620
   7 33 0000 0000 0000
0620 26 0000 1000 0000
   1 01 0664 0000 1623
   2 04 0000 0000 4623
   3 33 0000 0000 0000
   4 01 0665 0000 1626
   5 05 0000 0000 4626
   6 33 0000 0000 0000
   7 01 0666 0000 1631
0630 16 0000 0000 4631
   1 33 0000 0000 0000
```

```
   2 01 0667 0000 1634
   3 31 3036 0634 4634
   4 33 0000 0000 0000
   5 01 0670 0000 1637
   6 31 3035 4637 0637
   7 33 0000 0000 0000
0640 01 3146 0000 1642
   1 30 3026 0643 4642
   2 33 0000 0000 0000
   3 35 0671 1777 4000
   4 16 1777 0672 0646
   5 33 0000 0000 0000
   6 16 1000 0673 0650
   7 33 0000 0000 0000
0650 35 0674 4000 1777
   1 16 1000 0675 0653
   2 33 0000 0000 0000
   3 16 1777 0676 0655
   4 33 0000 0000 0000
   5 01 0677 0000 1000
   6 35 4000 1776 1777
   7 16 1776 3026 0661
0660 33 0000 0000 0000
   1 16 1777 0677 0663
   2 33 0000 0000 0000
   3 04 0000 0000 0700
   4 04 0000 0000 0624
   5 04 0000 0000 0627
   6 04 0000 0000 0632
   7 04 0000 0000 0635
0670 04 0000 0000 0640
   1 05 2525 2525 2525
   2 20 0000 0000 0001
   3 12 5252 5252 5252
   4 21 2525 2525 2525
   5 20 0000 0000 0003
   6 32 5252 5252 5250
   7 12 5252 5252 5252
0700 26 0000 0140 0000
   1 30 3026 0702 7006
0702 26 0000 0137 0000
   3 30 3026 0706 0704
   4 31 0670 7007 0705
   5 33 0000 0000 0000
   6 31 3000 0007 0566
HK:
3006 33 0000 0000 0000
3007 33 0000 0000 0000
```

<u>Storage Unit (ZU) Tests</u>

Addresses	0001-0013	— constants.
Instructions	0014-0021	— check summation of test.

<u>First OZU Test</u>

Instructions	0022-0027	— take word for writing.
Instructions	0030-0032	— write code into OZU.
Instructions	0034-0037	— read code and check.
Instructions	0040-0042	— number-of-readings counter.
Instruction	0043	— proceed to write next constant.

<u>Second OZU Test ("tickling" and partial selection)</u>

Instructions	0046-0047	— select constants for writing into OZU core array and into diagonal locations.
Instructions	0050-0052	— write into array.

Instructions 0053-0061 — write into diagonal locations with multiple readout ("tickling").

Instructions 0062-0070 — check nondiagonal locations.

Instructions 0072-0073 — change writing constants.

<u>Third OZU Test</u>

Instructions 0100-0103 — write word 1010 ... into OZU.

Instructions 0104-0110 — read for <u>x</u>.

<u>Fourth OZU Test</u>

Instructions 0122-0136 — write "traveling" one or zero.

Instructions 0140-0155 — read from OZU with check.

```
0001 37 7777 7777 7777
   2 25 2525 2525 2525
   3 12 5252 5252 5252
   4 01 0001 0001 0001
   5 33 0000 0000 0777
   6 01 0004 0000 0020
   7 01 0000 0000 0020
0010 00 0041 0041 0000
   1 26 1777 1737 0000
   2 00 0040 0040 0000
   3 26 2000 1740 0000
   4 01 0000 0000 0210

0015 26 0166 0000 0000
   6 07 4001 0210 0210
   7 27 4001 0016 0020
0020 22 0210 0210 0021
   1 33 0000 0000 0000
   2 31 3001 0025 0023
   3 01 3002 0000 0020
   4 05 0000 0000 0030
   5 01 0007 0001 0026
   6 01 0000 0000 0020
   7 03 3010 0026 0026
0030 26 1777 0177 0000

0031 01 0020 0000 4001
   2 27 4001 0031 0033
   3 03 3011 3200 0036
   4 26 0000 0200 0000
   5 16 4000 0020 0037
   6 33 0000 1000 0000
   7 27 4001 0035 0040
0040 03 3012 0036 0036
   1 05 0036 0165 0034
   2 31 3001 0043 0044
   3 05 0026 0006 0026
   4 31 3000 0045 0022
   5 01 0000 0000 0014
   6 01 0000 0000 0020
   7 01 3036 0000 0021
0050 26 1777 0200 0000
   1 01 0020 0000 4000
   2 27 4001 0051 0053
   3 26 1573 0000 0000
   4 01 0021 0000 4204
   5 03 0000 3200 0015
   6 07 4204 4204 0000
   7 03 3012 0015 0015
0060 05 0015 0005 0056
```

```
   1 27 4041 0054 0062
   2 01 0077 0000 0063
   3 26 0245 0205 0000
   4 16 4000 0020 0066
   5 33 0000 0002 0000
   6 27 4001 0064 0067
   7 03 0010 0063 0063
0070 05 0063 0011 0063
   1 31 0014 0072 0076
   2 02 0000 0000 0014
   3 01 0000 0000 0021
   4 01 3036 0000 0020
   5 05 0000 0000 0050
   6 31 3000 0100 0045
   7 26 0245 0205 0000
0100 01 0116 0000 0105
   1 26 1600 0000 0000
   2 01 0002 0000 4200
   3 27 4001 0102 0104
   4 03 3011 0065 0107
   5 26 0240 0200 0000
   6 16 4000 0002 0110
   7 33 0000 0000 0000

0110 27 4001 0106 0111
   1 03 3012 0107 0107
   2 05 0107 0166 0105
   3 03 0012 0105 0105
   4 05 0105 0013 0104
   5 31 3000 0117 0100
   6 26 0240 0200 0000
   7 01 0000 0000 0014
0120 01 3012 0000 0017
   1 01 0017 0000 0021
   2 26 0000 0200 0000
   3 01 0021 0000 0020
   4 31 00[illegible] 0126 0125
   5 01 3036 0020 0020
   6 01 0020 0000 4000
   7 16 0021 3042 0133
0130 16 0021 3026 0135
   1 13 3012 0021 0021
   2 05 0000 0000 0136
   3 01 3026 0000 0021
   4 05 0000 0000 0136
   5 01 3012 0000 0021
   6 27 4001 0123 0137
   7 01 0017 0000 0021
0140 26 0000 0200 0000
```

```
   1 01 0021 0000 0020
   2 31 0014 0144 0143
   3 01 3036 0020 0020
   4 16 4000 0020 0146
   5 33 0000 0004 0000
   6 16 0021 3042 0152
   7 16 0021 3026 0154
0150 13 3012 0021 0021
   1 05 0000 0000 0155
   2 01 3026 0000 0021
   3 05 0000 0000 0155
```

```
   4 01 3012 0000 0021
   5 27 4001 0141 0156
   6 13 3012 0017 0017
   7 16 0017 3042 0161
0160 05 0000 0000 0121
   1 31 0014 0162 0164
   2 02 0000 0000 0014
   3 05 0000 0000 0120
   4 31 3000 0022 0117
   5 33 0000 0001 0077
   6 33 0000 0003 0177
```

Arithmetic Unit (AU) Tests

Examples for AU check

Binary codes	instructions
(0) ∨ (1) = (1)	0043—0045
(01) ∨ (0) = (01)	0046—0053
(1) ∨ (1) = (1)	0054—0056
(1) ∧ (0) = (0)	0057—0061
(0) ∧ (0) = (0)	0062—0064
(0) ∧ (10) = (0)	0065—0067
(1) ∧ (1) = (1)	0070—0072
(01) ≃ (10) = (0)	0073—0075
(10) ≃ (01) = (0)	0076—0100
(01) ≃ (01) = (1)	0101—0103
(10) ≃ (10) = (1)	0104—0107
octal codes	
05 (25) + 05 (25) = 12 (52)	0110—0112
(25) + 12 (52) = 05 (25)	0113—0115
07 5736 7573 6745 + 07 5736 7573 6745 = = 17 3675 7367 5712	0116—0120
05 (25) + 32 (52) = (25)	0121—0123
12 (52) + (25) = 05 (25)	0124—0126
00 0100 0400 2001 + 30 0040 0200 1000 = = 27 7737 7577 6777	0127—0131
20 2001 0004 0020 + 00 4002 0010 0040 = = 00 2001 0004 0020	0132—0134
1 (0) + (0) 1 = (0) 1	0135—0142
1 (0) + (0) 20 = 0 (20)	0143—0145
1 (0) + 00 0100 0400 2001 = = 00 0100 0400 2001	0146—0150
17 3675 7367 5712 + 17 3675 7367 5712 = = 16 7573 6757 3624	0151—0154
12 (52) — 12 (52) = 1 (0)	0155—0157
(25) — (25) = 1 (0)	0160—0162
12 (52) *СлК* (25) = 3 (7)	0163—0165
(0) *СлК* 02 (52) = 02 (52)	0166—0170
2 (0) 1 × 3 (0) = (0) 1	0171—0173
3 (0) 1 × 0 (7) = 3 (0)	0174—0176
0 (7) × (07) = 1 (7) 5	0177—0201
05 (25) × 32 (52) = 23 (43)	0202—0204
01 (0) : 03 (0) = 05 (25)	0205—0207
200020 (0) : 000 (16) 0 = (2) 00	0210—0212

Instructions 0214-0226 — formation of pseudorandom number.
Instructions 0240-0245 — iterative division.
Instructions 0256-0345 — constants.
( ) — represents number in period.

```
0001 25 5564 3455 4124
   2 12 5252 5252 5252
   3 25 2525 2525 2525
   4 05 2525 2525 2525
   5 32 5252 5252 5252
   6 07 5736 7573 6745
   7 17 3675 7367 5712
0010 00 0100 0400 2001
   1 30 0040 0200 1000
   2 20 2001 0004 0020
0013 00 4002 0010 0040
   4 00 2001 0004 0020
   5 00 0000 0000 0001
   6 00 0000 0000 0040
   7 27 7737 7577 6777
0020 16 7573 6757 3624
   1 20 0000 0000 0001
   2 30 0000 0000 0000
   3 30 0000 0000 0001
   4 17 7777 7777 7775
   5 23 4343 4343 4343
   6 01 0000 0000 0000
   7 03 0000 0000 0000
0030 20 0020 0000 0000
   1 00 0160 0000 0000
   2 22 2222 2222 2200
   3 27 7777 7777 7777
   4 01 0800 0000 0367
   5 26 0346 0000 0000
   6 07 4002 0367 0367
   7 27 4001 0036 0040
0040 16 0367 0001 0351
   1 22 0367 0367 0256
   2 01 3066 0000 0353
   3 14 0000 3036 0347
   4 16 0347 3036 0046
   5 33 0000 0000 0000
   6 14 0002 0000 0347
   7 16 0347 0002 0051
0050 33 0000 0000 0000
   1 14 0003 0000 0347
   2 16 0347 0003 0054
   3 33 0000 0000 0000
   4 14 3036 3036 0347
   5 16 0347 3036 0057
   6 33 0000 0000 0000
   7 15 3036 0000 0347
0060 16 0347 0000 0062
   1 33 0000 0000 0000
   2 15 0000 0000 0347
   3 16 0347 0000 0065
   4 33 0000 0000 0000
   5 15 0000 0003 0347
   6 16 0347 0000 0070
   7 33 0000 0000 0000
0070 15 3036 3036 0347
   1 16 0347 3036 0073
   2 33 0000 0000 0000
   3 17 0002 0003 0347
   4 16 0347 0000 0076
   5 33 0000 0000 0000
   6 17 0003 0002 0347
   7 16 0347 0000 0101
0100 33 0000 0000 0000
   1 17 0002 0002 0347
   2 16 0347 3036 0107
   3 33 0000 0000 0000
   4 17 0003 0003 0347
   5 16 0347 3036 0107
   6 33 0000 0000 0000
   7 31 3000 0043 0110
0110 01 0004 0004 0347
   1 16 0347 0002 0113
   2 33 0000 0000 0000
   3 01 0003 0002 0347
   4 16 0347 0004 0116
   5 33 0000 0000 0000
   6 01 0006 0006 0347
   7 16 0347 0007 0121
```

```
0120 33 0000 0000 0000
   1 01 0004 0005 0347
   2 16 0347 0003 0124
   3 33 0000 0000 0000
   4 01 0002 0003 0347
   5 16 0347 0004 0127
   6 33 0000 0000 0000
   7 01 0010 0011 0347
0130 16 0347 0017 0132
   1 33 0000 0000 0000
   2 01 0012 0013 0347
   3 16 0347 0014 0135
   4 33 0000 0000 0000
   5 01 3026 0015 0347
   6 16 0347 0015 0140
   7 33 0000 0000 0000
0140 01 3026 0016 0347
   1 16 0347 0016 0143
   2 33 0000 0000 0000
   3 01 3026 0010 0347
   4 16 0347 0010 0146
   5 33 0000 0000 0000
   6 01 3026 0013 0347
   7 16 0347 0013 0151
0150 33 0000 0000 0000
   1 07 0007 0007 0347
   2 16 0347 0020 0154
   3 33 0000 0000 0000
   4 31 3000 0110 0155
   5 02 0002 0002 0347
   6 16 0347 3026 0160
   7 33 0000 0000 0000
0160 06 0003 0003 0347
   1 16 0347 3026 0163
   2 33 0000 0000 0000
   3 03 0002 0003 0347
   4 16 0347 3036 0166
   5 33 0000 0000 0000
   6 13 0000 0261 0347
   7 16 0347 0261 0171
0170 33 0000 0000 0000
   1 11 0021 0022 0347
   2 16 0347 0015 0174
   3 33 0000 0000 0000
   4 11 0023 3035 0347
   5 16 0347 0022 0177
   6 33 0000 0000 0000
   7 11 3035 3035 0347
0200 16 0347 0024 0202
   1 33 0000 0000 0000
   2 11 0004 0005 0347
   3 16 0347 0025 0205
   4 33 0000 0000 0000
   5 12 0026 0027 0347
   6 16 0347 0004 0210
   7 33 0000 0000 0000
0210 12 0030 0031 0347
   1 16 0347 0032 0213
   2 33 0000 0000 0000
   3 05 0000 0000 0263
   4 01 0353 0000 0354
   5 10 0354 3042 0353
   6 31 0353 0221 0217
   7 02 0000 0353 0353
0220 01 0353 3042 0353
   1 15 0354 0260 0352
   2 12 0352 3044 0352
   3 15 0354 3044 0351
   4 05 0000 0351 0226
   5 02 0000 0352 0352
   6 17 0352 0353 0353
   7 11 0353 0354 0355
0230 11 0354 0353 0356
   1 16 0355 0356 0234
   2 22 0353 0356 0233
   3 33 0000 0000 0000
   4 06 0353 0000 0355
   5 05 0354 0355 0240
   6 06 0354 0000 0355
   7 01 0353 0000 0354
```

```
0240 01 0354 0000 0356
   1 02 3035 0355 0357
   2 11 0356 0357 0352
   3 01 0356 0352 0356
   4 11 0357 0357 0357
   5 05 0357 0000 0247
   6 05 0000 0000 0242
   7 12 0354 0355 0357
0250 02 0357 0356 0352
   1 05 0352 3002 0254
   2 22 0354 0357 0253
   3 33 0000 0000 0012
   4 31 3000 0214 0043
   5 31 3001 0214 0043
   6 33 0000 0000 0000
   7 05 0000 0000 0042
0260 01 7777 7777 7777
   1 02 5252 5252 5252
   2 05 2525 2525 2524
   3 35 0002 0350 0347
   4 16 0347 0002 0266
   5 33 0000 0000 0000
   6 16 0350 3026 0270
   7 33 0000 0000 0000
0270 35 0000 0350 0347
   1 16 0347 0000 0273
   2 33 0000 0000 0000
   3 16 0350 3026 0275
   4 33 0000 0000 0000
   5 35 3026 0350 0347
   6 16 0347 3026 0300
   7 33 0000 0000 0000
0300 16 0350 3026 0302
   1 33 0000 0000 0000
   2 35 3177 0350 0347
   3 16 0347 0335 0305
   4 33 0000 0000 0000
   5 16 0350 0336 0307
   6 33 0000 0000 0000
   7 13 0337 0000 0347
0310 16 0347 0000 0312
   1 33 0000 0000 0000
   2 13 0337 3026 0347
   3 16 0347 0000 0315
   4 33 0000 0000 0000
   5 13 0340 3026 0347
   6 16 0347 0343 0320
   7 33 0000 0000 0000
0320 13 0341 3177 0347
   1 16 0347 3026 0323
   2 33 0000 0000 0000
   3 13 0342 3177 0347
   4 16 0347 3012 0326
   5 33 0000 0000 0000
   6 13 0344 0003 0347
   7 16 0347 3026 0331
0330 33 0000 0000 0000
   1 13 0345 0003 0347
   2 16 0347 0000 0334
   3 33 0000 0000 0000
   4 05 0000 0000 0360
   5 30 0000 0000 0000
   6 20 0000 0000 0047
   7 00 0000 0000 0014
0340 20 0000 0000 0014
   1 00 0000 0000 0050
   2 20 0000 0000 0050
   3 00 0020 0000 0000
   4 20 0000 0000 0100
   5 20 0000 0000 0101
```

Printout Test

Instructions 0001-0006 — check summation of test.

Instructions 0007-0013 — octal printout.

Instructions 0014-0017 — decimal printout.

Following octal printout, a halt occurs; the printout switch is then set to decimal printout.

```
0001    26 0654 0001  —
   2    07 4000 0654 0654
   3    27 4001 0002 0004
   4    16 0654 0655 0007
   5    22 0654 0654 0006
   6    33  —    —    —
   7    22 0020 0307 0010
0010    31 3000 0007 0011
   1    22 0310 0343 0012
   2    31 3000 0013 0011
   3    33  —    —    —
   4    22 0344 0627 0015
   5    31 3000 0014 0016
   6    22 0630 0653 0017
   7    31 3000 0006 0016
0020    01  —    —    —
   1    — 1000   —    —
   2    — 0100   —    —
   3    — 0010   —    —
   4    — 0001   —    —
   5    — 0000 1000   —
   6    —  —   0100   —
   7    —  —   0010   —
0030    —  —   0001   —
   1    —  —    —   1000
   2    —  —    —   0100
   3    —  —    —   0010
   4    —  —    —   0001
   5    31 —    —    —
   6    30 1000  —    —
   7    20 0100  —    —
0040    20 0010  —    —
   1    20 0001  —    —
   2    20 —   1000   —
   3    20 —   0100   —
   4    20 —   0010   —
   5    20 —   0001   —
   6    20 —    —   1000
   7    20 —    —   0100
0050    20 —    —   0010
   1    20 —    —   0001
   2    02 —    —    —
```

3	00	2000	—	—
4	—	0200	—	—
5	—	0020	—	—
6	—	0002	—	—
7	—	—	2000	—
0060	—	—	0200	—
1	—	—	0020	—
2	—	—	0002	—
3	—	—	—	2000
4	—	—	—	0200
5	—	—	—	0020
6	—	—	—	0002
7	32	—	—	—
0070	30	2000	—	—
1	20	0200	—	—
2	20	0020	—	—
3	20	0002	—	—
0074	20	—	2000	—
5	20	—	0200	—
6	20	—	0020	—
7	20	—	0002	—
0100	20	—	—	2000
1	20	—	—	0200
2	20	—	—	0020
3	20	—	—	0002
4	03	—	—	—
5	—	3000	—	—
6	—	0300	—	—
7	—	0030	—	—
0110	—	0003	—	—
1	—	—	3000	—
2	—	—	0300	—
3	—	—	0030	—
4	—	—	0003	—
5	—	—	—	3000
6	—	—	—	0300
7	—	—	—	0030
0120	—	—	—	0003
1	33	—	—	—
2	30	3000	—	—
3	20	0300	—	—
4	20	0030	—	—
5	20	0003	—	—
6	20	—	3000	—
7	20	—	0300	—
0130	20	—	0030	—
1	20	—	0003	—
2	20	—	—	3000
3	20	—	—	0300
4	20	—	—	0030
5	20	—	—	0003
6	04	—	—	—
7	—	4000	—	—
0140	—	0400	—	—
1	—	0040	—	—
2	—	0004	—	—
3	—	—	4000	—
4	—	—	0400	—
5	—	—	0040	—
6	—	—	0004	—
7	—	—	—	4000
0150	—	—	—	0400
1	—	—	—	0040
2	—	—	—	0004
3	34	—	—	—
4	30	4000	—	—
5	20	0400	—	—
6	20	0040	—	—
7	—	—	—	—
0160	—	—	—	—
1	20	0004	—	—
2	20	—	4000	—
3	20	—	0400	—
4	20	—	0040	—
5	20	—	0004	—
6	20	—	—	4000

0167	20	—	—	0400
0170	20	—	—	0040
1	20	—	—	0004
2	05	—	—	—
3	—	5000	—	—
4	—	0500	—	—
5	—	0050	—	—
6	—	0005	—	—
7	—	—	5000	—
0200	—	—	0500	—
1	—	—	0050	—
2	—	—	0005	—
3	—	—	—	5000
4	—	—	—	0500
5	—	—	—	0050
6	—	—	—	0005
7	35	—	—	—
0210	30	5000	—	—
1	20	0500	—	—
2	20	0050	—	—
3	20	0005	—	—
4	20	—	5000	—
5	20	—	0500	—
6	20	—	0050	—
7	20	—	0005	—
0220	20	—	—	5000
1	20	—	—	0500
2	20	—	—	0050
3	20	—	—	0005
4	06	—	—	—
5	—	6000	—	—
6	—	0600	—	—
7	—	0060	—	—
0230	—	0006	—	—
1	—	—	6000	—
2	—	—	0600	—
3	—	—	0060	—
4	—	—	0006	—
5	—	—	—	6000
6	—	—	—	0600
7	—	—	—	0060
0240	—	—	—	0006
1	36	—	—	—
2	30	6000	—	—
3	20	0600	—	—
4	20	0060	—	—
5	20	0006	—	—
6	20	0000	6000	—
7	20		0600	—
0250	20	—	0060	—
1	20	—	0006	—
2	20	—	—	6000
3	20	—	—	0600
4	20	—	—	0060
5	20	—	—	0006
6	07	—	—	—
7	—	7000	—	—
0260	—	0700	—	—
1	—	0070	—	—
0262	—	0007	—	—
3	—	—	7000	—
4	—	—	0700	—
5	—	—	0070	—
6	—	—	0007	—
7	—	—	—	7000
0270	—	—	—	0700
1	—	—	—	0070
2	—	—	—	0007
3	37	—	—	—
4	30	7000	—	—
5	20	0700	—	—
6	20	0070	—	—
7	20	0007	—	—
0300	20	—	7000	—
1	20	—	0700	—

2		20	—	0070	—
3		20	—	0007	—
4		20	—	—	7000
5		20	—	—	0700
6		20	—	—	0070
7		20	—	—	0007
0310		00	1234	5670	1234
1		04	0123	4567	0123
2		03	4012	3456	7012
3		02	3401	2345	6701
4		01	2340	1234	5670
5		00	1234	0123	4567
6		07	0123	4012	3456
7		06	7012	3401	2345
0320		05	6701	2340	1234
1		04	5670	1234	0123
2		03	4567	0123	4012
3		02	3456	7012	3401
4		01	2345	6701	2345
5		—	—	—	—
6		20	1234	5670	1234
7		34	0123	4567	0123
0330		23	4012	3456	7012
1		32	3401	2345	6701
2		21	2340	1234	5670
3		30	1234	0123	4567
4		27	0123	4012	3456
5		36	7012	3401	2345
6		25	6701	2340	1234
7		34	5670	1234	0123
0340		23	4567	0123	4012
1		32	3456	7012	3401
2		21	2345	6701	2340
3		37	7777	7777	7777
4	+	10	—	—	—
5	+	01	—	—	—
6	+	—	1000	—	
7	+	—	0100	—	
0350	+	—	0010	—	
1	+	—	0001	—	
2	+	—	—	1000	
3	+	—	—	0100	
4	+	—	—	0010	
0355	+	—	—	0001	
6	—	10	—	—	
7	—	01	—	—	
0360	—	—	1000	—	
1	—	—	0100	—	
2	—	—	0010	—	
3	—	—	0001	—	
4	—	—	—	1000	
5	—	—	—	0100	
6	—	—	—	0010	
7	—	—	—	0001	
0370	+	20	—	—	
1	+	02	—	—	
2	+	—	2000	—	
3	+	—	0200	—	
4	+	—	0020	—	
5	+	—	0002	—	
6	+	—	—	2000	
7	+	—	—	0200	
0400	+	—	—	0020	
1	+	—	—	0002	
2	—	20	—	—	
3	—	02	—	—	
4	—	—	2000	—	
5	—	—	0200	—	
6	—	—	0020	—	
7	—	—	0002	—	
0410	—	—	—	2000	
1	—	—	—	0200	
2	—	—	—	0020	
3	—	—	—	0002	
4	+	30	—	—	
5	+	03	—	—	
6	+	—	3000	—	
7	+	—	0300	—	
0420	+	—	0030	—	
1	+	—	0003	—	
2	+	—	—	3000	
3	+	—	—	0300	
4	+	—	—	0030	
5	+	—	—	0003	
6	—	30	—	—	
7	—	03	—	—	
0430	—	—	3000	—	
1	—	—	0300	—	
2	—	—	0030	—	
3	—	—	0003	—	
4	—	—	—	3000	
5	—	—	—	0300	
6	—	—	—	0030	
7	—	—	—	0003	
0440	+	40	—	—	
1	+	04	—	—	
2	+	—	4000	—	
3	+	—	0400	—	
4	+	—	0040	—	
5	+	—	0004	—	
6	+	—	—	4000	
7	+	—	—	0400	
0450	+	—	—	0040	
1	+	—	—	0004	
2	—	40	—	—	
3	—	04	—	—	
4	—	—	4000	—	
5	—	—	0400	—	
6	—	—	0040	—	
7	—	—	0004	—	
0460	—	—	—	4000	
1	—	—	—	0400	
2	—	—	—	0040	
3	—	—	—	0004	
4	+	50	—	—	
5	+	05	—	—	
6	+	—	5000	—	
7	+	—	0500	—	
0470	+	—	0050	—	
1	+	—	0005	—	
2	+	—	—	5000	
3	+	—	—	0500	
4	+	—	—	0050	
5	+	—	—	0005	
6	—	50	—	—	
7	—	05	—	—	
0500	—	—	5000	—	
1	—	—	0500	—	
2	—	—	0050	—	
3	—	—	0005	—	
4	—	—	—	5000	
5	—	—	—	0500	
6	—	—	—	0050	
7	—	—	—	0005	
0510	+	60	—	—	
1	+	06	—	—	
2	+	—	6000	—	
3	+	—	0600	—	
4	+	—	0060	—	
5	+	—	0006	—	
6	+	—	—	6000	
7	+	—	—	0600	
0520	+	—	—	0060	
1	+	—	—	0006	
2	—	60	—	—	
3	—	06	—	—	
4	—	—	6000	—	
5	—	—	0600	—	
6	—	—	0060	—	
7	—	—	0006	—	
0530	—	—	—	6000	
1	—	—	—	0600	
2	—	—	—	0060	
3	—	—	—	0006	
4	+	70	—	—	
5	+	07	—	—	
6	+	—	7000	—	

```
   7 + — 0700  —
0540 + — 0070  —
   1 + — 0007  —
   2 + —   —  7000
   3 + —   —  0700
   4 + —   —  0070
   5 + —   —  0007
   6 — 70  —    —
   7 — 07  —    —
0550 — — 7000  —
   1 — — 0700  —
   2 — — 0070  —
   3 — — 0007  —
   4 — —   —  7000
   5 — —   —  0700
   6 — —   —  0070
   7 — —   —  0007
0560 + 80  —    —
   1 + 08  —    —
   2 + — 8000  —
   3 + — 0800  —
   4 + — 0080  —
   5 + — 0008  —
   6 + —   —  8000
   7 + —   —  0800
0570 + —   —  0080
   1 + —   —  0008
   2 — 80  —    —
   3 — 08  —    —
   4 — — 8000  —
   5 — — 0800  —
   6 — — 0080  —
   7 — — 0008  —
0600 — —   —  8000
   1 — —   —  0800
   2 — —   —  0080
   3 — —   —  0008
   4 + 90  —    —
   5 + 09  —    —
```

```
   6 + — 9000  —
   7 + — 0900  —
0610 + — 0090  —
   1 + — 0009  —
   2 + —   —  9000
   3 + —   —  0900
   4 + —   —  0090
   5 + —   —  0009
   6 — 90  —    —
   7 — 09  —    —
0620 — — 9000  —
   1 — — 0900  —
   2 — — 0090  —
   3 — — 0009  —
   4 — —   —  9000
   5 — —   —  0900
   6 — —   —  0090
   7 — —   —  0009
0630 + 01 2345 6789
   1 + 90 1234 5678
   2 + 89 0123 4567
   3 + 78 9012 3456
   4 + 67. 8901 2345
   5 + 56 7890 1234
   6 + 45 6789 0123
   7 + 34 5678 9012
0640 + 23 4567 8901
   1 + 12 3456 7890
   2 — 01 2345 6789
   3.— 90 1234 5678
   4 — 89 0123 4567
   5 — 78 9012 3456
   6 — 67 8901 2345
   7 — 56 7890 1234
0650 — 45 6789 0123
   1 — 34 5678 9012
   2 — 23 4567 8901
   3 — 12 3456 7890
   4 — —   —    —
   5 — —   —    —
```

Magnetic Drum (MB) Test

Instructions 0001-0006 — initialize instructions calling to MB.

Instructions 0007-0010 — MB shift counter.

Instructions 0011-0015 — write codes into OZU.

Instructions 0016-0021 — generate instructions calling to MB.

Instructions 0024-0030 — write onto MB and read.

Instructions 0042-0047 — check summation of test.

```
  Bв 21 0001 0062 0000
0001 01 3200 0000 0032
   2 01 0051 0000 0030
   3 01 0052 0000 0024
   4 01 0053 0000 0025
   5 01 0054 0000 0026
   6 01 0055 0000 0027
   7 01 3011 0000 0056
0010 01 3011 0000 0057
   1 10 3000 3040 0060
   2 34 0057 0000 0000
   3 01 3007 0000 4062
0014 01 3011 0057 0057
   5 05 0057 3000 0012
   6 03 3002 0024 0024
   7 03 3002 0026 0026
0020 03 3000 0025 0025
   1 03 3000 0027 0027
   2 12 3000 3040 0061
   3 03 0061 0030 0030
   4 00 0000 0000 0000
   5 00 0000 0000 0000
   6 00 0000 0000 0000
   7 00 0000 0000 0000
```

```
0030 00 0000 0000 0000
   1 16 4063 5031 0033
   2 33 0000 0000 0000
   3 27 4001 0031 0034
   4 03 3012 0032 0032
   5 03 0060 0024 0024
   6 03 0060 0026 0026
   7 01 3011 0056 0056
0040 05 0056 3001 0024
   1 31 0000 0001 0000
   2 01 0000 0000 0062
   3 26 0062 0000 0000
   4 07 4001 0062 0062
   5 27 4001 0044 0046
0046 22 0062 0062 0047
   7 33 0000 0000 0000
0050 31 0000 0001 0000
   1 26 0000 0000 0000
   2 25 0001 0000 0000
   3 23 0063 0062 0026
   4 25 0000 0000 0000
   5 24 1031 1030 0030
   6 00 0000 0000 0000
   7 00 0000 0000 0000
0060 00 0000 0000 0000
   1 00 0000 0000 0000
   2 00 0000 0000 0000
```

Instruction System for "Kiev" Computer

1 Арифметические и логические операции			
3 КОп	4 Название операции	Возможность модификации адресов 5	6 Аварийный останов
01	Сложение 11	Да 10	|Рез| > 1
02	Вычитание 12	»	|Рез| > 1
03	Сложение команд	13 »	—
06	Вычитание модулей	14 »	—
07	Циклическое сложение	15 »	—
10	Умножение без округления	16 »	—
11	Умножение с округлением	17 »	—
12	Деление 18	»	—
13	Сдвиг логический 19	»	—
14	Логическое сложение 20	»	—
15	Логическое умножение 21	»	—
17	Неравнозначность 22	»	—
35	Нормализация 23	»	—

2 Операции управления						
3 КОп	7 Проверяемое условие	8 В случае выполнения	9 В случае невыполнения	'А	'Ц	'Р
04	$'('A_1 + 'E_1'A) < '('A_2 + 'E_2'A)$	ПУ $'A_2 + 'E_2'A$	$'C+1 \Rightarrow C$	Не изменяется 24	Не изменяется 24	Не изменяется 24
05	$\|'('A_1 + 'E_1'A)\| < \|'('A_2 + 'E_2'A)\|$	ПУ $'A_2 + 'E_2'A$	$'C+1 \Rightarrow C$	То же 25	То же 25	То же 25
16	$'('A_1 + 'E_1'A) = '('A_2 + 'E_2'A)$	ПУ $'A_2 + 'E_2'A$	$'C+1 \Rightarrow C$	» »	» »	» »
31	$'('A_1 + 'E_1'A) < -0$	ПУ $'A_2 + 'E_2'A$	ПУ $'A_2 + 'E_2'A$	» »	» »	» »
30	$'('A_1 + 'E_1'A) < -0$	ПУ $'A_2 + 'E_2'A$	$'C+1 \Rightarrow C$	» »	» »	$'A_2 \Rightarrow P$
32	—	ПУ 'Р	—	» »	» »	$0 \Rightarrow P$
26	$'A = 'Ц$	ПУ $'A_2$	$'C+1 \Rightarrow C$	$'A_2 \Rightarrow A$	$'A_1 \Rightarrow Ц$	Не изменяется 24
27	$'A = 'Ц$	ПУ $'A_2$	ПУ $'A_2$	$'A + 'A_1 \Rightarrow A$	24 Не изменяется	То же 25
34	—	$'A_1 \Rightarrow A_2$	—	$'A_1 \Rightarrow A$	То же 25	» »

33 Операции обращения к ВУ			
20	Ввод с переводом чисел в ячейки от $'A_1$ до $'A_2$ 26	23	Запись на МБ 29
21	Ввод без перевода чисел в ячейки от $'A_1$ до $'A_2$ 27	23	Чтение с МБ 30
22	Вывод кодов из ячеек от $'A_1$ до $'A_2$ 28	25	Подготовительная операция для МБ 31
		33	Останов 32

1) Arithmetic and logical operations; 2) control operation; 3) KOp; 4) name of operation; 5) possibility of address modification; 6) emergency halt; 7) tested condition; 8) where satisfied; 9) where not satisfied; 10) yes; 11) addition; 12) subtraction; 13) instruction addition; 14) subtraction of absolute values; 15) cyclic addition; 16) multiplication without rounding; 17) multiplication with rounding; 18) division; 19) logical shift; 20) logical addition; 21) logical multiplication; 22) nonequality; 23) normalization; 24) no change; 25) the same; 26) loading with number conversion into locations from $'A_1$ to $'A_2$; 27) loading without number conversion into locations from $'A_1$ to $'A_2$; 28) reading words out of locations from $'A_1$ to $'A_2$; 29) write onto MB; 30) read from MB; 31) preparatory operation for MB; 32) halt; 33) VU call operations.

REFERENCES

1. Glushkov, V.M., Ob optimal'nom ob"yeme operativnykh zapominayushchikh ustroystv [On Optimum Size of High-Speed Storage Devices], DAN USSR [Proc. Acad. Sci. UkrSSR], 1960, No. 5.

2. Glushkov, V.M., Dva universal'nykh kriteriya effektivnosti

vychislitel'nykh mashin [Two Universal Criteria for the Efficiency of Computers], DAN USSR, 1960, No. 4.

3. Glushkov, V.M., Keruyuchi mashini avtomatizovanogo virobnitstva [Control Computers in Automated Production], Tovaristvo po rozpovsyudzhennyu politichnikh i tekhnichnikh znan' [Society for the Dissemination of Political and Technical Knowledge], Kiev, 1960.

4. Gnedenko, B.V., Glushkov, V.M. and Yushchenko, K.L., Matematichniparametri universal'noi tsifrovoi mashini "Kiiv" [Mathematical Parameters of the "Kiev" General-Purpose Digital Computer], Zbirnik prats' OTs AN URSR [Collected Papers of the Computer Center, UkrSSR Acad. Sci.], Vol. II, Kiev, 1961.

5. Gnedenko, B.V., Korolyuk, V.S. and Yushchenko, Ye.L., Elementy programmirovaniya [Elements of Programming], Fizmatgiz [State Publishing House for Physical-Mathematical Literature], Moscow, 1961.

6. Dashevs'kiy, L.N, Pogrebins'kiy, S.B. and Shkabara, K.O., Strukturna skema ta osnovni printsipi pobudovi tsifrovoi avtomatichnoi mashini "Kiiv" [Structural Diagram and Basic Design Principles of the "Kiev" Automatic Digital Computer], Zbirnik prats' OTs AN URSR, Vol. II, Kiev, 1961.

7. Yershov, Ye.P., Programmiruyushchaya programma dlya BESM [Translator Program for BESM], Moscow, Izd-vo AN SSSR [Acad. Sci. USSR Press], 1958.

8. Ivanenko, L.M. and Yushchenko, K.L., Osnovni pitannya pobudovi programuyuchoy programi dlya mashini "Kiiv" [Basic Problems in the Design of Programming Programs for the "Kiev" Computer], Zbirnik prats' OTs AN URSR, Vol. II, Kiev, 1961.

9. Korolyuk, V.S., Shkabara, K.O. and Yushchenko, K.L., Grupovi

opyeratsii mashini "Kiev" [Group Operations of the "Kiev" Computer], Zbirnik prats' OTs AN URSR, Vol. II, Kiev, 1961.

10. Korolyuk, V.S. and Yushchenko, K.L., Pitannya teorii i praktiki programuvannya [Problems of Programming Theory and Practice], Zbirnik prats' OTs AN URSR, Vol. I, Kiev, 1961.

11. Letichevs'kiy, O.A., Ekvivalentnist' v odnomu klasi adresnikh algoritmiv [Equivalence in One Class of Address Algorithms], Zbirnik prats' OTs AN URSR, Vol. I, Kiev, 1961.

12. Sistema standartnykh podprogramm [System of Standard Routines], Collection edited by Shura-Bura, M.R., Fizmatgiz, Mowcow, 1959.

13. Soobshcheniye ob algoritmicheskom yazyke Algol [Report on the Algol Algorithmic Language], Izd-vo AN SSSR, Moscow, 1960.

14. Faddeyev, D.K. and Faddeyeva, V.N., Vychislitel'nyye metody lineynoy algebry [Computational Methods of Linear Algebra], Fizmatgiz, Moscow, 1960.

15. Yushchenko, Ye.L., Adresnoye programmirovaniye [Address Programming], Postal Seminar "Kibernetika na transporte" [Cybernetics in Transportation], KDNTP [not identified], 1962.

16. Yushchenko, K.L., Adresni algoritmi ta tsifrovi matematichni mashini [Address of Algorithms and Digital Mathematical Machines], Zbirnik prats' OTs AN URSR, Vol. II, Kiev, 1961.

17. Yushchenko, K.L., Adresni algoritmi ta tsifrovi avtomatichni mashini [Address Algorithms and Automatic Digital Computers], DAN URSR [Proc. Acad. Sci. UkrSSR], Vol. VI, Kiev, 1962.

18. Yushchenko, K.L., Rivni ta stili adresnoi movi ta problema avtomatizatsii programuvannya [Address Language Levels and

the Problem of Programming Automation], DAN URSR, Vol. VII, Kiev, 1962.

19. Yushchenko, K.P. and Bistrova, L.P., Programuyucha programa, informatsieyu dlya yakoi sluzhit' adresniy algoritm [A Programming Program Using an Address Algorithm as Information], Zbirnik prats' OTs AN URSR, Vol. III, Kiev, 1961.

20. Yushchenko, K.L., Dryuchina, M.O., Biblioteka pidprogram mashini "Kiev" [Subroutine Library of the "Kiev" Computer], Zbirnik prats' OTs AN URSR, Vol. II, Kiev, 1961.

21. Yushchenko, K.L. and Kostyuchenko, O.I., Algoritm perekladu duzhkovogo zapisu formul u bezduzhkoviy zapis Lukasevicha [Algorithm Translating Bracketed Formula Notation to Unbracketed Lukashevich Notation], Zbirnik prats' OTs AN URSR, Vol. III, Kiev, 1961.

22. Yushchenko, K.L. and Mikhaylova, O.I., Algoritmi formal'noi perevirki pravil'nosti duzhkovogo ta bezduzhkovogo zapisu formul z odno- i dvomistsevimi operatsiyami [Algorithm for Formal Verification of Bracketed and Unbracketed Formular Notation with One- and Two-Position Operations], Zbirnik prats' OTs AN URSR, Vol. III, Kiev, 1961.

23. Yablonskiy, S.V., Trudy matematicheskogo instituta im. Steklova [Transactions of the Steklov Mathematics Institute], Vol. I, AN SSSR [Acad. Sci. USSR], Moscow, 1958.