

ROYAL AIRCRAFT ESTABLISHMENT
(FARNBOROUGH)

TECHNICAL NOTE No: M.S.38

**A PROGRAMMING HANDBOOK
FOR THE COMPUTER DEUCE**

by

D.G.BURNETT-HALL, B.A. & P.A.SAMET, Ph.D., B.Sc.

APRIL, 1959

MINISTRY OF AVIATION, LONDON, W.C.2

UNCLASSIFIED

U.D.C. No. 518.5 : 531.791

Technical Note No. M.S.38

April, 1959

ROYAL AIRCRAFT ESTABLISHMENT

(FARNBOROUGH)

A PROGRAMMING HANDBOOK FOR THE COMPUTER DEUCE

by

D.G. Burnett-Hall, B.A.

and

P.A. Samet, Ph.D., B.Sc.

SUMMARY

The technique of programming the digital computer DEUCE is described.

UNCLASSIFIED

LIST OF CONTENTS

	<u>Page</u>
1 INTRODUCTION	4
PART I THE ESSENTIALS OF PROGRAMMING	5
BASIC	
I.1 Description of a computing machine	5
I.2 Description of DEUCE	7
I.3 DEUCE instruction code	8
I.4 Some simple programs	12
I.5 Loops and counting	16
I.6 The instruction word	24
I.7 Subroutines	34
I.8 Modification of instructions and counting	38
I.9 Automatic modification and counting	43
I.10 Optimum coding	49
I.11 The double length accumulator	51
I.12 The multiplier and divider	57
I.13 Logical operations	62
I.14 Destination 'Triggers'	65
I.15 The reader and the punch	69
I.16 The magnetic drum	78
I.17 Input of programs	82
PART II PROGRAM TESTING	91
II.1 The DEUCE console	91
II.2 Common errors in programs	100
II.3 The use of checking routines	102
II.4 Machine aids to program testing	104
II.5 Programmed aids	107
PART III AVAILABLE PROGRAMS	111
III.1 The DEUCE library	111
III.2 DEUCE library subroutines	115
III.3 List of recommended subroutines	118
III.4 Specifications of programs	124
PART IV SPECIAL TOPICS	126
IV.1 The detailed coding program (R.A.E.415)	126
IV.2 The program changing routine (R.A.E.180)	127
IV.3 Floating point arithmetic	130
IV.4 Interpretive and translating programs	134
IV.5 The R.A.E. translation routine for a single address code (R.A.E.424)	135
IV.6 The N.P.L. General Interpretive Program (G.I.P.)	138
IV.7 Matrix schemes A, B, Z	143
IV.8 Simplified programming, the "alphacode"	146
IV.9 An introduction to logical design	148
PART V EXERCISES AND SOLUTIONS	152
V.1 Exercises	152
V.2 Solutions to the previous exercises	159
LIST OF REFERENCES	186
ADVANCE DISTRIBUTION LIST	187

LIST OF CONTENTS (Contd)

	<u>Page</u>
APPENDICES 1-4	188-198
ILLUSTRATIONS - Figs.1-8	-
DETACHABLE ABSTRACT CARDS	-

LIST OF APPENDICES

Appendix

1 - Function code of DEUCE	188
2 - Binary Arithmetic	191
3 - Useful constants	195
4 - Hollerith timings	197

LIST OF ILLUSTRATIONS

	<u>Fig.</u>
DEUCE logical diagram	1
The DEUCE console	2
Numbering switches on the punch	3
Logical diagrams	4, 5 & 6
Sources 14, 23 and 24	7
The output staticiser	8

1 INTRODUCTION

This Note describes the technique of programming the DEUCE electronic computer. Two such machines* are in operation in Mathematical Services Department.

The original programming manual (Ref.2) was published in 1955, before a machine was available. In it there was an account of how to write programs but no mention was made of techniques used in program testing and the methods that can be used to simplify programming. These techniques, however, are important to all machine users and are not readily accessible to beginners. Many of the methods now in general use, at R.A.E. and elsewhere, are of more recent date than the programming manual. It was therefore decided to write a new manual, which was to be a comprehensive guide to all machine users, especially beginners.

The Note consists of five parts. Part I contains all that is necessary for efficient use of the machine's facilities in the construction of programs. Part II is concerned with program testing. The techniques described were mostly developed at R.A.E. and are the result of much cooperative effort and shared experience in Mathematical Services Department. Part III gives some details of the library of programs and subroutines available for DEUCE, as well as a list of recommended subroutines to perform standard operations. Part IV is meant for the more advanced programmer. It contains sections that deal with specialized topics, such as interpretive programs, and also has a simple introduction to logical design. Part V consists of exercises for the beginner, although others may learn something useful from the examples.

The text is, to a large extent, based on a course given by Mathematical Services Department in May 1957. The order of presentation of the subjects in Part I is the result of experiment with this course.

One small point concerns the spelling of 'program'. There is a rather meaningless controversy about this, some supporting 'program' while others favour 'programme'. The spelling adopted here was used for the EDSAC in Cambridge in 1949, when there were no other machines. We are therefore in good company and the spelling has the virtue of English precedent but really it is only a matter of personal taste.

A word of explanation is needed for the long delay between the time of writing of most of this Note and its publication. The authors both left R.A.E. at the end of the summer of 1957, when most of the text was in manuscript form. Their commitments elsewhere allowed little time for writing the remaining sections until late 1958. By this time certain alterations had been made to the DEUCE, which required a further revision of some sections of the text. Even now (early 1959) the exercises of Part V are fewer than had originally been hoped and there is no index. For all these delays and omissions we beg our readers' pardon.

Thanks are due to many people. To our colleagues in Mathematical Services Department, especially Mr. E.J. York and Mr. J.M. Watt, we owe much. Their comments and suggestions, when followed, have improved the presentation in many places. Mr. J.M. Hahn, of Bristol Aircraft Company, has allowed us to copy Fig.1, 'A Schematic Diagram of DEUCE'. Dr. S.H. Hollingdale has been patient with us and supported our efforts over a long and often frustrating period. To all these persons we say 'thank you'.

* Affectionately known as 'Gert' and 'Daisy'.

PART I THE ESSENTIALS OF PROGRAMMING

I.1 Description of a computing machine

The function of any computing machine, whether it is a simple desk calculator or an electronic high speed machine, is to do arithmetic. Some features that are essential and are common to all machines are:-

- (i) A store
- (ii) An arithmetic unit
- (iii) Input and output mechanism.

The exact nature of each of these requirements varies from machine to machine: for example, on a Brunsviga hand machine the store contains 3 numbers, on DEUCE the total storage space is more than 8,500 numbers and on the newer (American) machines the capacity is several times this figure. Similarly, input and output devices vary from hand-switches and dials to tape readers and punches, etc.

As the speed of the arithmetic operations is increased it becomes essential to store more numbers inside the machine. This allows storage of intermediate results and therefore a certain speeding up of the calculation. If the speed of operations is increased still further, very little of the time of a calculation is spent in computing and a great deal of time is taken up by the operator in deciding what is to be done next. The high speed of operation is completely lost if the operator takes as long to decide on the next step as on a slow machine. One way out of the difficulty is to store the sequence of operations (the 'program') in the machine. The selection of consecutive 'instructions' is now done by the machine and can be accomplished at much the same rate as the arithmetic. Often it is very much faster. It is necessary to allow the machine a conditional choice of next instruction under certain circumstances - a number may be positive or negative and each case may require different consideration. Such a choice is called a 'discrimination'.

It is desirable to make out the program in a general way, i.e. the program should be applicable to a calculation independent of the figures in the particular case. For example, the method of multiplying matrices does not depend on the elements of the matrices. The program therefore is made to refer to the locations in the store, where the required numbers will be found. Every location in the store is given a unique 'address'. To specify a number we give the address where the number is stored. We denote by "word" the contents of any location. Just how numbers find their way into the store in the first place we must leave for the moment.

For machines of the 'stored program' variety it is necessary to have a 'control unit' in addition to requirements (i), (ii) and (iii) given earlier. The function of the control unit is to select instructions in the correct sequence and to initiate the operations required.

The questions arise of how instructions are made up and how they are to be stored in the machine. Let us first consider the nature of the instructions. Every arithmetic operation must specify at least one operand, and in general two, and also where the result is to be stored. This implies that in general three addresses must be given, two 'sources' and one 'destination'. A certain simplification can be achieved if one of the operands is always in a particular register (generally called the 'accumulator') and the result is also placed in the same register. Only one address has to be stated now. However, we must add an extra function, to transfer the contents of the accumulator to other addresses in the store. All the basic functions of which

the machine is capable are numbered (preferably systematically!) and every instruction now specifies the requisite number of addresses and the required function. And this gives the clue as to how instructions are stored - they are kept as if they are numbers. Of the digits that correspond to an instruction some will be the function number and some will be an address (or addresses if several are to be given in every instruction).

The choice of what functions are to be included in the machine's basic set is one of design and varies from machine to machine. Suffice it to say that the user is never satisfied with the set supplied. Sometimes engineering convenience is the dominant factor, at other times simplicity of programming is purchased at the cost of much extra equipment. The functions on modern machines include addition, subtraction, multiplication, division, shifts, comparison and transfer. Other operations can be reduced to combinations of these simple ones.

Storage of instructions as numbers suggests the possibility of doing arithmetic on instructions. This is indeed done and is one of the striking features of automatic computing. Frequently operations are to be performed on each of a sequence of numbers. Rather than store the instructions required for dealing with each term we have them only once and always add sufficient to the address part of the instructions to make them refer to different members in turn. On some machines there are facilities for automatic modification of instructions, on others it has to be programmed. A further consequence of storing instructions in this way is that care must always be taken that the control unit selects only actual instructions not numbers.

Selection of instructions is done in one of two ways. The simpler, and historically earlier, one is to store instructions in consecutive addresses and to go through these in sequence. The other way, adopted on DEUCE, is for each instruction to name its successor. The extra complication of this method is tolerated because, on machines like DEUCE, there can be a great saving of time between operations.

Most electronic machines work in the scale of 2, the binary scale. This is purely for reasons of electronic convenience as the only digits occurring are 0 and 1, and it is very simple to represent one of two states. The computer itself is used to convert numbers to and from the more familiar decimal scale*.

Machines can be classified into two groups, serial and parallel. In the serial machine the digits of any word appear one after the other, with the least significant digit first. Such machines often have delay line storage, where words are stored as trains of pulses which circulate indefinitely in delay lines. DEUCE is of this type. The delay line store is the reason for the non-sequential type of instruction selection - instructions and numbers are placed so that they emerge from the delay line at the right time, with as little waiting time as possible. In parallel machines all the digits of a number are available at the same time and their significance depends on their position in the number. Parallel machines can be much faster than serial ones but require more equipment.

Modern machines generally have more than one type of store attached to them, a high-speed store, with very short access time to all words, and a much larger backing store with longer access time. On DEUCE the high speed store consists of mercury delay lines holding 402 words with a maximum access time of 1024 μ sec. and there is a magnetic drum, with a capacity of 8192 words. Access to a particular word takes about 13 m.sec. in general but may be as long as 50 m. sec.

* Binary arithmetic is discussed in Appendix 2.

All of the foregoing has been quite general and applies to all high speed machines. In future we shall be concerned entirely with DEUCE.

I.2 Description of DEUCE

DEUCE (Digital Electronic Universal Computing Engine) is a machine built by the English Electric Co. Ltd., from designs used by N.P.L. for the ACE Pilot Model (Automatic Computing Engine). The differences between DEUCE and the ACE Pilot Model concern certain rationalizations of design and the size of store.

The machine is of the serial type and uses a system of instructions which state one source, one destination and the 'next instruction source'. No function is stated in the instruction word. Instead, a rather curious system of 'functional sources' and 'functional destinations' is used. Basically, this means that certain locations in the store have more than one address attached to them and these extra addresses allow the operations of arithmetic to be performed. Just how this is done is described in detail in the succeeding chapters. Every instruction specifies the one that is to follow - this system causes certain complications in programming but allows greater speed of operation. Such a system has two addresses for operands and one for the next instruction and is often known as a '2 + 1 address' code.

There are two kinds of storage. The high speed store consists of mercury delay lines and holds 402 words in all. There are 12 long delay lines holding 32 words each and some short lines, known as temporary stores, which hold one, two and four words. The time taken for the contents of one delay line to circulate is called a 'major cycle', the time of one word is a 'minor cycle'. There are 32 minor cycles in one major cycle. The words are all of the same length, 32 binary digits (about 10 decimal digits). Words are stored in the delay lines as a succession of pulses of 1μ sec duration. A minor cycle is therefore 32μ sec and a major cycle is 1024μ sec, or just over 1 msec. All operations are performed in the high speed store.

The secondary storage is a magnetic drum which holds 8192 words, disposed in 256 tracks of 32 words each. The drum itself is a rapidly rotating cylinder coated with magnetic material. There are 'reading heads' and 'writing heads', to transfer information to and from the high speed store. A whole track is transferred with one instruction. Access to the drum is much slower than to the high speed store.

Additional storage on magnetic tape is projected. This has virtually unlimited capacity, and transfer to and from the high speed store is more rapid than with the drum. Access time, however, may be much longer because it may be necessary to search through the tape.

Input and output are by means of conventional Hollerith punched card equipment. Cards may be read at a rate of 200 per minute and punched at 100 cards per minute. Results are printed on an ordinary tabulator - there is no printer attached to DEUCE. 64 columns of a card are used by DEUCE, which means that 24 words punched in binary can be read from one card. Special conversion routines are required to convert decimally punched information into the binary required by the machine and vice versa*.

(The operation of the reader and punch ^{is} controlled entirely by the program.) Extra input and output devices using 5-hole teleprinter tape may be added. Nothing will be said here about such devices.

* Such routines already exist and can be incorporated into programs.

The console, at the front of the machine, has a bewildering array of keys and lights and two cathode ray monitor tubes. These tubes display the contents of the temporary stores and any selected long delay line. Inspection of the monitors can give an indication of how a computation is proceeding and can be used when testing programs. The most important row of hand-switches is called the Input Dynamicizer (I.D.) and can be used as an extra input. One word can be stored on the I.D. switches. There are two rows of lights above the I.D. switches. The lower is connected to the I.D. and shows which keys have been depressed. The upper row of lights is an additional form of output, called the Output Staticizer (O.S.). One word can be displayed on the O.S. lamps. A further row of 13 lamps displays the instruction (in abbreviated form) that is about to be obeyed. These are the Instruction Staticizer lamps (I.S. lamps). Normally they are changing too rapidly to be of any use, but can be very valuable when testing programs. All the other keys are concerned, in some way or other, with facilities for program testing and will be described in Part II.

Physically, the DEUCE consists of a cabinet, roughly 10' x 8' x 6', which houses all the circuitry needed and has the console at one end. The card reader is on the left of the console and the punch is on the operator's right. The delay lines are kept in a thermostatically controlled mushroom-shaped bin, away from the main body of the machine. The power units associated with the machine are also separate. Power consumption is about 8kW and air-cooling of the frame is necessary.

1.3 DEUCE instruction code

In DEUCE (most) instructions involve a transfer of a number in one position in the store (the 'source') to somewhere else in the store (the 'destination'). In general the number in the source is unchanged. Every instruction also specifies the address of the next instruction. This arrangement gives a system known as a '3-address code'.

It is not immediately obvious how moving numbers from one position to another performs any useful operations. The trick is to associate several addresses with a particular storage position and then make each of these addresses have a different function. As an example, one of the stores, called TS 13, has three destinations associated with it, D 13, D 25 and D 26. Sending a number to D 13 replaces the content of TS 13 by this number. If the number is sent to D 25 it is added to the content of TS 13: the sum is now stored in TS 13. Subtraction from TS 13 is done by sending the number to D 26. In this way it is possible to do arithmetic.

Some of the addresses used for sources and destinations do not correspond to a position in the store but to a part of the control unit. An example is D 27, which is the 'discriminator' that examines the sign of a number and affects the choice of the next instruction. Source 27 does not exist as a storage position, either, but always gives a word with a P 1. The reader will notice that there is no connection between source 27 and destination 27.

A complete list of sources and destinations will be found in Appendix 1. The schematic diagram of DEUCE (Fig.1) will also be found helpful.

We now examine the high speed store in some detail.

These are:-

* The DEUCE system is sometimes referred to as a '2 + 1 address code'.

(i) 12 Delay Lines (DL) each holding 32 words. The DL's are numbered 1, ..., 12 and the minor cycles of a DL are numbered 0, ..., 31. We write A_m for minor cycle m of DL A . Only those minor cycles with the same number are available at the same time.

DL's 1, ..., 8 are the only ones available directly as next instruction sources.

(ii) 4 Temporary Stores (TS), numbered 13, 14, 15, 16. They each hold one word, which is available in every minor cycle.

(iii) 2 Quadruple Stores (QS), QS 17 and QS 18: each holds four words. The minor cycles are numbered $17_0, 17_1, 17_2, 17_3$ and $18_0, 18_1, 18_2, 18_3$; minor cycle m is available at the same time as minor cycle $4r + m$ of a DL, (see Appendix 1).

(iv) 3 Double Stores (DS), DS 19, 20, 21, with two words in each. The minor cycles are written as $19_2, 19_3$, etc. (not $19_0, 19_1$). They are available for computation only with minor cycles of same parity.

Addresses 1, ..., 21 used as source or destination refer to the appropriate store. The exact method of addressing a particular word in a DL will be explained later (in I.6). For the moment, we may assume that there is a way of executing an instruction like

$$9_3 - 13,$$

which copies the content of 9_3 into TS 13.

In general, no confusion arises if we drop the letters DL, TS, QS, DS. If necessary, the contents of A_m are denoted by $C(A_m)$. Thus, $C(9_3)$ means 'the word in m.c. 3 of DL 9', $C(14)$ is the word in TS 14, etc. Quite often, one refers to 14, say, meaning $C(14)$. Saying 'Is 14 positive or negative?' does not refer to the number 14, but to $C(14)$. It is convenient to write $C'(n)$ to denote the contents of address n immediately after an operation. On occasion, however, even this is abbreviated to n' .

There are some extra addresses in the computer. These are 0, 22, ..., 31. Their effects are different depending on whether they are used as source or destination. The only special sources and destinations that we shall need for the moment are:-

Sources	0, 23, 24, 27, 28, 29, 30, 31
and	
Destinations	25, 26, 27, 28, 29

Their effects are:-

Source 0: This is the input of the machine. All information that passes from the external world into the computer has to go through S.O. If the card reader is running S.O. gives the word corresponding to the contents of the row just passing the reading brushes. Otherwise S.O. takes the word on the I.D. switches.

Source 23 and Source 24: are connected to TS 14. They give words which are respectively half and double $C(14)$. The instruction

23 - 13
places $\frac{1}{2}C(14)$ in TS 13. Similarly,
24 - 15
puts $2C(14)$ into TS 15.

Because the machine works in binary notation, multiplication and division by 2 corresponds to a bodily shift of the digits in a word. Sources 23 and 24 give a shift down and shift up (of C(14)) respectively. These shifts are correct arithmetic shifts. Care is necessary when doubling a number to avoid exceeding capacity.

It is perfectly permissible to send a number from Source 23 or 24 to destination 14. Note that the effects of

$$\begin{aligned} 23 &- 14 \\ 24 &- 14 \end{aligned}$$

and

$$\begin{aligned} 24 &- 14 \\ 23 &- 14 \end{aligned}$$

need not be the same. In the first case a digit is lost from the least significant end, in the second example we lose the most significant digit. These are the only cases in which the use of Sources 23 and 24 can affect the content of TS 14.

Sources 27, 28, 29, 30, 31 all give special combinations of digits.

S 27 is a P 1	} and nothing else.
S 28 is a P 17	
S 29 is a P 32	
S 30 gives a word consisting entirely of zeros.	
S 31 gives a word consisting entirely of ones (This is - P1).	

The reason for the choice of these combinations will become apparent in due course.

27 - 13
puts a P1 into TS 13,
30 - 14
clears TS 14

Destinations 25 and 26 respectively add to TS 13 and subtract from TS 13.

14 - 25
gives $C'(13) = C(13) + C(14)$,
15 - 26
gives $C'(13) = C(13) - C(15)$

Note that it is possible to use 13 as a source with these destinations. Thus

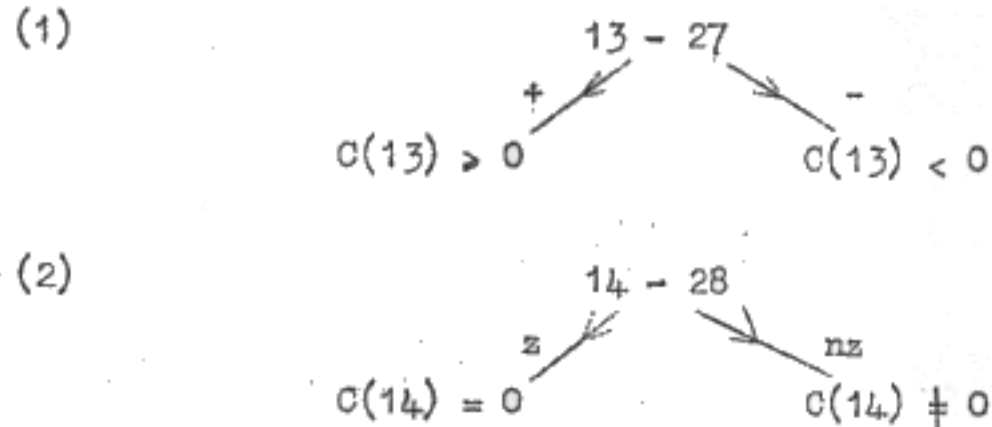
13 - 25
gives $C'(13) = C(13) + C(13) = 2C(13)$,
13 - 26
has the same effect as
30 - 13

Destinations 27 and 28 are the discriminators, which allow a choice of next instruction.

D 27 examines the sign; (zero is a positive number).

D 28 tests for zero.

Examples are:



These discriminators allow one to use a particular loop of instructions several times until some criterion is satisfied. The normal state of the discriminators is 'off', i.e. positive for D 27, zero for D 28. We make the convention in writing out programs that the off side is the left hand branch. It is an additional safeguard to mark the branches, as in the examples. The two alternative instructions following a discrimination must be in consecutive minor cycles of the same DL.

This leaves

Destination 29 This is the output organ of the machine. All information from the machine to the external world has to go through D 29.

If the punch is running, the instruction

A - 29

causes C(A) to be punched on a row of cards. If the punch is idle C(A) is displayed on the O.S. lamps.

All the sources and destinations described affect only one word, of 32 binary digits, at a time. This sort of arithmetic is called 'single length'.

There are special facilities in DEUCE that allow the transfer by one instruction of more than one word. All the transfers illustrated above were single transfers. It is possible to transfer two (consecutive) words or any number up to 32*. When transferring more than two words certain extra conditions have to be satisfied. These will all be explained in I.6. For multiple transfers we write the number of minor cycles in brackets beside the instruction. Examples of such long transfers are:-

* It may soon be possible to transfer four words.

(1) 12 - 1 (32 m.c.)

which transfers all the 32 words in DL 12 into DL 1.

(2) 16 - 17 (4 m.c.)

which puts C(16) into the four minor cycles of Q.S. 17.

(3) 24 - 14 (3 m.c.)

which gives $C'(14) = 2^3 C(14)$.

(4) 11 - 25 (32 m.c.)

which gives $C'(13) = C(13) + C(11_0) + C(11_1) + \dots + C(11_{31})$, i.e. adds all of DL 11 into TS 13, all numbers being single length. Any carry is lost.

The next section, I.4, shows how simple programs can be made up with the apparatus available.

I.4. Some simple programs

There are usually several ways of programming a problem, and often no single best way. The solutions given below should be as good as any the reader can produce.

(1) $x = C(14)$, $y = C(16)$. Put $(x + y)$ in 15.

14 - 13 $C'(13) = x$, $C'(14) = x$.

16 - 25 $C'(13) = x + y$, $C'(16) = y$.

13 - 15 $C'(13) = x + y$, $C'(15) = x + y$.

Notice that if $w = C(13)$ is non-zero, the program

14 - 25 $C'(13) = w + x$

16 - 25 $C'(13) = w + x + y$

13 - 15 $C'(15) = w + x + y$

puts $(w + x + y)$ into TS 15 and so gives an incorrect result. Never assume that any storage location contains zero unless this is stated.

(2) $x = C(14)$, $y = C(16)$. Put $(2y - x)$ in 13

16 - 13 $C'(13) = y$.

16 - 25 $C'(13) = 2y$.

14 - 26 $C'(13) = 2y - x$.

If $w = C(13)$ is not zero, the program

14 - 26 $C'(13) = w - x$

16 - 25 $C'(13) = w - x + 2y$
(2 m.c.)

gives the wrong result.

(3) $x = C(14)$, $y = C(16)$. Put $(2x - y)$ in 13.

$$24 - 13 \quad C'(13) = 2x, \quad C'(14) = x$$

$$16 - 26 \quad C'(13) = 2x - y$$

The number in 14 is unchanged by the first instruction.

(4) $x = C(19_2)$, $y = C(20_3)$. Put $(2x + y)$ in 14.

$$20_3 - 13 \quad C'(13) = y$$

$$19_2 - 25 \quad C'(13) = x + y$$

$$19_2 - 25 \quad C'(13) = 2x + y$$

$$13 - 14 \quad C'(14) = 2x + y$$

The program

$$20_2 - 13$$

$$19_2 - 25 \text{ (2 m.c.)}$$

$$13 - 14$$

is meaningless. If z is the number in 19_3 , the instruction

$$19 - 25 \text{ (2 m.c.)}$$

will add from 19 into 13 for two successive minor cycles, one even and one odd. So x and z will be added into 13 and the result will be $(x + y + z)$. It is impossible to add $2x$ in one instruction in this case.

(5) $x = C(14)$. Put $2x$ in 14.

$$24 - 14$$

(6) $x = C(14)$. Put $8x$ in 14.

$$24 - 14 \text{ (3 m.c.)}$$

At the end of one minor cycle $2x$ has replaced x in 14. So in the second minor cycle $4x$ replaces $2x$, and $8x$ replaces $4x$ in the third minor cycle.

(7) $x = C(14)$. Put $5/4x$ in 14.

$$14 - 13 \quad C'(13) = x, \quad C'(14) = x$$

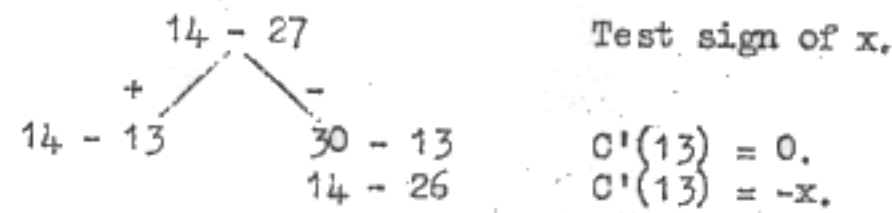
$$23 - 14 \quad C'(13) = x, \quad C'(14) = \frac{1}{2}x$$

$$23 - 25 \quad C'(13) = \frac{1}{2}x, \quad C'(14) = \frac{1}{2}x.$$

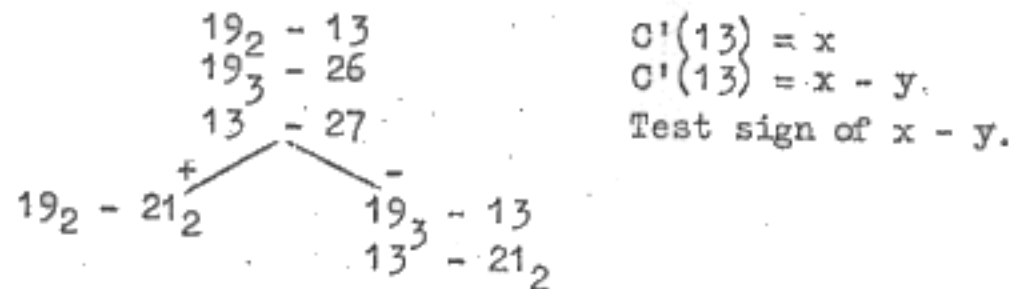
$$13 - 14$$

(8) $x = C(14)$ Put $|x|$ in 13.

$[|x|]$, the "modulus of x ", means the positive number x if x is positive;
the positive number $-x$ if x is negative].



(9) $x = C(19_2)$, $y = C(19_3)$. Put the greater into 21_2 .

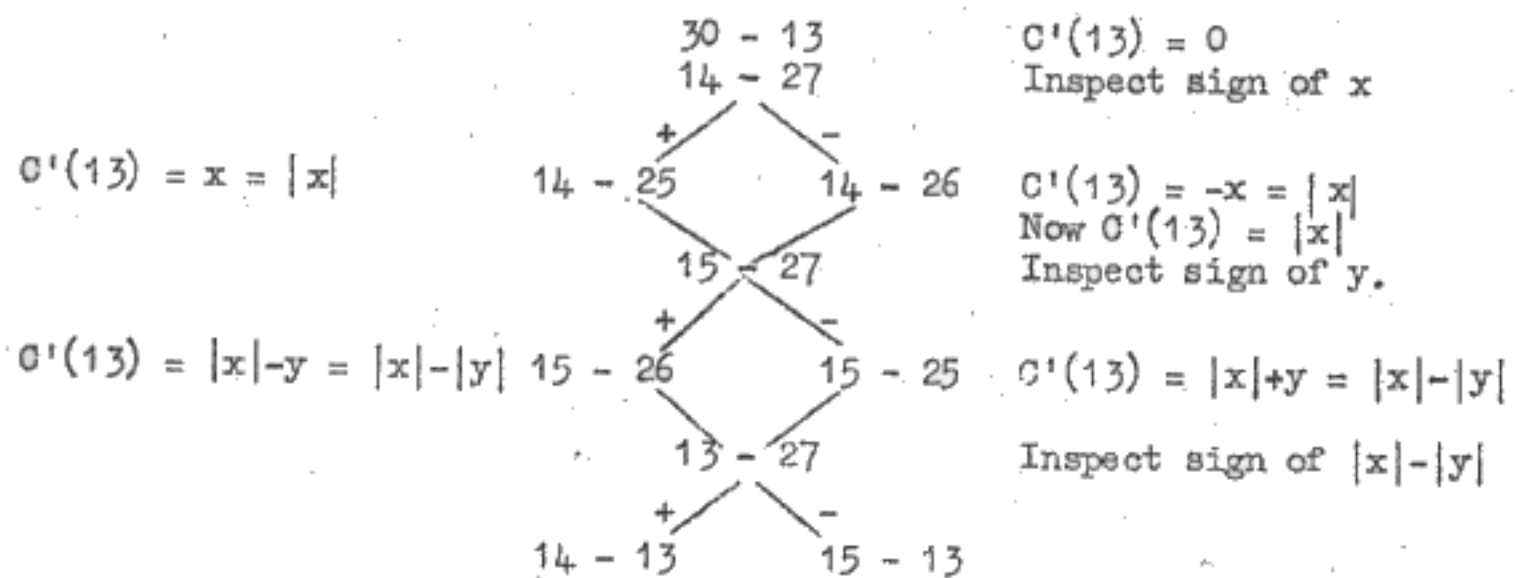


It is impossible to write the one instruction

$$19_3 - 21_2$$

on the negative side of the discrimination, because a word cannot come out of DS 19 in an odd minor cycle and simultaneously go into DS 21 in an even minor cycle.

(10) $x = C(14)$, $y = C(15)$. Put the numerically greater in 13;
i.e. if $|x| \geq |y|$ put x in 13, if $|x| < |y|$ put y in 13.



Notice that it is x or y , not $|x|$ or $|y|$, which must be put in 13 at the end.

There is a shorter way of doing this example when source 26 has been explained; see I.13.

Although the use of the multiplier will not be explained in full until I.12, more interesting examples can be constructed if multiplication is possible. So the following method of multiplying will be used for the moment:

If x and y are the fractions to be multiplied, put them in 14 and 16 and write MULT as an order. The product xy will be found in 21_3 . The numbers in 14 and 16 are undisturbed, but the numbers in 13, 15, 21_2 and 21_3 are lost.

(11) $x = C(17_0)$, $y = C(17_1)$. Put xy in 17_2 .

$$17_0 - 14 \quad C'(14) = x.$$

$$17_1 - 16 \quad C'(16) = y.$$

$$\text{MULT} \quad C'(21_3) = xy.$$

$$21_3 - 13 \quad C'(13) = xy.$$

$$13 - 17_2 \quad C'(17_2) = xy.$$

(12) $x = C(14)$. Calculate x^{13} as quickly as possible. Multiplication is a slow process compared with addition and subtraction, so the aim must be to use as few multiplications as possible. 5 multiplications are necessary

$$14 - 16 \quad C'(16) = x, \quad C'(14) = x$$

$$\text{MULT} \quad C'(21_3) = x^2$$

$$21_3 - 16 \quad C'(16) = x^2, \quad C'(14) = x$$

$$\text{MULT} \quad C'(21_3) = x^3$$

$$21_3 - 14 \quad C'(14) = x^3, \quad C'(16) = x^2$$

$$\text{MULT} \quad C'(21_3) = x^5$$

$$21_3 - 16 \quad C'(16) = x^5, \quad C'(14) = x^3$$

$$\text{MULT} \quad C'(21_3) = x^8$$

$$21_3 - 14 \quad C'(14) = x^8, \quad C'(16) = x^5$$

$$\text{MULT} \quad C'(21_3) = x^{13}$$

$$21_3 - 13 \quad C'(13) = x^{13}.$$

(13) r is an integer. Given $r = C(15)$, $r^2 = C(16)$, put $(r + 1)$ in 15 and $(r + 1)^2$ in 16 and punch $(r + 1)^2$; assume that the punch is running. This can be done without using the multiplier because $(r + 1)^2 = r^2 + 2r + 1$; in fact this is the quickest way. In any case the method of multiplication described above applies only to fractions.

16 - 13	$C'(13) = r^2$
15 - 25 (2 m.c.)	$C'(13) = r^2 + 2r$
27 - 25	$C'(13) = r^2 + 2r + 1$
13 - 16	$C'(16) = (r + 1)^2$
15 - 13	$C'(13) = r$
27 - 25	$C'(13) = r + 1$
13 - 15	$C'(15) = r + 1$
16 - 29	Punch $C(16) = (r + 1)^2$

If the last instruction is made to lead back to the first one, the next time this string of instructions has been obeyed $(r + 2)^2$ will be in 16 and $(r + 2)$ in 15; the following time $(r + 3)^2$ and $(r + 3)$ will be in 16 and 15 respectively.

I.5 Loops and counting

One of the great advantages of digital computers is their speed of operation. On DEUCE about 16,000 instructions could be obeyed in one second. This is hardly an advantage if all 16,000 instructions have to be written out and punched separately. The speed of a digital computer can only be utilised if it is possible to obey a loop of instructions many times and eventually to leave the loop.

The decision whether or not to leave the loop will consist of a discrimination on some quantity which will be encountered each time round the loop. What this quantity is depends entirely on the problem; sometimes it will be suggested by the form of the problem (see Example 4 below); sometimes it will be a counter which goes up (or down) in steps of 1 every time the loop is traversed, and this counter may or may not be used elsewhere in the loop - Examples 2 and 3 below give different uses of counters.

Example 1

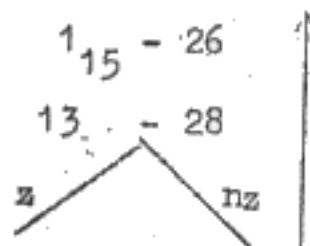
Punch the squares of the numbers 1, 2, ... 10 in binary.

Example 13 of I.4 gives a method by which, if r^2 and r are given in 16 and 15, they can be replaced by $(r + 1)^2$ and $r + 1$, and $(r + 1)^2$ punched. So if the last instruction leads straight back to the first the numbers $(r + 1)^2$, $(r + 2)^2$, $(r + 3)^2$ and so on will be punched. What remains to be done is to arrange to start with the right value of r and to come out of the loop at the right time.

It is a good principle in general to arrange the method of leaving the loop before deciding how to start going round it, because the initial conditions which have to be set before the loop is entered may depend on what test is used at the end. The next example will show this more clearly, but this principle will be followed in all examples.

After the sequence of instructions given in Example 13 of I.4 have been obeyed, 13 and 15 both contain $(r + 1)$ and 16 contains $(r + 2)^2$; $(r + 1)^2$ has just been punched. Now suppose that the number 10 has been stored somewhere, say in 145.

Add the instructions



$$C'(13) = (r + 1) - 10$$

to those already given, with the non-zero side of the discrimination leading to the 16 - 13 instruction. The zero side will be taken only when $r + 1 = 10$, i.e. 10^2 has just been punched.

If the 16 - 13 instruction is preceded by

30 - 16

30 - 15

we have arranged that $r^2 = 0$ and $r = 0$ initially. Therefore 1^2 will be the first number punched.

The complete program is:

[illegible]

(Three details, the 10 - 24 and 9 - 24 instructions and the X after the 16 - 29 instruction, have been put in to make the example realistic. They are explained in I.13.)

This method has considerable flexibility. If the squares of the integers up to 1000^2 were wanted, the only modification necessary would be to change the constant 10 in 1_{15} to 1000.

Example 2

$x = 0(14)$. Calculate x^{13} in as few instructions as possible and place it in 13.

Suppose at some stage x^r is in 16 and that x is never disturbed from 14. Then the instructions

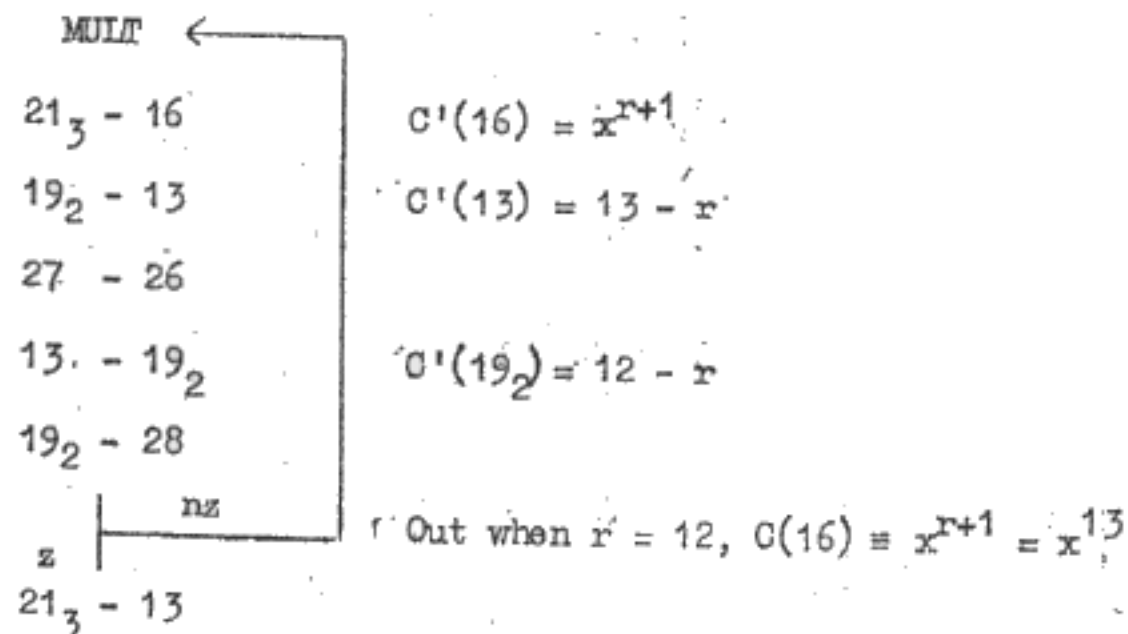
MULT

$$21_3 - 16$$

replace x^r by x^{r+1} . These will be the basic instructions of our loop.

We must next decide how to get out of the loop. In the previous example we had a number r stored in 15 which increased by 1 each time round the loop; a discrimination on r was used to make the exit from the loop. This time we do not have a convenient counter like r already in use, so one will have to be introduced. It turns out that it is a little more convenient to keep $(13 - r)$, which goes down in steps of 1 until it reaches 0, than r , which would go up in steps of 1 until it reaches 13. (The reader might like to do this case as an exercise.)

This counter cannot be kept in 13, 15 or 21_2 because MULT disturbs the contents of these stores, so it will be kept in 19_2 . Therefore, if 19_2 contains $(13 - r)$ when 16 contains x^r , we must change $C(19_2)$ to $(12 - r)$ when we change $C(16)$ to x^{r+1} . So the loop consists of the instructions



We must now arrange to enter the loop with x^r in 16 and $(13 - r)$ in 19_2 for whatever value of r we like to choose. It is simplest to put x in 16, so 12 must be put into 19_2 .

As in the last example, when the constant 10 was assumed to be stored in 15, we shall need to have a constant. How this constant gets into the store is a question which often worries beginners; it cannot be satisfactorily answered until the way in which the program is read in is explained in I.17, but it can be said now that it gets into the store at the time the instructions are read in.

Suppose that the constant is stored in 1_{18} . The obvious choice of constant seems to be 12, so that we add the two instructions

$$14 - 16 \quad C'(16) = x$$

$$1_{18} - 19_2 \quad C'(19_2) = 12$$

at the head of the loop. As we have to calculate x^{13} it would look tidier if the constant 13 were used; but it looks as though four instructions

$$\begin{array}{lll}
 14 & - & 16 \quad C'(16) = x \\
 1_{18} & - & 13 \quad C'(13) = 13 \\
 27 & - & 26 \quad C'(13) = 12 \\
 13 & - & 19_2 \quad C'(19_2) = 12
 \end{array}$$

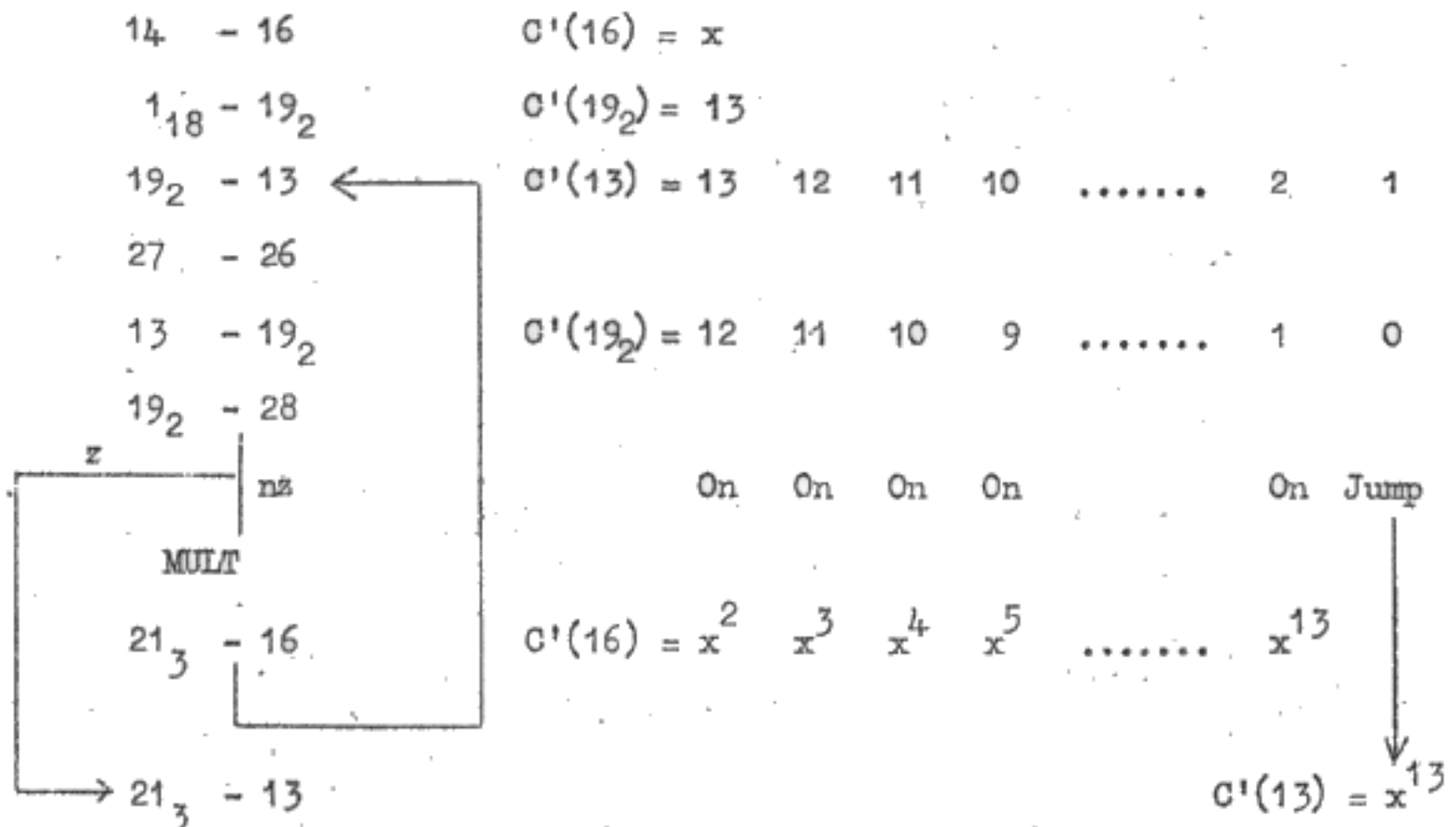
would have to be added at the head of the loop.

But suppose we have two instructions

$$\begin{array}{lll}
 14 & - & 16 \quad C'(16) = x \\
 1_{18} & - & 19_2 \quad C'(19_2) = 13
 \end{array}$$

after which we enter the loop at its third instruction, $19_2 = 13$. Then 1 will be subtracted from the number 13 in 19_2 before any multiplication is done.

Writing the instructions in a different order on the paper, but obeying them in exactly the same order, we get the final form of the loop:



There are 6 general points which this example illustrates:-

- (a) The order in which the loop was constructed was:
 - (i) Basic orders: form x^{r+1} from x^r ;
 - (ii) "Red-tape" orders: changing the counter and testing its value;
 - (iii) Setting the initial conditions;

although in its final form of the loop this is not apparent.

Moral: When a loop is being constructed the programmer should keep to this order of working.

(b) If the arrow leading from the $21_3 - 16$ instruction to the $19_2 - 13$ instruction had been taken one line higher to include the $1_{18} - 19_2$ instruction in the loop, the number in 19_2 would be set at 13 every time round the loop and so the loop would never be left. This may seem a stupid thing to do, but it illustrates one of the common mistakes in programming.

Moral: Do not include in the loop one or more of the instructions which set initial conditions.

(c) The three instructions

$19_2 - 13$

$27 - 26$

$13 - 19_2$

reduce the counter, and the one instruction

$19_2 - 28$

following then tests the counter. In fact, what the program does is

Reduce count.

Test count.

If, instead, the sequence had been

Test count.

Reduce count.

the result would have been to produce x^{14} unless the constant in 1_{18} were changed from 13 to 12.

The blunder of going round a loop one time too few or too many is probably the most common of all programming mistakes.

Moral: Beginners (and others) would be well advised always to use the sequence

Change count

Test count.

(d) The reader may wonder why the number 13 was transferred from 1_{18} to 19_2 before being used for counting. There are two reasons of which the second is the more important.

Firstly, the sequence

$1_{18} - 13$

$27 - 26$

$13 - 1_{18}$

takes longer to obey than that used in the loop. The number 13 cannot be stored in DS 19 when the program is being read in, but it can be stored in a delay line.

Secondly, in the loop given, the number in 1_{18} is not disturbed. So if there is an outer loop leading back from the $21_3 - 13$ instruction at the end of this inner loop to the $14 - 16$ at its beginning, leaving y in 14 this time, the inner loop will produce y^{13} in 13 . If counting had taken place in 1_{18} there would have been 0 in 1_{18} the second time the loop was started, and -1 when the test was first made; to get a correct result the second time, an instruction in the outer loop would have to reset the number 13 in 1_{18} before the inner loop was re-entered. A section of program for which this does not have to be done, such as the example given, is called "self-resetting".

Moral: Each section of a program should be made self-resetting unless there is a good reason to the contrary.

(e) No extra thought is needed to make this program calculate x^{15} , say, instead of x^{13} . All that needs modifying is the parameter in 1_{18} . Compare this with example 12 of I.4, which would have to be completely reprogrammed. Of course this gain in flexibility is accompanied by a loss in speed; the two usually go together. So do a saving in the number of instructions and a loss in speed.

Moral: Programs should be made flexible unless speed is essential.

(f) The constant in 1_{18} is n if x^n is to be calculated. If "Test count" has preceded "Change count" (see (d) above) it would have to be $(n - 1)$. The parameter n is much more likely to have been used in another part of the program than the parameter $(n - 1)$.

Moral: The parameters of a program should be the convenient ones.

Example 3

Given x in (16) , and 10.2^{-14} in 19_3 , sum the polynomial $2^{-14}(10x^9 + 9x^8 + \dots + 1)$ and put the result in 15 . On DEUCE fractions are normally regarded as having 30 binary digits below (to the left of) the binary point and 2 bits above the point. The fraction 2^{-14} will consist of one digit only, the P_{17} digit. This digit is always given by source 28.

The method used to sum the polynomial is that known as "nested multiplication" and is commonly used on desk machines. We evaluate $(\dots(((10.2^{-14}x + 9.2^{-14})x + 8.2^{-14})x + 7.2^{-14}) \dots)x + 1.2^{-14}$.

Suppose at some stage we have

$$S_r = 10.2^{-14}x^{9-r} + 9.2^{-14}x^{8-r} + \dots + (r+1)2^{-14}$$

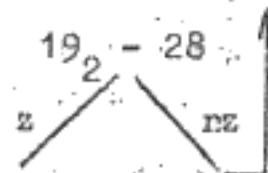
in 14 . Obviously we need to end when $r = 0$.

One step in the calculation consists of the multiplication of S_r by x and the addition of $r.2^{-14}$; the result will be S_{r-1} . We shall need to keep a counter $r.2^{-14}$, say in 19_2 , and this counter will have to be reduced by 2^{-14} .

The operative orders in the loop are:

MULT $C(14) = S_r, C(16) = x$, undisturbed
 $21_3 - 13$ $C'(13) = S_r x$
 $19_2 - 25$ $C'(13) = S_r x + r 2^{-14} = S_{r-1}$
 $13 - 14$ $C'(14) = S_{r-1}$
 $19_2 - 13$ $C'(13) = r 2^{-14}$
 $28 - 26$ Source 28 gives P_{17} , i.e. 2^{-14}
 $13 - 19_2$ $C'(19_2) = (r - 1) 2^{-14}$

We have now replaced S_r and $r 2^{-14}$ by S_{r-1} and $(r - 1) 2^{-14}$ respectively. Work has finished when S_0 is obtained, so one further instruction.



is needed to close the loop. The non-zero side leads back to MULT.

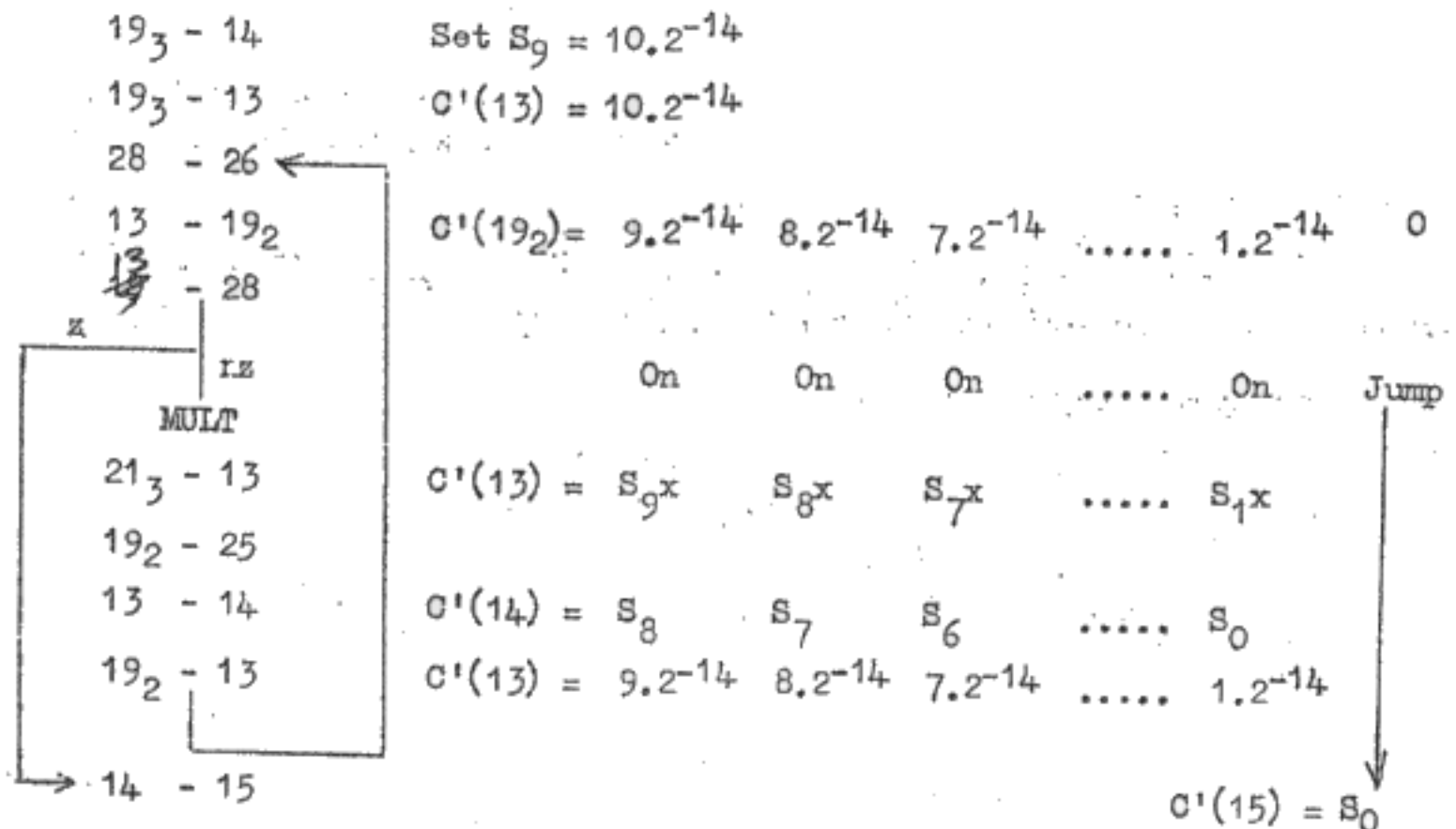
Finally we set the initial conditions. There are two alternatives here; either $S_{10} = 0$ and 10.2^{-14} in 14 and 19_2 respectively, or $S_9 = 10.2^{-14}$ and 9.2^{-14} in 14 and 19_2 respectively. By a trick similar to that in example 2 we can use the second alternative and so save going round the loop once compared with the first alternative.

We start with the instructions

$19_3 - 14$ $C'(14) = 10.2^{-14} = S_9$
 $19_3 - 13$ $C'(13) = 10.2^{-14}$

and enter the loop at the 28 - 26 instruction, not at the MULT instruction.

The full program is:



The reader should check that none of the morals of the previous example ~~are~~ offended.

As far as moral (e) is concerned, he may notice that changing the number in 19_3 will give different approximations to the infinite series

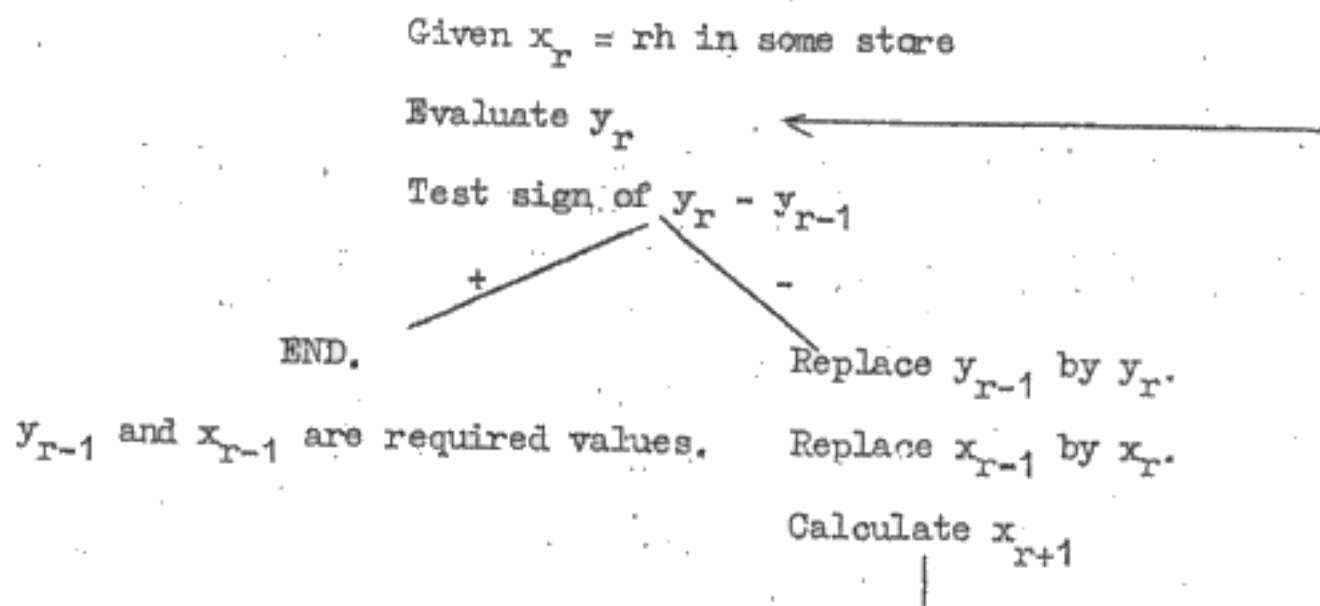
$$2^{-14} (1 + 2x + 3x^2 + 4x^3 + \dots) = 2^{-14} (1 - x)^{-2}.$$

Example 4

Given constants a , b , c and h in 17_1 , 17_2 , 17_3 and 17_0 respectively where a , b , and h are strictly positive, find the value of r (an integer) such that $x_r = rh$ gives the smallest value of $y_r = ax_r^4 - bx_r + c$. Place the corresponding values of x_r and y_r in 19_2 and 19_3 respectively. Assume that numbers are within the capacity of the machine.

If $y = ax^4 - bx + c$, $y' = 4ax^3 - b$. The gradient is negative at $x = 0$ and becomes 0 at a point where x is positive. The question does not ask that this point should be found; what is wanted is that y should be evaluated for $x = 0, h, 2h, 3h, \dots$ and the smallest y found.

The basic operations of the loop will be:

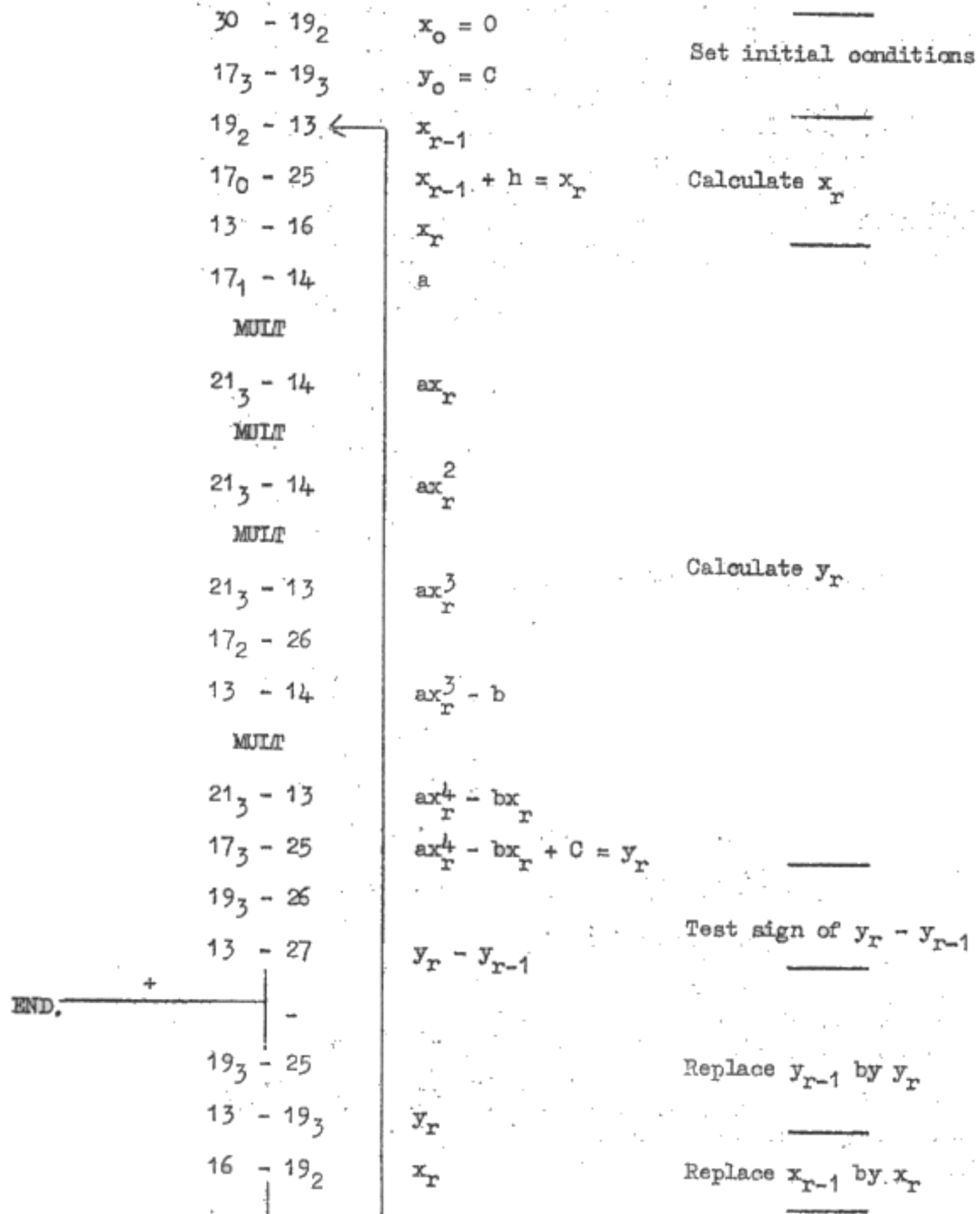


Notice that x_{r-1} and y_{r-1} are both needed until the discrimination has taken place.

The first discrimination must be on $y_1 - y_0$, as the simplest method of starting the calculation is to set $x_0 = 0$, $y_0 = c$ and enter the loop at "Calculate x_{r+1} ".

We shall keep x_{r-1} and y_{r-1} in 19_2 and 19_3 ; x_r will be stored in 16 when it has been calculated for use in forming y_r .

The full program is:



In this example it is y_r that determines whether the exit from the loop should be made, although the other variable x_r is the counter which goes up in equal steps.

1.6 The instruction word

Until this point we have written flow diagrams for programs without considering how they are fed into the machine or what the machine does with them. This section describes the form of the instruction used inside DEUCE which is not the same as that used so far.

So far we have not considered where each instruction is stored when writing flow diagrams. It is only when the flow diagram has been written that the next stage allocation of storage positions for the instructions and for constants can take place; and this is followed by the stage known as "detailed coding", in which instructions are written out in the exact form in which they are stored in the machine. The second stage allocation of stores is an art in itself. An experienced programmer will usually code a loop of instructions to take less time in operation than would an inexperienced man: this is the art of "Optimum Programming" (I.10), often (wrongly) considered to be one of DEUCE'S chief snags by programmers used to other machines. It can only be learnt by practice when the methods of detailed coding are familiar, so in this section we shall assume that storage locations have already been allotted and that this has not been done very well.

DEUCE'S word-length of 32 bits is rather shorter than that of more modern machines. These often have a word of about 40 bits and pack 2 instructions into a word: because of its cumbersome order code DEUCE has only one instruction in each word.

A DEUCE instruction has eight things to specify:-

- (i) Source
- (ii) Destination
- (iii) Minor cycle when transfer starts
- (iv) Length of transfer
- (v) Delay line of next instruction
- (vi) Minor cycle of next instruction
- (vii) Whether the instruction is a stopper
- (viii) Auxiliary staticizer

Source and destination

When we write the instruction $1_{15} - 26$ we mean that the machine must make a transfer from Source 1 to Destination 26, starting (and ending) in minor cycle 15. In the detailed form of the DEUCE instruction the source number will be 1 and the destination number 26.

Wait number

The designers of DEUCE might have decided that the transfer specified by each instruction should always take place 2 minor cycles after the minor cycle of the instruction*. This would have meant that any instruction referring to a number in 1_{15} would have to be stored in minor cycle 13, and so there could be only 8 such instructions in the high-speed store at any one time. This would place an impossible burden on the programmer; so 5 bits in the instruction, the Wait Number, are provided to specify the minor cycle in which transfer starts. The wait number is not just the number of this minor cycle. The precise form is described below.

* This is virtually what the designers of MOSAIC did decide, as rather different considerations applied on their machine.

Next Instruction Source (NIS) and Timing Number

Some machines expect the instructions to be stored in the sequence in which they are written in the flow diagram so that instructions generally do not specify where their successors are stored; but on DEUCE every instruction specifies the delay line and the minor cycle in which the next instruction is stored, and a string of instructions obeyed consecutively may be scattered through the store.

Instructions can go into the control of DEUCE to be obeyed from delay lines 1 to 8 only, so 3 binary digits are needed to specify the Next Instruction Source - always abbreviated to NIS. NIS 1 to 7 refer to delay lines 1 to 7 respectively and NIS 0 is used for DL 8. In addition a 5-bit number, the Timing Number, specifies which of the 32 words in that delay line is the next instruction; but the timing number like the wait number is not just the number of the minor cycle of the next instruction.

Characteristic

In I.3 the possibility of one transfer going on for up to 32 minor cycles was mentioned. The length of the transfer depends on the wait number, the timing number and a 2-bit number called the Characteristic; but whatever the value of the characteristic, the wait number specifies the first minor cycle of transfer and the timing number specifies the position of the next instruction*. DEUCE like certain aborigines, distinguishes only between one, two and many minor cycles of transfer; the three cases are called single transfer, double transfer and long transfer and the corresponding values of the characteristic are 0, 2 and 1.

Characteristic 3 at present has rather an odd effect; it should never be used because a proposal has been made to modify DEUCE and produce a different result.

Go digit

The most significant bit of the word, used as the sign digit of a number, is the Go Digit of an instruction. Normally it is 1; but if it is 0 the machine waits, just before obeying the instruction, until a pulse called a "single-shot" is given. Then the instruction is obeyed and normal operation continues.

The single-shot can be given from a key on the console: this facility is used when testing programs. It is also given by the reader or the punch whenever a row of a card comes into position. On the punch for instance the user makes the instruction to punch each row a "stopped" instruction by making the go digit 0 - he writes a cross after the instruction in the flow diagram, e.g.

16 - 29 X

When this instruction is reached the machine waits for a single-shot before obeying it; as the next row of the card comes into position under the punch knives a circuit in the punch supplies the single-shot and the number 16 is punched. The machine then goes on to obey the following instructions at full speed.

Auxiliary Statioizer

This is used in Automatic Instruction Modification and is explained in 9.

*Except, perhaps, in the case of a discrimination.

Layout of instruction word

Representing each of the 32 digits by a 1, the instruction is laid out as follows:

<u>1</u>	<u>111</u>	<u>11111</u>	<u>11111</u>	<u>11</u>	<u>11111</u>	<u>1111</u>	<u>11111</u>	<u>1</u>	<u>1</u>
A	NIS	S	D	Ch	W	J	T		G

Digit 1	:	A	Auxiliary staticizer: used in automatic modification of instructions (see I.9)
Digits 2-4	:	NIS,	the Next Instruction Source
Digits 5-9	:	S,	the Source
Digits 10-14	:	D,	the Destination
Digits 15,16	:	Ch,	the Characteristic
Digits 17-21	:	W,	the Wait number
Digits 22-25	:	J,	the Joe digits (etymology unknown); not used in interpreting the instruction; but see I.8 for their use in counting
Digits 26-30	:	T,	the Timing number
Digit 31	:		not used in interpreting the instruction
Digit 32	:	G,	the Go digit.

Wait and timing numbers

There is a fifth temporary store, TS Count, as well as TS 13 to 16. In operation it behaves very differently from them because instructions are sent to it to be obeyed; and they do not usually go to it as they would to any other destination, but along a special Next Instruction highway. (Nothing can be taken out of TS Count into the rest of the store.)

If an instruction specifies that the next one is stored in 1₁₇ circuits allow one instruction to flow from DL 1 along this special highway into TS Count during minor cycle 17. Minor cycle 18 is spent in setting-up this instruction, so this minor cycle is known as the set-up minor cycle. The earliest that this instruction can be obeyed is in minor cycle 19; in this case the wait number is 0. If the transfer is not to start until minor cycle 20 the wait number must be 1; if the machine is to wait two minor cycles until minor cycle 21 the wait number is made 2.

In general, if an instruction stored in minor cycle m is to be obeyed starting in minor cycle F , the wait number is made $(F - m - 2)$ because, when the instruction is obeyed, the order of events is:

m.c. m	Instruction flows into TS Count
m.c. $m + 1$	Set-up minor cycle
m.c. $m + 2$ to m.c. $F - 1$	Wait for $(F - 1) - (m + 2) + 1 = F - m - 2$ minor cycles
m.c. F	Instruction starts to be obeyed.

So the wait number W is defined by

$$W = F - m - 2 \pmod{32}$$

$$\text{Conversely } F = m + W + 2 \pmod{32}$$

(The words "mod 32" are included because F , m and W must all lie between 0 and 31: they say that if a number calculated by one of these relations is not in this range, 32 must be added or subtracted until it does.)

Exactly the same procedure holds when the timing number is being calculated. If an instruction in minor cycle m is followed by one in minor cycle t , all the minor cycles from $(m + 2)$ to $(t - 1)$ inclusive are spent waiting for the next instruction - the instruction may be obeyed during this time - so the timing number T is defined by

$$T = t - m - 2 \pmod{32}$$

$$\text{Conversely } t = m + T + 2 \pmod{32}$$

The way in which the machine obeys any instruction is:

If the instruction N , S - D , Ch , W , T is stored in minor cycle m , its effect is to start the transfer S - D in minor cycle $(m + W + 2)$ and then to fetch the next instruction from minor cycle $(m + T + 2)$ of delay line N (DL 8 if $N = 0$).

There are three points still to be considered:

- (i) what happens when the wait number is greater than the timing number;
- (ii) the exact effect of the characteristic,
- (iii) the effect of discriminations.

Wait number greater than timing number

Let us code the three instructions

430	21 - 13
43	28 - 25
45	13 - 21

leading to 57.

The instruction in 430 has wait number $1-30-2 = 1 \pmod{32}$ and timing number $3-30-2 = 3 \pmod{32}$. Its detailed form is 4, 2-13, 0, 1, 3. (This shows a property of instructions in m.c. 30 - the wait number is just the first minor cycle of transfer and the timing number is just the minor cycle of the next instruction. This does not hold for instructions stored in any other minor cycle.)

The instruction in 43 can be obeyed at any time: it is usual to obey it as soon as possible, in m.c. 5 (m.c. 4 is the set-up minor cycle). The wait number is $5-3-2 = 0$. The timing number is the same because the next instruction is in m.c. 5, so the detailed form of the instruction is 4, 28-25, 0, 0, 0. In minor cycle 5 the machine is

simultaneously obeying 28 - 25 and picking up the next instruction; it is quite capable of doing this.

The instruction in 4₅ has wait number $1-5-2 = 26 \pmod{32}$ and timing number $7-5-2 = 0 \pmod{32}$, so its detailed form is 5,13-2,0,26,0. This time the next instruction becomes available (in m.c. 7) before the present one is available in m.c. 1; the fact that the wait number is greater than the timing number is symptomatic of this. If the next instruction went into TS Count in m.c. 7 it would cancel the setting-up of the instruction which is still waiting to be obeyed. To prevent this, circuits in the control of DEUCE prevent a new instruction from going into TS Count until the previous one has started to be obeyed. In the present case the order of the operations will be:

m.c. 5	Instruction enters TS Count
m.c. 6	Set-up minor cycle
m.c. 7	Next instruction available, not allowed into TS Count
m.c. 8	} Waiting before obeying instruction
⋮	
⋮	
m.c. 10	
m.c. 1	Transfer 13 - 2 occurs
m.c. 2	} Waiting for next instruction
⋮	
⋮	
m.c. 6	
m.c. 7	Next instruction enters TS Count.

If the programmer applies the rule-of-thumb $T = t - m - 2 \pmod{32}$, and thinks of the machine (not himself) as adding 32 to the timing number when this is necessary to make it larger than the wait number, he will get the right results.

Effect of the Characteristic

The example given above used only single transfers, i.e. transfers for one minor cycle, so each instruction had characteristic 0. The operation of the characteristic must now be explained.

Characteristic 0, Single Transfer

The transfer takes place for the one minor cycle, m.c. $(m + W + 2)$; the next instruction is picked up in m.c. $(m + T + 2)$. If $W > T$ the machine waits for one major cycle before picking up the next instruction.

Characteristic 2, Double Transfer

The transfer takes place for two minor cycles, m.c. $(m + W + 2)$ and $(m + W + 3)$; the next instruction is picked up in m.c. $(m + T + 2)$. If $W > T$ the machine waits for one major cycle before picking up the next instruction; notice that $W = T$ is included in this case.

Characteristic 1, Long Transfer

The transfer starts in m.c. $(m + W + 2)$ and ends in m.c. $(m + T + 2)$ the minor cycle when the next instruction is picked up. If L is the length of transfer

$$L = (m + T + 2) - (m + W + 2) + 1 = T - W + 1 \pmod{32}$$

$$\text{Conversely } W = T - L + 1 \pmod{32}$$

As with characteristic 0, $W > T$ causes the machine to wait for one major cycle before picking up the next instruction.

Characteristic 3

The present effect of characteristic 3 is that of characteristic 1, with the single exception that, when $W = T$, characteristic 3 causes a transfer for 33 minor cycles and characteristic 1 causes a transfer for only 1 minor cycle.

A proposed modification of DEUCE would make characteristic 3 a quadruple transfer rather like double transfer; transfer would take place in minor cycles $(m + W + 2)$ to $(m + W + 5)$ inclusive. As the present effect of characteristic 3 is of no special value it should never be used.

Discrimination instructions

When the "off" side of a discrimination is taken (positive for D 27, zero for D 28, absence of TIL for 2 - 24) the next instruction is taken from minor cycle $(m + T + 2)$ of delay line N; so, when a discrimination instruction is coded, the next instruction is always taken as the one on the "off" side. If the "on" side is taken, however, the next instruction is taken from minor cycle $(m + T + 3)$ of delay line N. This is the reason for saying that the instructions on each side following a discrimination have to be stored in successive minor cycles of the same delay line.

/Example

Example

Code the following program:

(1)	→ 5 ₈	2 ₁₂ - 14	
(2)	6 ₁₇	19 ₃ - 7 ₂₅	
(3)	5 ₂₉	18 ₀ - 12 ₁₂	
(4)	4 ₁₂	13 - 15	
(5)	1 ₁₄	20 ₃ - 26	
(6)	8 ₁₇	0 - 17 ₁ X	
(7)	7 ₂₂	18 ₀ - 19 ₂	
(8)	3 ₂₉	18 ₃ - 27	
		+	
(9)	5 ₉	9 _{6,7} - 21 _{2,3} (2 m.c.)	
(10)	1 ₁	6 ₇₋₉ - 17 _{3,0,1} (3 m.c.)	
(11)	8 ₉	19 - 25 (2 m.c.)	
(12)	6 ₁₆	21 _{2,3} - 17 _{0,1} (2 m.c.)	
(13)	6 ₂₄	17 - 18 (4 m.c.)	
(14)	2 ₃	10 - 25 (32 m.c.)	
(15)	3 ₇	24 - 14 (5 m.c.)	
(16)	5 ₁₁	21 - 22 (4 m.c. e.o.)	
(17)	8 ₃₁	11 - 8 (32 m.c.)	
	8 ₀		

As a program for DEUCE this example is useless: its only object is to demonstrate certain points in coding. The instructions have been numbered for reference in the section of comments which follows the results.

		N,	S - D,	Ch,	W,	T
(1)	5_8 contains	6,	2 - 14,	0,	2,	7
(2)	6_{17} "	5,	19 - 7,	0,	6,	10
(3)	5_{29} "	4,	18 - 12,	0,	13,	13
(4)	4_{12} "	1,	13 - 15,	0,	0,	0
(5)	1_{14} "	0,	20 - 26,	0,	1,	1
(6)	8_{17} "	7,	0 - 17,	0,	2,	3 X
(7)	7_{22} "	3,	18 - 19,	0,	0,	5
(8)	3_{29} "	5,	18 - 27,	0,	0,	9
(9)	5_9 "	1,	9 - 21,	2,	27,	22
(10)	1_1 "	0,	6 - 17,	1,	4,	6
(11)	8_9 "	6,	19 - 25,	2,	0,	5
(12)	6_{16} "	6,	21 - 17,	2,	2,	6
(13)	6_{24} "	2,	17 - 18,	1,	6,	9
(14)	2_3 "	3,	10 - 25,	1,	3,	2
(15)	3_7 "	5,	24 - 14,	1,	30,	2
(16)	5_{11} "	0,	21 - 22,	1,	15,	18
(17)	8_{31} "	0,	11 - 8,	1,	0,	31

In the comments which follow, the sequence in which the detailed coding is worked out will always be the same:

S, D, Ch, N, T, W.

When the wait number has to be evaluated there are two possibilities;

- (i) if characteristic = 1 use $W = T-L+1 \pmod{32}$
- (ii) otherwise, determine first minor cycle of transfer and use $W = F-m-2 \pmod{32}$.

(1) Source 2, Destination 14, Characteristic 0, NIS.6 (this is the delay line of the next instruction). Timing number $17-8-2 = 7$. As 2 is a DL and 14 only a TS, the first minor cycle of transfer is 12, and the wait number $12-8-2 = 2$.

(2) Source 19, Destination 7, Characteristic 0, NIS.5. (Hereafter it will usually be assumed that these are easy.) Timing number $29-17-2 = 10$. As 19 is a DS and 7 a DL, it is the suffix 25 to the DL which gives F; so $W = 25-17-2 = 6$.

(3) $T = 12-29-2 = -19+32 = 13 \pmod{32}$. 18 is a QS, 12 is a DL so $F = 12$ and $W = 12-29-2 = -19+32 = 13 \pmod{32}$. (The phrase $\pmod{32}$ will be omitted in future.)

- (4) $T = 14-12-2 = 0$. As both 13 and 15 are TS's this can be obeyed in any minor cycle; conventionally we use the first possible minor cycle and put $W = 0$.
- (5) $N = 0$ as the next instruction is in DL 8, $T = 17-14-2 = 1$. 20 is a DS; D 26 behaves like a TS (see Appendix 1); so we use the suffix 3, meaning "any odd minor cycle". Conventionally we use the first available odd minor cycle so $F = 17$, $W = 17-14-2 = 1$.
- (6) $T = 22-17-2 = 3$. S.O behaves like a TS (see Appendix 1); 17 is a QS; so, using Appendix 1, $F = 21$ and $W = 21-17-2 = 2$. As the instruction is a stopper, X is written after the timing number in the detailed coding. When the instruction is being punched this indicates that the most significant digit is not punched.
- (7) $T = 29-22-2 = 5$. $F = 24$, so $W = 24-22-2 = 0$.
- (8) $T = 8-29-2 = -23+32 = 9$, because the next instruction is taken as 5₈, the one on the off side of the discrimination. $F = 31$, so $W = 31-29-2 = 0$.
- (9) $Ch = 2$. $T = 1-9-2 = -10+32 = 22$. $W = 6-9-2 = -5+32 = 27$. This happens to have $W > T$. In no way does this affect the coding; all that occurs is that the machine waits of its own accord for an extra major cycle before picking up the next instruction.
- (10) $Ch = 1$. $T = 9-1-2 = 6$. $F = 7$, so $W = 7-1-2 = 4$. We can check that the length of transfer $L = 6-4+1 = 3$. Notice that the instruction 6₇₋₉ - 17_{3,0,1} (3 m.c.) can be coded only when the next instruction is also in minor cycle 9.
- (11) $Ch = 2$. $T = 16-9-2 = 5$. Since this instruction can be obeyed in any pair of successive minor cycles we conventionally put $W = 0$.
- (12) $Ch = 2$. $T = 24-16-2 = 6$. 21 is a DS, 17 is a QS, so using Appendix 1 $F = 20$ and $W = 20-16-2 = 2$.
- (13) $Ch = 1$. $T = 3-24-2 = -23+32 = 9$. The instruction 17 - 18 (4 m.c.) could be obeyed in any four successive minor cycles, but because $Ch = 1$, the last minor cycle of transfer must be the minor cycle of the next instruction. (If the effect of characteristic 3 is changed this will no longer be necessary.) We could count backwards from m.c. 3 and so find $F = 0$, but a better method is to use $W = T-L+1 = 9-4+1 = 6$.
- (14) $Ch = 1$. $T = 7-3-2 = 2$. $W = 2-32+1 = -29+32 = 3$. Transfers for 32 minor cycles are among the commonest long transfers; it is easy to see that $W = T+1$ in such cases.
- (15) $Ch = 1$. $T = 11-7-2 = 2$. $W = 2-5+1 = -2+32 = 30$. This is an example of bad allocation of storage; if the next instruction were in minor cycle 13 or a little later we should have $W < T$ and a major cycle would not be wasted.
- (16) $Ch = 1$. $T = 31-11-2 = 18$. The meaning of this instruction is explained fully in I.10. What needs to be said here is that D 22 behaves like a DS; the letters "e.o." indicate that the instruction must start to be obeyed in an even minor cycle and must end in an odd one. Of course, because $Ch = 1$ the next instruction must be in an odd minor cycle. $W = 18-4+1 = 15$.
- (17) $Ch = 1$. $T = 0-31-2 = -33+2.32 = 31$. $W = 31-32+1 = 0$. This is the most economical way of arranging a transfer for 32 minor cycles.

This instruction starts being obeyed in m.c.1; in the following minor cycle 0 the last word is transferred from DL 11 to DL 8, and, simultaneously the next instruction is taken from DL 8. This next instruction will be the one just going into 8₀, not the one which is just about to be wiped out.

Checking

The reader who has worked through the above example is likely to have concluded that detailed coding is a mechanical process fraught with error. He should then have concluded that the process must be checked, or better still, avoided altogether.

In II.3 there is a description of a DEUCE program which is used to check detailed coding done by hand; if this program is not used, the length of time spent removing all the errors in one's program is usually at least doubled.

However, the "Detailed Coding Program", described in IV.1, is the best solution to this problem. For this the instructions are punched in decimal (not binary) in the order in which they occur in the flow diagram; and only what is written on the flow diagram is punched - the user has to do no arithmetic. The detailed coding program does all the arithmetic, including all the conversion from decimal to binary, and punches out the complete program in binary in the form in which the user would have to punch his detailed coding.

The reasons why the use of this program is recommended in general, but not to beginners, are two: the beginner must know the form of the instruction inside the machine instinctively if he is ever to test a program, and also if he is going to make up a program in which instructions are modified.

I.7 Subroutines

Much time and effort is wasted if the programmer has to start every programming job from scratch. It is obviously desirable to be able to draw on the results of the experience of others. In the first place, if we can incorporate a tested program in our own there is less chance of error. Secondly, the complete program can be written more quickly.

These ideas lead to the concept of a 'subroutine'. This is a small program to do a specific job, such as evaluating $\sin x$, find a square root, read or punch a decimal number. A fairly large library of subroutines, with full instructions for their use, is available for DEUCE. A selection of recommended subroutines is given in III.3.

To use a subroutine in a program of our own we must know where the routine expects to find its data, which stores it uses as working space, how much space it occupies in the high speed store, where its first instruction is and where we wish to re-enter our own program.

The last of these requirements is the only one that presents any difficulty. One possible way is to leave the NIS. and timing number blank on the last instruction of the library copy of the subroutine. In constructing the program we should have to fill in these blanks to lead to the rest of the program. The trouble with this method is that we need several copies of the subroutine if we wish to perform the same operation several times at different stages in the program. A very simple and elegant method is to plant an instruction at the end of the subroutine every time we use it. This means that we need only one copy of the subroutine and one extra instruction for every entry to the subroutine. Such an instruction is

called a 'link'. In practice, we arrange to place the link in a specified store, usually a TS, and the subroutine itself plants the link as its last instruction.

As we can do only one thing at a time, we can effect a further economy of space by making every subroutine plant its link in the same place. The fact that it is overwritten as soon as we enter another subroutine does not matter, as it has already served its purpose. The standard minor cycle that we use is 130. This has the advantage that wait and timing numbers are now the numbers of the minor cycles referred to.

Example

Read a positive number K from a card. Then read a succession of numbers a_i and find the least n such that

$$\sum_{i=1}^n |a_i|^{\frac{1}{2}} > K$$

Punch out this n and also the sum.

The steps would be as follows:-

- (1) Read K
- (2) Read a_i
- (3) Find $|a_i|$
- (4) Find $|a_i|^{\frac{1}{2}}$
- (5) Add $|a_i|^{\frac{1}{2}}$ to previous sum, i.e. $S' = |a_i|^{\frac{1}{2}} + S$
- (6) Is $S' > K$? If no, increase i and go to 2, otherwise go on
- (7) Punch out S
- (8) Punch out n
- (9) Stop the machine. (Or better: return to 1, for a fresh K .)

In the list of recommended subroutines we find R01/3 (No. 228) which reads one decimal number, P01 (No. 12) which punches one number in decimal and F04/1 (No. 136) which will find a square root. We shall use R01/3 in DL 2, P01 in DL 3, F04/1 in DL 4. Our complete program would be as given below, page 36.

We use 19₂ for storing $\sum |a_i|^{\frac{1}{2}}$, 19₃ for counting the number of terms, 20₂ for storing $+K$. We shall put $-K$ into 19₂ at the beginning and add successive values of $|a_i|^{\frac{1}{2}}$ to this. When a change of sign occurs we have read enough terms. As the punch routine will deal only with fractions, we calculate $n \cdot 10^{-9}$, i.e. instead of adding P1's we store the number 10^{-9} to 30 b.p. and add this.

30 - 19 (2 m.c.)

Clear 19₂, 19₃

Link to 13

2₂₈

R 01/3

Reak K.

1₃₀

30 - 13

21₂ - 26

+K to 20₂, -K to 19₂

21₂ - 20₂

13 - 19

(2) - 13

Link to 13

2₂₈

R 01/3

Read a₁

1₃₀

30 - 13

21₂ - 27

21₂

13

21₂ - 26

(3) - 14

|a₁| to TS 13

Link to TS 14.

4₂₈

F 04/1

|a₁| already in TS 13, as required

1₃₀

19₃ - 13

Increase count.

(a) - 25

(a) = 10⁻⁹

13 - 19₃

14 - 13

19₂ - 25

13 - 27

20₂ - 25

13 - 14

(4) - 13

|a₁|^{1/2} to TS 13

Add previous sum.

Is S - K > 0? If no, store S - K in 19₂ and read next card.

If yes, add K, to give S.

S to TS 14

Link to TS 13

3₂₈

P 01

Punch S

1₃₀

19₃ - 14

Count to TS 14

(5) - 13

Link to TS 13

3₂₈

P 01

Punch count

1₃₀

30 - 19 (2 m.c.)

Clear DS 19 and return to beginning.

up arrow
not
connected

The complete program, the 'master routine', that we have had to write takes 25 instructions and can be placed in DL 1.

This program requires 5 links, which have been denoted by (1), ..., (5), and the number 10^{-9} , which has been denoted by (a). The links themselves would be the instructions.

- (1): 1, 30 - 13
- (2): 1, 30 - 13
- (3): 1, 19₃ - 13
- (4): 1, 19₃ - 14
- (5): 1, 30 - 19 (2 m.c.),

with appropriate wait and timing numbers in each case. In the actual program, of course, (1), ..., (5) would be replaced by their correct addresses, so that the program starts

30 - 19 (2 m.c.)
 1 - 13

 2_{28} R 01/3
 1_{30} : 30 - 13
 ...

It is usual practice to write the entry point to the subroutine on the left and to write the address of the link in brackets by the actual instruction. The above might therefore be written

30 - 19 (2 m.c.)
 17 - 13

 2_{28} R 01/3
 1_{30} : (17) 30 - 13
 ...

if the link is stored in 17. It is obeyed as if it is stored in minor cycle 30 and the wait and timing numbers computed accordingly. One of the first instructions of the read subroutine is

$13 - 1_{30}$,

which plants the link in 1₃₀. The TS where it was previously kept is now available for other purposes. In general the link is no longer in this TS at the end of the subroutine.

Ocasions arise when one subroutine requires the use of another inside it. Such a subroutine is called a 'second order subroutine'. It cannot place its link in 1_{30} because this would be overwritten by the link of the inner subroutine (which can be used elsewhere in its own right). We therefore plant the link in the next available place, in 1_{31} . This idea can naturally be extended to higher order subroutines. Generally, an m^{th} order subroutine uses at least one $(m-1)$ order subroutine in it and plants its own link in 1_{29+m} .

Although DEUCE has a multiplier and divider we often use subroutines to perform these operations. A very useful one is A 12/1 (subroutine No. 222), which has three entry points, for multiplication, division and summation of series respectively. In all cases it is first order?

I.8 Modification of instructions and counting

The situation often arises that the same arithmetical operations are to be performed on a sequence of numbers. Examples are the summation of series and matrix multiplication. There are now two important requirements, to select the numbers in turn and to count whether the operation has been done the desired number of times. Selection of numbers can be done in two ways - either there is a separate selection instruction for each number of the sequence or we add some constant to the address of a basic selection instruction to make it refer to different numbers in turn. The latter technique is preferable, in general, because it saves much space and is very flexible. Note, however, that it may take longer than the other way.

The technique of instruction modification is very important and we shall now examine it in some detail.

Suppose, for example, that we wish to evaluate the power series

$$S(x) = a_0 + a_1x + a_2x^2 + \dots + a_nx^n$$

where the coefficients are stored in DL A. For simplicity, let us assume they are in m.c.'s 0, ..., n and that $n < 32$. A convenient economical method (economical in the number of operations required) is that of nested multiplication, i.e.

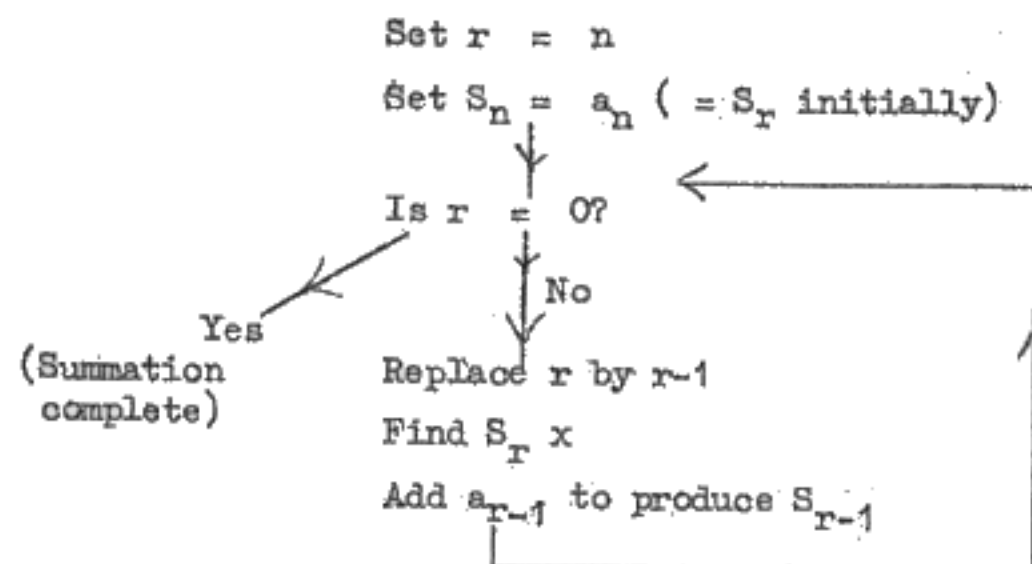
$$S(x) = (\dots ((a_nx + a_{n-1})x + a_{n-2})x + \dots)x + a_0 ,$$

and we use this. If we define

$$S_n = a_n \text{ and } S_{r-1} = S_r x + a_{r-1} \text{ for } r = n, n-1, \dots, 1,$$

it is evident that $S(x) = S_0$.

The block flow diagram of the summation would be



Here we require the coefficients, a_r , in turn. If we have the instruction

$$A_0 - 14, \quad (\text{which picks out } a_0)$$

the addition of r.P17 (i.e. r is the wait number) will change this to

$$A_r - 14,$$

to fetch a_r .

This suggests that we can use r.P17 for two purposes: to modify a selection instruction and to count the number of times we perform the operation. In the above example we must leave the loop when r has become zero, which occurs when we have added a_0 . There is also the implication that it may be advisable to store the basic instruction and add the modification, rather than storing the modified instruction. This is not entirely borne out in practice, (especially on automatic modification - to be described in I.9), as several later examples will show.

In the example, the modification would be done like this:

$$(I) - 13$$

$$(r) - 25$$

$$13 - B_m$$

$$B_m: A_r - 14$$

where (I) and (r) as sources are abbreviations for the addresses where I and r are stored. This method needs a minor cycle^B of a next instruction DL to store the modified instruction. There is, however, a facility on DEUCE for avoiding this - Destination 0.

Destination 0 is TS Count, which has been mentioned earlier. What is sent to TS Count is normally determined by the NIS, and timing number - it is the place in the machine where an instruction is interpreted and executed. By sending an instruction to DO in the minor cycle when the next instruction is selected we can override the natural NIS, and obey this instruction instead. As an example, the above would become:

$$(I) - 13$$

$$(r) - 25$$

$$13 - 0_m$$

$$Q_m [A_r - 14]$$

The Q stands for 'quasi' and we write this to show that the instruction is obeyed as if it stood in minor cycle m. Frequently, as here, we write the minor cycle number with the Destination 0 as well. As with links, it is common practice to write

$$Q_m [(B_s) : A_r - 14]$$

if the basic instruction is stored in B_s . Normally we write the instruction with square brackets.

To use Destination 0 for modification it is essential that the transfer is still going on in the selection minor cycle of the next

instruction. This is ensured by having $W = T$ and $Ch = 0$, or by having $Ch = 1$. Normally the NIS. is irrelevant in a transfer to DO and we leave it blank*.

Many programs use the instruction

0, 13 - 0, 0, 0, 0

in 1_{28} , with the next instruction Q_{30} . This is one of the most useful places for a modified instruction, as W is then the actual minor cycle number. The sequence

$1_{26} : 14 - 25$

$1_{28} : 13 - 0_{30}$

$Q_{30} : [\quad]$

is very common. The modifying constant is placed in TS 14.

This method of modification has one grave disadvantage, in that one of the few places in the DEUCE where addition is possible is required. Sometimes this may not matter, on other occasions it necessitates many 'red tape' instructions to store the contents of TS 13 in a safe place. To overcome this difficulty to a certain extent, the two quadruple stores, QS 17 and QS 18, behave quite differently in instructions sending information to DO. For example, the instruction

17 - 0

adds a particular constant (depending on the NIS.) to the instruction before obeying it and stores the modified instruction. This method of automatic instruction modification (abbreviated to A.I.M.) is described in I.9.

A feature of automatic computation is the loop of instructions, repeated until some particular criterion is satisfied. Most of the programs given as examples up to now have made use of this idea. It is obviously important to traverse the loop the correct number of times, which leads to the subject of counting.

Let it be said at once that this is where a large proportion of mistakes arise. It is only too easy to go round a loop one time too many or too few. It is generally best to count down. If the loop is to be done n times, we start with n in some convenient store and reduce this by 1 every time. When the count has reached zero the loop has been obeyed the correct number of times and is left. Great care must be taken, though, to see that the orders are executed as

Reduce Count

Test Count

not as

Test Count

Reduce Count.

(In the latter case the counting number must be $(n-1)$ at the start.) When tested the counting number in 'counting down' is always equal to the number of times that the loop must still be done.

* But see I.9 for cases when the NIS. is important, in automatic modification.

Counting can be done anywhere in the word. Convenient places are the P1 and P17 position, because P1 and P17 are available as special sources. For automatic counting (I.9) the P5, P10 and P18 positions may also be used. Just where we count depends on the problem. In the example given previously the count is $\times P17$, because it can then be used for modifying an instruction. If, on the other hand, we are merely counting how many times something has happened there is no reason for not using P1.

It is, of course, possible to count upwards from zero. If the loop is to be done n times we increase the count, r , by one each time and test either

$$r - n \text{ or } n - r.$$

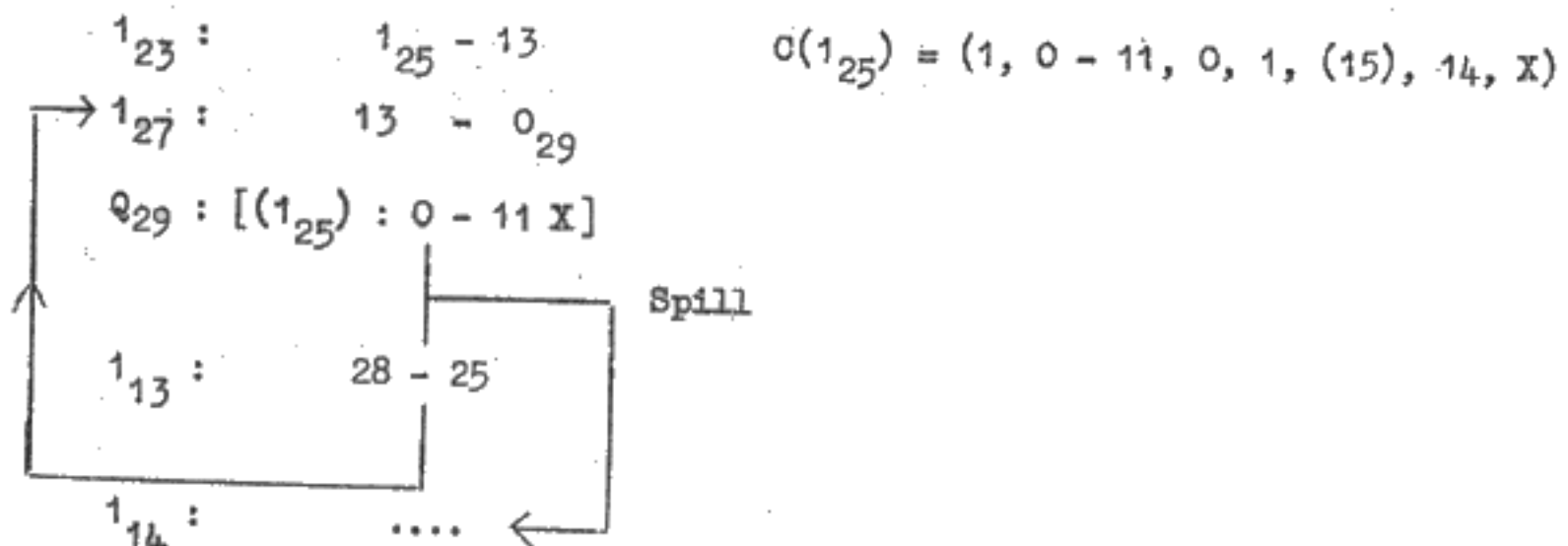
In this case r is the number of times we have already been round the loop. The comparison of n and r may be done simply by using the 'not equivalent' relation, described in I.12. An alternative method is to start with $-n$ and count upwards until a change of sign occurs.

In all these methods n is a parameter that varies from program to program. Quite often, however, the case arises when a loop has always to be done the same number of times. An example is the digit-by-digit method for finding square roots. The loop is done the same number of times, irrespective of the size of the number involved. In such cases we can sometimes use a slightly different technique. If the loop is to be done 32 times we can use a marker digit (a 'strobe'), which starts at P1 and is shifted up by one place every time. When the strobe disappears we have been round the loop 32 times. This method is used in an example in I.14.

A very elegant method that has the advantage that no discrimination is needed to escape from the loop is the 'spill-over'. This uses the 'Joe digits' (P22 - 25) in the instruction word. These digits are ineffective in an instruction. They can, however, be used to propagate a carry from the wait number to the timing number, thereby changing the selection minor cycle of the next instruction. The 'Joe digits' are normally written in brackets between the wait and timing numbers.

The following is an example of the spill-over technique. It comes from an input program.

Example 1



The effect is to obey the Q 29 instruction in such a way that we fill m.c.'s 0, 1, 2, ... in turn. When the instruction fills m.c. 30 it is

$$1, 0 - 11, 0, 31, (15), 14, X$$

which is then modified to

1, 0 - 11, 0, 0, (0), 15, X ;

this fills 11₃₁ and takes its next instruction one m.c. later than before. Note that the instruction is obeyed Q₂₉. Escape from the loop occurs when the wait number has become zero - if the last instruction is to refer to m.c.31, with a wait number of 0, it must be in m.c.29.

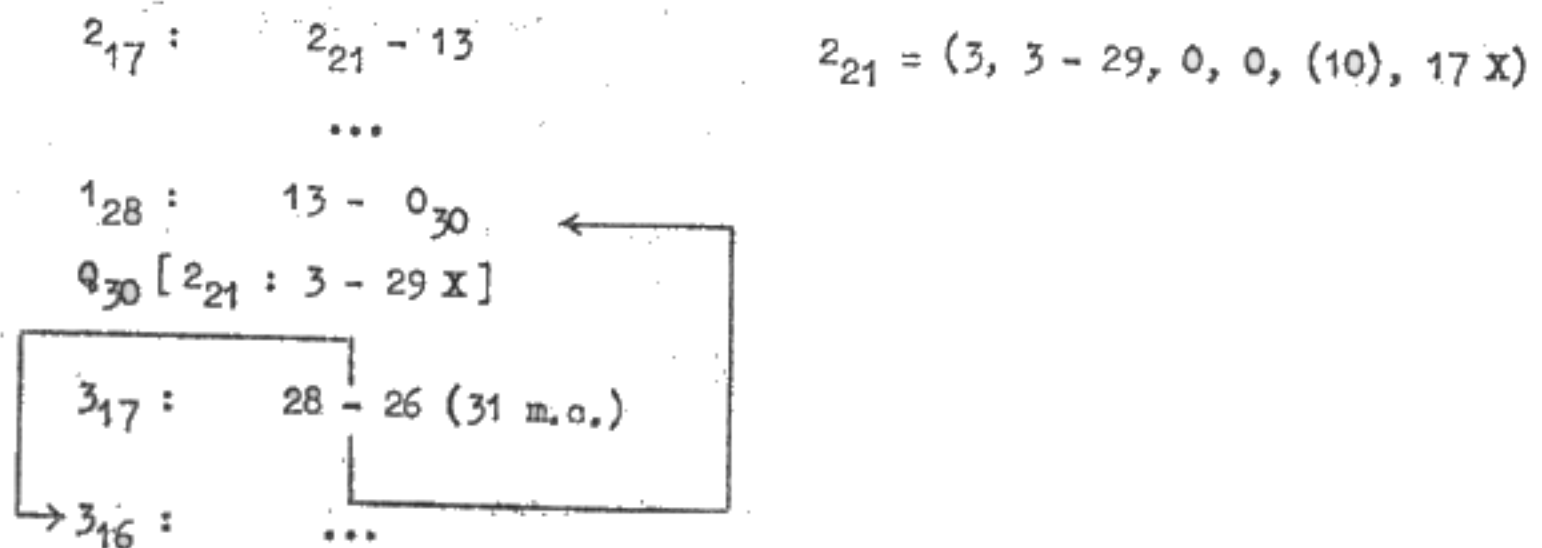
It is often useful to have a constant like

P17 and P22

available as a modifier. In this way we can obtain a small count in the Joe digits. This can also be done automatically, as described in I.9.

It is possible to use the spill over technique backwards, by subtracting from the wait number. If the Joe digits are zero the timing number is reduced by one when the wait number 'goes negative'. (In case the timing number starts at zero it will be necessary to put a P31 into the instruction, to avoid disturbing the go digit.) A further refinement of this method is to add one to the wait number at the same time as decreasing the Joe digits by one: this can be achieved by subtracting 31.P17 from the instruction.

Example 2



This comes from a punch subroutine: the loop is executed twelve times.

A further useful technique of instruction modification occurs when we have two instructions referring to the same minor cycle, one to fetch and one to store.

Example 3

In the same punch subroutine the instruction

Q₃₀ [3_m - 15]

is followed shortly after by

Q₃₀ [26 - 3_m]

TS 13 is not disturbed between these instructions. Here we add a suitable constant to

to produce

3 - 15

26 - 3

The actual instructions are:

$$1_{28} : 13 - 0_{30}$$

$$Q_{30} [3_m - 15]$$

$$2_{12} \dots$$

.....

$$2_{25} : 2_{27} - 25$$

$$1_{28} : 13 - 0_{30}$$

$$Q_{30} [26 - 3_m]$$

$$3_{12} : \dots$$

2_{27} is the 'pseudo-instruction'

$$1, 23 - 20, 3, 31, (15), 31, (1),$$

where the (1) represents the P31 digit.

In all the examples given the modified instruction has been obeyed with a.

$$13 - 0$$

instruction. This is not essential and many similar instructions occur, such as

$$14 - 0,$$

$$21 - 0, \text{ etc.}$$

Note that with $21 - 0$ we must specify which minor cycle is sent to D0.

In the whole of this section the modification and counting has been done by program. Originally, this was the only method available on DEUCE. The next section describes certain alterations that were made, late in 1957, to the instructions

$$17 - 0 \text{ and } 18 - 0$$

to allow a certain amount of automatic modification and counting.

I.9 Automatic modification and counting

The technique of modification, described in the preceding section, is probably the most important single advance in the recent history of computing devices. It is from this that much of the power of the modern high speed computer derives. When the Pilot ACE was first constructed no facilities were included for the automatic modification of instructions and this course was also followed by the English Electric Co. in the first models of the DEUCE. After such machines had been in operation for several months it was felt that modification was of such frequent occurrence and general use that extra facilities would be helpful. These would result in a general speeding up of programs because the (restricted) addition facilities in DEUCE would be left free for arithmetic and would not be needed for modified instructions.

Few programs in the DEUCE library used the instructions 17 - 0 and 18 - 0* and accordingly these instructions were chosen for engineering changes to the computer, to give some of the more commonly required modification effects. These alterations were made to the DEUCE late in 1957. The new facility is called 'Automatic Instruction Modification', abbreviated to A.I.M.

We have seen in I.8 that a frequent requirement in modification is the addition of a P17. When numbers are to be used in pairs, e.g. in double-length arithmetic (I.11), addition of a P18 is needed. Other constants that are often needed are

$$P\ 17 + P22 \text{ or } P18 + P22,$$

to achieve a small count in the Joe digits. To enable the DL's to be used as a continuously numbered store (Example 2 below) it is often desirable to have P5 and P10 available for modification. In I.16 we shall find a use for the combination $P5 + P22$, for transfers to and from successive tracks on the drum. All of these constants are available with A.I.M.

Usually the NIS. digits are irrelevant in an instruction sent to D0 and for this reason they are available for use in a sense different from the normal one. This is what has been done with A.I.M. Different combinations of NIS. digits have the effect of adding different modifying constants. The P1 digit, which is not used by other instructions, has also been passed into service, as the auxiliary staticizer 'A', to widen the scope of A.I.M. by giving a greater variety of additive constants. It is also possible to subtract the modifiers instead of adding them.

One extra facility supplied by the A.I.M. alteration is automatic counting. For this, all of the previous paraphernalia is used in conjunction with the characteristic. We know that it is only when a transfer to D0 happens in the m.c. when the next instruction is due to enter TS Count that we override the normal selection of instructions. A.I.M. uses this by adding a constant to QS17 or QS18 if the

$$17 - 0 \text{ or } 18 - 0$$

instruction is for a single minor cycle which is not the selection m.c. of the next instruction**.

Up to now nothing has been said to imply that QS17 and QS18 behave differently with A.I.M. Actually they do, in that additions in QS17 become subtractions in QS18 and vice versa. This means that QS18 is normally used for counting down, QS17 for counting up.

The table below gives the exact values of the modifying constants that are used by A.I.M. under all possible combinations of controlling digits. We use the symbol 'Y' for the four digits P1, P2, P3, P4.

* Programs of one of the authors' were exceptions!

** In normal circumstances such a transfer to D0 does nothing.

Y	17 - 0, 0	17 - 0, 1
0	+ P5	+ P5
1	+ P10	+ P10
2	+ P17	+ P17
3	+ P18	+ P18
4	+ P5	+ P5 + P22
5	+ P10	+ P10 + P22
6	+ P17	+ P17 + P22
7	+ P18	+ P18 + P22
8	+ P5	- P5
9	+ P10	- P10
10	+ P17	- P17
11	+ P18	- P18
12	+ P5	- P5 - P22
13	+ P10	- P10 - P22
14	+ P17	- P17 - P22
15	+ P18	- P18 - P22

For QS18 all signs are reversed.

The lower of the digits to be added is determined solely by the P1, P2 digits, as is obvious from the table. If Ch = 0, i.e. no P15, the other digits of Y have no effect. On the other hand, if a P15 is present, (Ch = 1), the presence of a P3 adds a P22 to the modifying constant and a P4 reverses the effect of the modification.

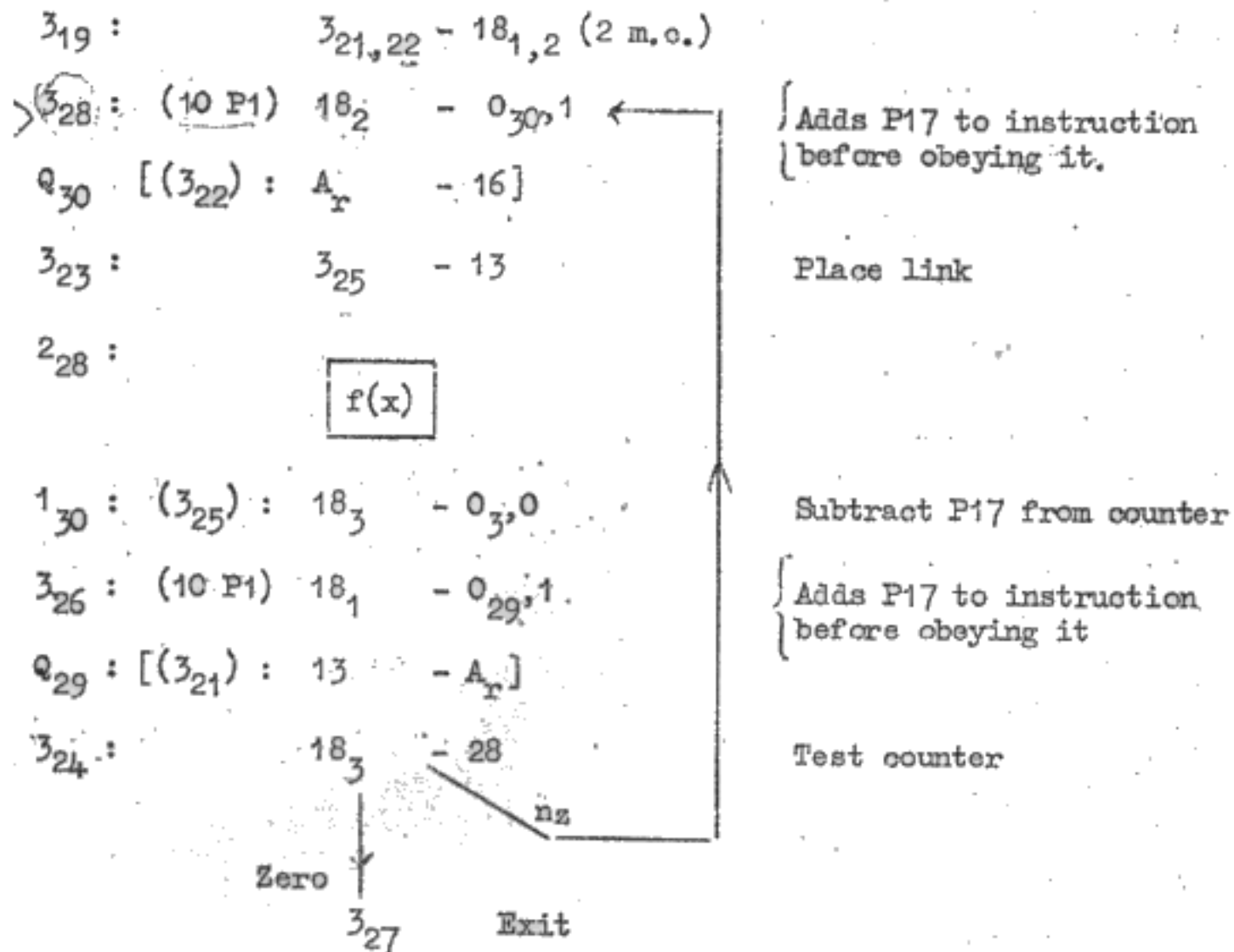
The modifying constant is added only in the first minor cycle of the transfer to DO. It is the modified instruction that is obeyed and also stored in the appropriate QS. When the modifier contains a P22 there is no carry to this position from the lower order digits. Carry can occur between successive minor cycles.

The case of Ch = 0 is used for automatic counting. Here the NIS. will have its normal effect (because the transfer has ceased before the next instruction is presented to TS Count) and so there is a restriction - either we count in the place determined by the NIS. or, if we wish to count in a particular digit position, the next instruction must be in a certain position. Counts on P5 and P10 are possible only with an even numbered NIS., while those on P17 and P18 require an odd numbered NIS.

The remainder of this section is composed of simple examples using A.I.M.

It should be noticed that the A.I.M. facility is the only one in the machine where the number in the source is altered on being sent to a destination that is not related to it.

The first example of I.8 can be shortened as follows using A.I.M.



The contents of 3_{21} , 3_{22} and 3_{25} are

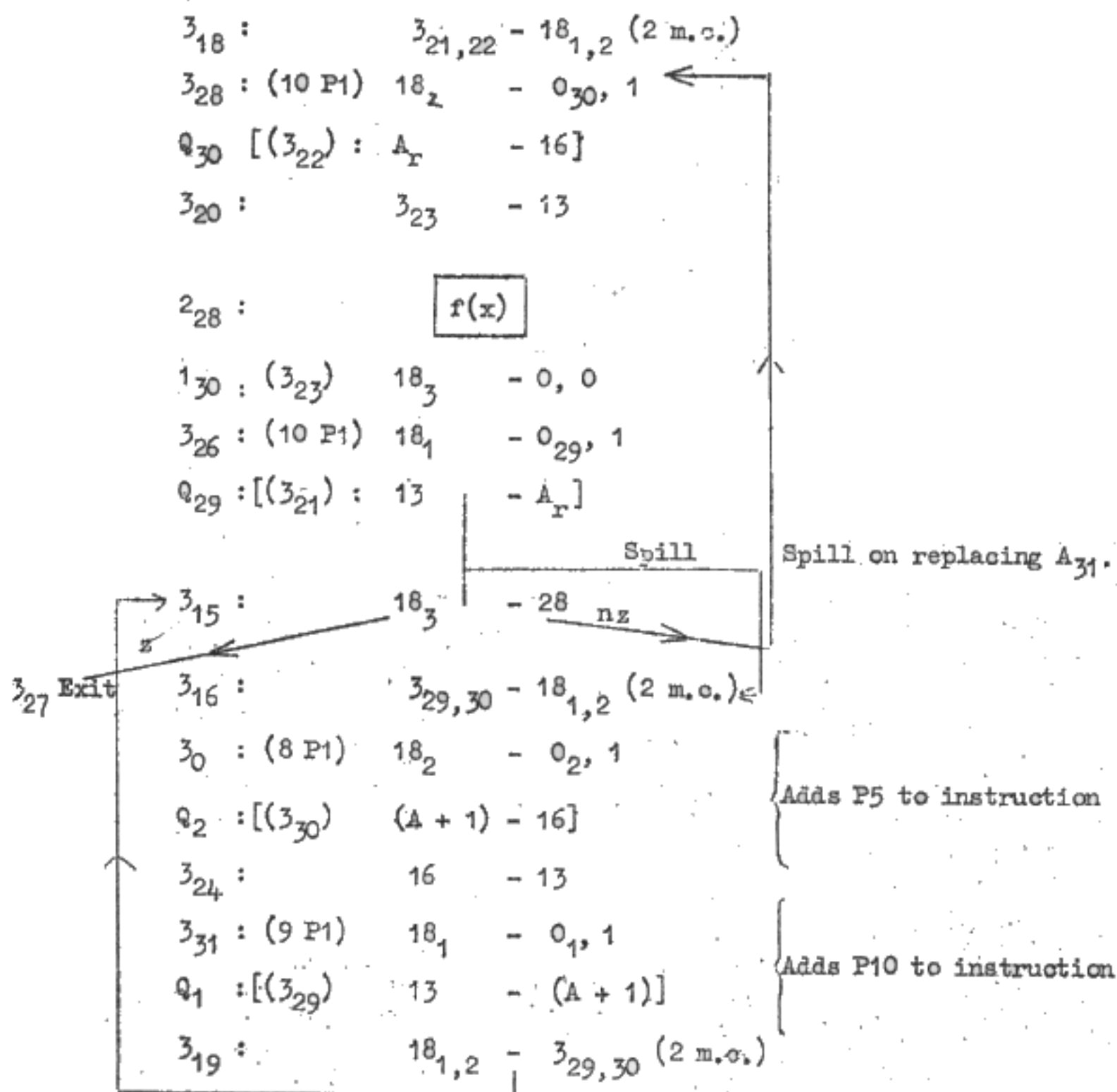
$$\begin{aligned}
 C(3_{21}) &= (3, 13 - A, 0, 0, (0), 25) \text{ obeyed } Q_{29} \\
 C(3_{22}) &= (3, A - 16, 0, 31, (0), 23) \text{ obeyed } Q_{30} \\
 C(3_{25}) &= (3, 18 - 0, 0, 3, (0), 26) \text{ obeyed } Q_{30}, \text{ used as link.}
 \end{aligned}$$

There are some points to notice.

- (i) The two instructions that perform modifications have $Y = 10$, to add P17. ($Y = 10$, not 2, because the normal modification in QS 18 is subtracted.)
- (ii) The fetch instruction, in 3_{22} , and the store instruction, in 3_{21} , are coded to have a wait number 1 less than is required to pick out the first number, because the first modification occurs before any numbers have been used.
- (iii) The link instruction does the counting. We are able to count X P17, as the NIS. = 3. The P17 is subtracted in m.o.3.
- (iv) The master routine has been coded to be placed at one end of DL 3, it uses 3_{19} and 3_{21-28} only.

Example 4. The same as Example 3 without the restriction on n . A possible program follows. Again, n is in 18_3 at the start.

/Example



where $C(3_{21}) = (3, 13 - A, 0, 0, (15), 16)$ obeyed Q_{29}
 $C(3_{22}) = (3, A - 16, 0, 31, (0), 20)$ obeyed Q_{30}
 $C(3_{23}) = (3, 18 - 0, 0, 3, (0), 26)$ Link; obeyed in 1_{30}
 $C(3_{29}) = (3, 13 - A, 0, 0, (15), 16)$ obeyed Q_1
 $C(3_{30}) = (3, A - 16, 0, 31, (0), 20)$ obeyed Q_2

NOTES: (i) As far as the test instruction in 3_{15} , everything is as In Example 3, (except for the placing of some instructions).

(ii) The store instruction, in 3_{21} , has 15 in the Joe digits, to make use of spill-over after A_{31} has been replaced by $f(x)$.

(iii) The instructions in $3_{29,30}$ at start are the same as those In $3_{21,22}$. They will be changed to

$13 - (A + 1)$ and $(A + 1) - 16$

respectively and then replaced in $3_{29,30}$. In this way, if there are more

than 64 numbers, the instructions will later become

$$13 - (A + 2) \text{ and } (A + 2) - 16,$$

etc.

(iv) The instruction

$$16 - 13$$

in 3_{24} is necessary because otherwise a wrong number will be placed in $A + 1$. The purpose of the two modified instructions is to add P5 and P10 to them respectively. The instruction in 3_{24} avoids any harmful effects. If the subroutine for $f(x)$ required x in TS 13, instead of TS 16, this instruction would not be necessary.

(v) The coding of the instructions following 3_{16} . It is essential that the two $18 - 0$ instructions are obeyed in consecutive minor cycles, as here, since this is the way they are obeyed in the main loop. If this is not done the number placed in DL ($A + 1$) by the second instruction does not go back where it came from. Just when the two instructions are obeyed is fairly immaterial but they dictate the placing of the instructions in 3_{24} and 3_{19} - these must follow from the modified instructions with timing numbers 20 and 16 respectively.

To satisfy all these requirements can be something in the nature of a jig-saw puzzle. As in Example 3 the instructions have been placed quite compactly. Wherever possible the delay between instructions has been kept small.

I.10 Optimum coding

The use of wait and timing numbers to control the actual execution of an instruction and selection of its successor was explained in I.6. The wait number is often non-zero, because it is not always possible to place the required numbers two minor cycles away from the instruction calling for them. This means that the computer has to wait until the minor cycles asked for are available. It is obvious that this wasted waiting time would be reduced if data were so disposed in the high-speed store that numbers became available with as small a delay as possible. Naturally, this requires very careful planning.

Again, the time the computer waits between instructions depends on their relative placing. Generally speaking, if the timing numbers are small the wasted time is small. This implies that it is necessary to place instructions in advantageous positions relative to each other if the full speed of the computer is to be realised. In I.6 we saw the consequences of having a timing number that was less than the wait number - the DEUCE waits a whole major cycle before the next instruction can enter TS Count. A loss of 1 millisecond may not seem vital, but if this is done with every instruction the speed of the machine is reduced from a maximum of about 16,000 operations per second (when $W = T = 0$ in every instruction) to about 500 operations per second ($W = 31, T = 30$ everywhere). This is, of course, an extreme case but it does illustrate what could happen with completely thoughtless coding of instructions.

It is therefore reasonable to lay down as a principle that wherever possible instructions should be coded in such a way that the waiting time until actual execution is small (i.e. W small) and that the next instruction emerges from its DL as soon as possible after the end of the operation.

This procedure is known as 'optimum coding'. In general, if it is necessary to deal with large quantities of numbers it is not possible to optimize both the instructions and the data without a completely uneconomic amount of effort by the programmer.

Usually, the optimum coding is done only for instructions. Here, also, the amount of time spent on the actual allocation of storage should be commensurate with the expected use of the program. The places where the technique pays the highest dividends are loops that are repeated many times. For example, suppose that at the first attempt at coding a loop it is found that 3 major cycles are necessary but that the last instruction has a high timing number, say $T = 28$. This means that 30 minor cycles are wasted every time round the loop. By reallocation of the positions of some instructions it may be possible to reduce the total time of the loop to 2 major cycles. If this loop is the innermost of a program this saving may be quite significant. It is a matter of experience to recognise when such a saving is possible. If the excess over a whole number of major cycles is small, say 3 or 4 m.c.'s at most, it is almost always possible to save these by recoding. It is certainly worth trying. On the other hand, this loss of time may not matter, that even if the minor cycles could be saved there would be no overall saving in time. As an example, the punch allows 116 major cycles of computing time between successive cards. If the necessary calculation can be done in this time it may be possible to keep the punch running continuously. Shortening the time of the calculation would be no use, as the time saved would merely lengthen the time DEUCE waits for the next card to reach the punching knives. If this hypothetical calculation took just over the 116 major cycles allowed between cards, it would be necessary to stop the punch between cards, thereby halving its speed. In this instance some reprogramming might save considerable time.

When a program is being coded, the normal procedure is to tackle first those sections that are obeyed most frequently. Any saving in machine time that results from optimum programming here is most noticeable. We then work outwards to the less frequent loops, until we reach the master program, when there is generally little need for great rapidity. If it is possible to optimize this as well as all other sections so much the better but this is the least important part - as far as total machine time is concerned. Subroutines are normally optimum coded, as these are frequently used sections of programs.

Optimizing a subroutine may be quite a tricky operation, as there are often conflicting requirements. The routine is to be fast, yet its instructions should be packed into a limited number of DL's, not dotted all over the store. It may even be necessary to pack the instructions compactly into several consecutive m.c.'s of one DL, for example entirely within m.c.'s 16 - 31, so that it may be used in the same DL as another subroutine occupying the other minor cycles.

There are many operations on DEUCE that are autonomous, in the sense that once started the machine is free to perform other duties. These are multiplication and division, reading and punching cards, and transfers of information to and from the magnetic drum. All of these operations are comparatively lengthy. Optimum coding can save much time here. If at all possible the instruction should be coded so that many succeeding instructions can be obeyed before the autonomous one is complete. For example, it is common to anticipate a transfer from the drum to the high speed store, so that the information is already available when required.

The whole art of optimum coding, and the necessity for it, is often held up as a drawback of the DEUCE*. Actually, it is not nearly as bad as is suggested by those accustomed to other computers (see Ref.11). It is not difficult to write programs that are 'nearly optimum' at the first attempt. Like other things it is less fearsome when familiar and familiarity only comes with practice in programming.

I.11 The double-length accumulator

Any digital computer, whether a desk machine working in decimal or a large automatic machine working in binary which multiplies together two numbers with n digits will need a register with $2n$ digits to take the product; and this register will need facilities for addition so that the multiplication can take place. This is the immediate reason why DEUCE has an accumulator, DS 21, which holds 64 binary digits as well as the short accumulator, TS 13; however DS 21 has facilities for subtraction and shifting as well as for addition.

The use of TS 13 for addition and subtraction of numbers was explained in I.3. In this section destinations 22 and 23 and source 22 will be used for addition, subtraction and shifting in DS 21. However the double-length accumulator is not quite as simple to use as the short accumulator, TS 13.

The result of any operation using S.22, D.22 or D.23 may depend on the state of an electronic trigger called TCB: source 21 and destination 21, as already explained, act in the same way as the source and destination of any other mercury store, and their behaviour is unaffected by the state of TCB. TCB can be "off" or "on". Its usual state is off - in fact all subroutines assume that TCB is off when they are entered and they should all leave it off at exit.

TCB is put off by the instruction 4 - 24 and is put on by the instruction 5 - 24. These come in the special class of instructions with destination 24 known as "Destination Triggers", which are fully explained in I.14. All that needs to be said now is that the source numbers 4 and 5 do not refer to the corresponding delay line when the destination is 24. The machine realises this and switches TCB accordingly.

When TCB is off the number is treated as one long 64-bit number, so the operations of addition, subtraction and shifting must cause a carry from the less significant half of the number (in 21_2) into the more significant half (in 21_3). When TCB is on the two halves of DS 21 they are treated as separate 32-bit numbers and there can be no carry from one half to the other. This means that there will be no carry from 21_3 into 21_2 , whatever the state of TCB, and there can only be carry from 21_2 into 21_3 when TCB is off.

Operations in 21_3

The state of TCB does not affect these operations.

Example 1 $9_5 - 21_3$ $C'(21_3) = C(9_5)$, whether TCB is off or on.

Example 2 $9_5 - 22_3$ $C'(21_3) = C(21_3) + C(9_5)$, whether TCB is off or on.

Example 3 $9_5 - 23_3$ $C'(21_3) = C(21_3) - C(9_5)$ whether TCB is off or on.

* It is also used as a selling point in favour of the machine - 'Optimum coding.' has a fine verbal flourish about it.

- Example 4 $22_3 - 21_3$ $C'(21_3) = \frac{1}{2}C(21_3)$, whether TCB is off or on.
This is a shift down exactly like $23 - 14$.
- Example 5 $22_3 - 9_5$ $C'(9_5) = \frac{1}{2}C(21_3)$, whether TCB is off or on.
The numbers in 21_2 and 21_3 are unaffected.
- Example 6 $21_3 - 22_3$ $C'(21_3) = 2C(21_3)$, whether TCB is off or on.
Shifting up in 21 is done by addition, just like $13 - 25$.

Not one of these operations affects the number in 21_2 .

Double-length operations in 21

TCB is off in all the following examples because the numbers in 21 are all double-length.

- Example 7 $9_{4,5} - 21_{2,3}$ $C'(21) = C'(9_4)$, $C'(21_3) = C'(9_5)$.
(2 m.c.) Of course this result is also true if TCB is on.
- Example 8 $9_{4,5} - 22_{2,3}$ $C'(21) = C(21) + C(9_{4,5})$.
(2 m.c.) Double-length addition.
- Example 9 $9_{4,5} - 23_{2,3}$ $C'(21) = C(21) - C(9_{4,5})$.
(2 m.c.) Double-length subtraction.
- Example 10 $22_{2,3} - 9_{4,5}$ $C'(9_{4,5}) = \frac{1}{2}C(21)$.
(2 m.c.) Double-length shift down. This is like S.23; the shift is correctly signed and is truncated. The number in 21 is unaffected.
- Example 11 $22 - 21$ $C'(21) = 2^{-n} C(21)$.
(2n m.c., e.o.) The letters (e.o.) mean "transfer starts in an even minor cycle and ends in an odd one". Of course, $2n < 32$. This is just like $23 - 14$ (n m.c.). Unless $n = 1$ characteristic 1 must be used. The next instruction will then have to be in an odd minor cycle.
- Example 12 $21 - 22$ $C'(21) = 2^n C(21)$. Of course, $2n < 32$. Shifting up is done by addition just like $13 - 25$ (n m.c.). Unless $n = 1$ characteristic 1 must be used. The next instruction will then have to be in an odd minor cycle.
- Example 13 $20 - 22$ $C'(21) = C(21) + C(20)$. As in all of examples 7-12 it is essential that transfer ends in an odd minor cycle. The instruction
(2 m.c., e.o.)
 $20 - 22$
(2 m.c., o.e.) may well give the wrong result. The reason is explained in Example 26.

Operations in 21_2

- Example 14 $9_4 - 21_2$ $C'(21_2) = C(9_4)$, whether TCB is off or on.
Of course, $C'(21_3) = C(21_3)$.
- Example 15 $9_4 - 22_2$ $C'(21_2) = C(21_2) + C(9_4)$, whether TCB is off or on.
But $C'(21_3) = C(21_3)$ if TCB is on,
only may be if TCB is off.
- Example 16 $9_4 - 23_2$ $C'(21_2) = C(21_2) - C(9_4)$, whether TCB is off or on.
But $C'(21_3) = C(21_3)$ if TCB is on,
only may be if TCB is off.
- Example 17 $22_2 - 9_4$ $C'(9_4) = \frac{1}{2}C(21_2)$ if TCB is on,
only may be if TCB is off.
The numbers in 21_2 and 21_3 are unaffected whatever the state of TCB.
- Example 18 $21_2 - 22_2$ $C'(21_2) = 2C(21_2)$, whether TCB is off or on.
But $C'(21_3) = C(21_3)$ if TCB is on,
only may be if TCB is off.

It will be seen that destinations 22 and 23 can always be used for addition and subtraction in 21_2 with TCB on, and they can also be used with TCB off if the programmer does not mind changing the content of 21_3 .

Operations in 21_2 with TCB off must be explained further.

- Example 19 $4 - 24$ TCB off
 $31 - 21$ (2 m.c.) "Ones" in both halves of 21
 $27 - 22_2$ Add P_1 to 21_2 .

TCB is off and the double-length integer -1 is placed in DS 21. The integer 1 is added to the least significant half of 21, so the result ought to be 0 in 21_2 and 21_3 . This means that carry from 21_2 to 21_3 must be allowed.

- Example 20 $4 - 24$ TCB off
 $27 - 21_2$ P_1 to 21_2
 $30 - 21_3$ 0 to 21_3
 $31 - 22_2$ Add "ones" to 21_2 .

TCB is off. This time we place the double-length integer 1 in 21 and add "ones" to 21_2 . If we just allow carry from 21_2 to 21_3 the result is a P_1 in 21_3 , i.e. 2^{32} , whereas we should like the result to be 0. The addition has been $1 + (2^{32} - 1) = 2^{32}$

$$\text{not } 1 + (-1) = 0.$$

DEUCE does get the result 0, but to do this it has to treat the number added as -1, not as $2^{32} - 1$. The method used when adding a number into 21_2 with TCB off is to allow carry from 21_2 into 21_3 and to add a number into 21_3 which is 32 copies of the sign digit of the number added - all zeros or all ones.

When a number is subtracted from 21_2 with TCB off carry is allowed from 21_2 into 21_3 and a number is subtracted from 21_3 - this number is again 32 copies of the sign digit of the number subtracted from 21_2 .

What DEUCE does, in effect, is to extend the single-length number being added or subtracted to double-length by copying the sign digit. In example 19, 10.0...00 is extended by 00...00 so that it is the double-length number 1 which is being added; in example 20, 111....11 is extended by 11...11 so that the double-length number -1 is added and the result is 0.

Example 21 4 - 24 TCB off
 30 - 21 (2 m.c.) 0 in 21_2 and 21_3
 27 - 23_2 Subtract P_1 from 21_2 .

100...00 is extended by 00...00 to become a double-length number and subtracted from 0. The result is the double-length number 11...11, 11...11, i.e. -1.

Example 22 4 - 24 TCB off
 30 - 21 (2 m.c.) 0 in 21_2 and 21_3
 31 - 23_2 Subtract "ones" from 21_2 .

111...11 is extended by 11....11 and the resulting double-length number is subtracted from 0, giving 100..00, 00..00, i.e. +1.

Example 23 4 - 24 TCB off
 29 - 23_2 .

00...001 is extended by 11.....11 and the double-length number is subtracted from the number already in 21 . This is equivalent to adding the double-length number 00..001, 00...00. The instruction appears to subtract a P_{32} from 21_2 , but in fact it adds P_{32} to 21_2 , so rounding off the double-length number in 21 to single-length. This instruction is very common at the end of multiplication subroutines.

Example 24 4 - 24 TCB off
 29 - 22_2 .

The converse effect to the previous example: P_{32} is subtracted from 21_2 .

Example 25 4 - 24 TCB off
 21_2 - 23_2

The second instruction subtracts $0(21_2)$ from (21_2) and leaves in 21_2 . Also, if the number in 21_2 had a P_{32} , $(-P_1)$ is subtracted from 21_3 ;

but if the number in 21_2 had no P_{32} , 0 is subtracted from 21_3 . The total effect is to clear 21_2 and to round off the double-length number in 21 in a single instruction.

Example 26

As already explained,

$$4 - 24$$

$$20 - 22 \text{ (2 m.c., e.o.)}$$

is the correct method of adding $C(20)$ to 21 .

The instructions

$$4 - 24$$

$$20 - 22 \text{ (2 m.c., o.e.)}$$

use destination 22 ending in an even minor cycle; so DEUCE goes through the same procedure of extending the number added in the even minor cycle to double-length. This gives an incorrect result.

The same is also true if destination 23 is used.

Example 27

$$5 - 24 \quad \text{TCB on}$$

$$22_2 - 9_4$$

$\frac{1}{2}C(21_2)$ is put into 9_4 ; neither half of 21 is affected. The shift is a correctly signed arithmetical shift - the P_{31} digit of $C'(9_4)$ is the P_{32} digit of $C(21_2)$ and the P_{32} digit of $C'(9_4)$ is the same - and the P_1 digit of $C(21_2)$ is truncated.

Example 28

$$4 - 24 \quad \text{TCB off}$$

$$22_2 - 9_4$$

The only difference from the previous example is in the P_{32} digit of $C'(9_4)$. It is now a copy of the P_1 digit of $C(21_3)$. It is difficult to see any application of this.

Example 29

$$4 - 24 \quad \text{TCB off}$$

$$\begin{array}{l} 22 - 21 \\ (2n \text{ m.c., e.o.}) \end{array} \quad C'(21) = 2^{-n} C(21). \quad n < 16$$

is the correct method of shifting down n places;

$$4 - 24$$

$$22 - 21 \text{ (2n m.c., o.e.)}$$

is incorrect.

For instance, if $n = 1$ and the digits in 21 are numbered 1, 2, ..., 31, 32; 33, 34, ..., 63, 64,

$$4 - 24$$

$$22 - 21 \text{ (2 m.c., e.o.)}$$

gives the correct result 2, 3, ..., 32, 33; 34, 35, ..., 64, 64; whereas

$$4 - 24$$

$$22 \div 21 \text{ (2 m.c., o.e.)}$$

gives the result 2, 3, ..., 32, 34; 34, 35, ..., 64, 64.

Example 30

The instruction

$$22_3 - 21_3$$

gives a truncated shift of one place, whatever the state of TCB. To get a rounded shift, call the number in 21_3 the integer $a + 2b$, where a is the least significant digit; $a = 0$ or 1 . A rounded shift should give $a + b$; the truncated shift gives b from source 22_3 . The instruction

$$22_3 - 23_3$$

gives $(a + b)$ in 21_3 as required (whatever the state of TCB).

Similarly

$$22 - 23 \text{ (2 m.c., e.o.) with TCB off}$$

gives a rounded shift of the double-length number in 21; and

$$22_2 - 23_2 \text{ with TCB on}$$

gives a rounded shift of the number in 21_2 .

Example 31

$$C(15) = r P_{17} \text{ where } r \leq 32.$$

$\rightarrow 1_3$	$1_{6,7} - 21_{2,3} \text{ (2 m.c.)}$	$[1, 9-16, 0, 0, 27 \text{ and } 1, 16-9, 0, 0, (15) 4]$
1_8	$15 - 22 \text{ (2 m.c.)}$	
1_{28}	$21_2 - 0_{30}$	
$Q_{30} [1_6, 9_r - 16]$		
1_{27}	$21_3 - 0_{29}$	
$Q_{29} [1_7, 16 - 9_{r-1}]$		
Spill over, $r=32$		

1_4

In this example it does not matter whether TCB is off or on, and it does even not matter whether the second instruction is obeyed (e.o.) or

(o.e.) - because the number in 15 is positive. (The fact that the numbers in both halves of 21 are negative does not affect the position.)

I.12 The multiplier and divider

DEUCE is probably the only machine with an automatic multiplier and divider (electronically they are parts of the same unit) in which sub-routines are regularly used for both these operations. Multiplication, in particular, is such a common operation that other machines make the multiplication instruction one which is easy to use, but DEUCE's multiplier operates on positive integers (integers in which the presence of the P_{32} digit means that the number is greater than 2^{31}) although signed fractions are the form numbers usually taken. To make things more illogical, DEUCE's divider does work with signed fractions.

Because subroutines are regularly used for multiplication and division there is no necessity for the beginner to read this section before he can program any useful arithmetic - in fact a multiplication subroutine was first used in I.4. This section is addressed to the programmer who wishes to have a complete knowledge of DEUCE.

The multiplier and divider are started by the instructions 0 - 24 and 1 - 24 respectively. These are two more instructions with destination 24: again it must be emphasised that the machine does not use the normal sources 0 and 1 (I.D. and D.L.1) when it encounters these instructions.

The multiplier

The one time the programmer may want to use the multiplier without recourse to a subroutine is when the product of two positive integers is required.

The two positive integers are placed in 16 and 21_3 , and 21_2 must be cleared*. Then the multiplier is started by the instruction 0 - 24, obeyed in an odd minor cycle. After 65 minor cycles the product, a double-length number, is in 21: the number which was placed in 21_3 has been lost but the number in 16 is still available for use. The result would not be affected if the 0 - 24 had characteristic 1 or 2, nor if another 0 - 24 instruction were obeyed while the multiplication started by the previous one is still running. TOB is put off as the multiplication starts.

If the program is

```

21 31 - 16
23 31 - 213
25 30 - 212
28 0 - 24 (in m.c. 11)

```

the multiplier takes the product of $11\dots 11$ and $11\dots 11$. These are both the positive integer $2^{32} - 1$, so the product obtained is $(2^{32} - 1)^2$, i.e. $100\dots 00, 011\dots 11$.

* If 21_2 contains a reasonably small positive number when multiplication starts this number will be added to the more significant half of the product. If the number in 21_2 is negative, however, the number in 16 is multiplied by a number which is not the content of 21_3 .

If these numbers represent signed integers we should be squaring -1, and the result should be 1, i.e. 100...00,00...00.

If the numbers represent signed fractions with 30 binary places (i.e. with that number of binary digits below the binary point) we should be squaring -2^{-30} and the result should be $+2^{-60}$, i.e. 00100...00,00...00.00.

The rest of this section on the multiplier is concerned with the corrections necessary to get these results. It is also concerned with the question of timing.

In the above example the multiplier was started in m.c.11 (not m.c.10 because this is an even minor cycle). The 65 minor cycles during which the multiplier works run from the beginning of this minor cycle to the end of m.c.11 in the next major cycle but one. The earliest the result can be taken from 21 is therefore m.c.12 and 13 of this major cycle.

While the multiplier is working destinations 16, 21, 22 and 23 must not be used or an incorrect product will be given. Source 21 will give the partly-formed product during this period and so will source 22: using them will not affect the result of the multiplication, but it is difficult to see any application except the one given below. Source 16 can be used freely during the multiplication process. The second reason for using a subroutine for multiplication is to waste sufficient time for the multiplication to have ended before the master routine can affect the result.

Multiplication of signed integers

a and b are signed integers, say in 14 and 16. The subroutine to multiply them will start

30 - 21₂

14 - 21₃

0 - 24 in an odd m.c.

If both a and b are positive this gets the correct answer ab.

If a is negative the multiplier treats it as $(2^{32} + a)$ because it takes the presence of the P₃₂ digit in a to mean that $a > 2^{31}$, not that a is negative. So if a is negative and b is positive the multiplier produces $(2^{32} + a)b$, and the subroutine must subtract $2^{32}b$ from the result.

Conversely, if a is positive and b negative, the multiplier produces $a(2^{32} + b)$ and the subroutine must subtract $2^{32}a$ from the result.

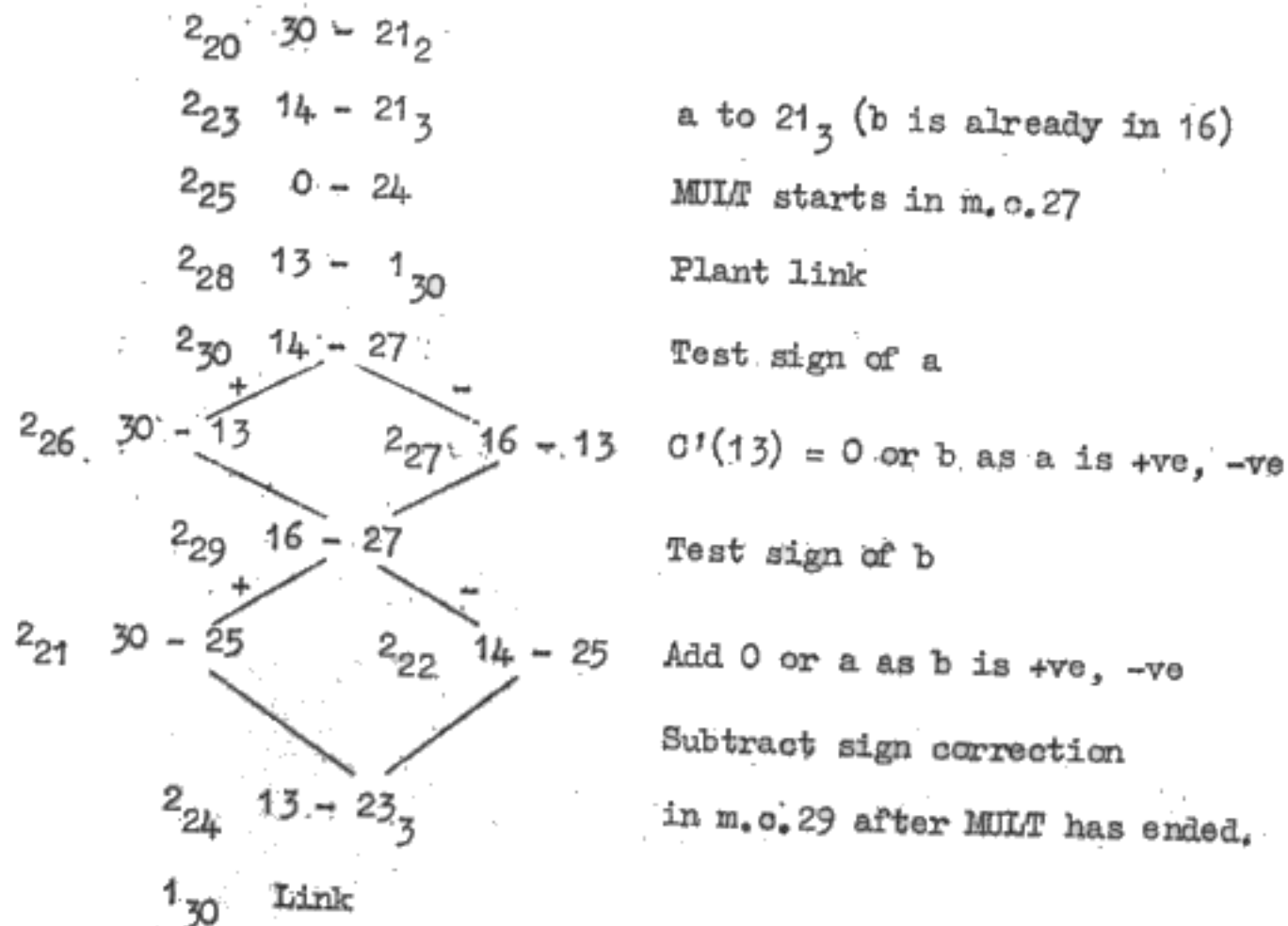
If both a and b are negative the multiplier produces $(2^{32} + a)(2^{32} + b) = 2^{64} + 2^{32}(a + b) + ab$, so the subroutine must subtract $2^{32}(a + b)$ from the result given by the multiplier. This appears to give $2^{64} + ab$, not the required result, but as DS 21 only holds 64 binary digits the 2^{64} is a single digit beyond the more significant end of the result.

The method used to subtract $2^{32}c$ from the double-length number in 21 is simply to subtract c from 21₃.

The rule for sign-correction is:

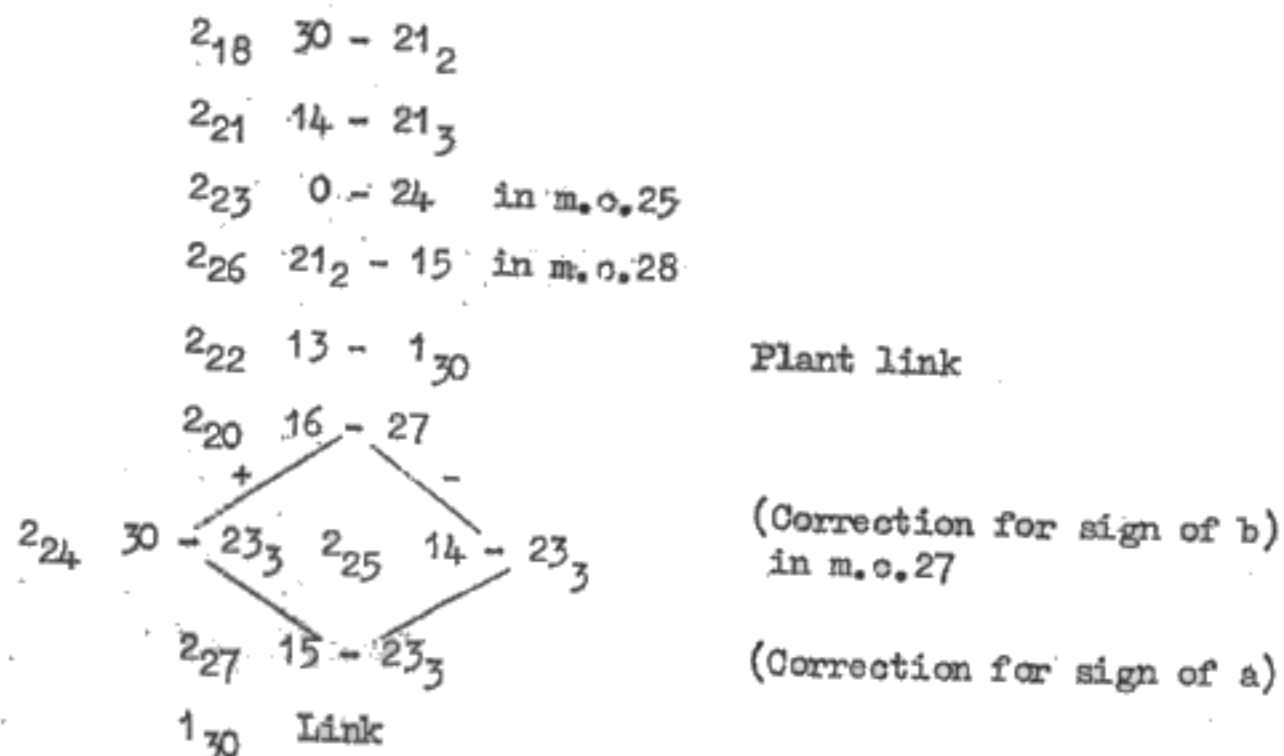
If either of a or b is negative, subtract the other from 21₃.

The complete subroutine for multiplying signed integers is



As much of the work as possible has been done while the multiplier is working so that the subroutine is as fast as possible. It fills m.c.20 to 30 of IL 2. The reason for the apparently haphazard locations for the instructions after the one in 2_{30} is time wasting. The time between starting the multiplier and reaching the instruction in 2_{29} is 34 minor cycles. Another 32 minor cycles pass before a number is sent to DS 21 so that 66 minor cycles elapse between the starting of the multiplier and the next use of DS 21.

There is an alternative method of evaluating the sign-correction which uses the first partial product. If the multiplier is started in m.c. 25 with a in 21_3 and b in 16, it looks at the sign of a in m.c.25 and adds b to 21 in m.c.26 only if a is negative. This is exactly what is done by the three instructions in 2_{30} , 2_{26} and 2_{27} in the above example. So source 21 in m.c.28 will give 0 or b according as a is positive or negative. The subroutine then becomes



The subroutine now fills m.c.18 and 20 - 27 of DL 2; two instructions have been saved at the expense of two minor cycles in the time taken to obey the subroutine.

The disadvantages of this version compared with the previous one are:

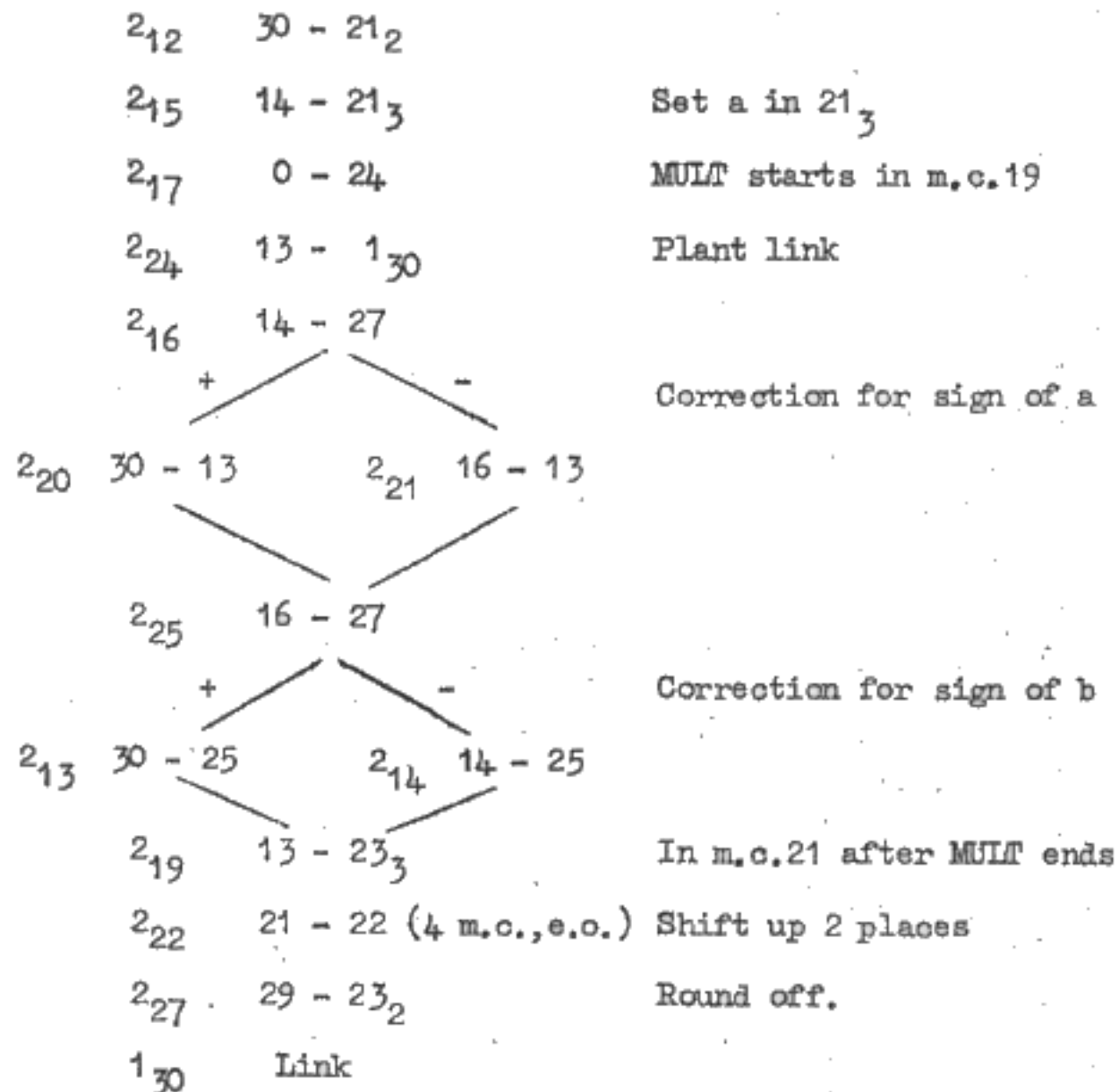
- (i) TS 15 is used.
- (ii) The instructions in 2_{23} and 2_{26} must be obeyed at full speed.

Obeying the instructions with the machine stopped will give the wrong result.

Multiplication of signed fractions

Suppose one of the subroutines for multiplying signed integers is given two fractions a and b with 30 binary places. It multiplies the integers 2_{30}^a and 2_{30}^b and gets the product 2_{60}^{ab} , or ab as a fraction with 60 binary places (28 b.p. in 21_3). If the product is then shifted up two places the result is the fraction ab to 30 b.p. in 21_3 . It is usual to round off ab by the method of example 23 of I.10 - example 25 is less common because the digits left in 21_2 can be of use, e.g. in accumulating the sum of products, which it is preferable to do double length rather than single-length for the sake of accuracy.

A subroutine to multiply signed fractions with 30 b.p. is as follows:



This is the subroutine M.12 (No.14).

The divider

The divider is also given its numbers a and b in 21_3 , and 16. It tests them as signed numbers and, provided they have the same number of binary places, it gives the quotient q in 21_2 and a form of the remainder in 21_3 .

The division process is defined by

$$a = qb + r$$

where

$$0 \leq r < b \text{ for } b > 0$$

$$b \leq r < 0 \text{ for } b < 0$$

q cannot be greater than the exact quotient: in fact it is the exact quotient truncated to 31 b.p. Because q must be single-length we must have $|a| < |b|^*$.

r has the same number of binary places as a and b. The divider produces not r but $R = 2(2r - b)$ in 21_3 : in any case, no known program uses this remainder.

To operate the divider place b in 16 and a in 21_3 . Curiously, b must not be placed in 16 in the last odd minor cycle before division starts. 21_2 does not have to be clear, but many subroutines do clear it because they have a validity test (see below). The divider is started by the instruction 1 - 24 in an odd minor cycle, runs for 66 minor cycles and leaves q and R in 21_2 and 21_3 . TCB is switched on by the 1 - 24 instruction and is left on at the end of the division.

If the division instruction is obeyed in m.c.25 the divider runs until m.c.26 of the next major cycle but one. Just as with the multiplier, destinations 16, 21, 22 and 23 must not be used, sources 21 and 22 give nearly useless partial quotients and remainders and source 16 can be used freely.

A subroutine is normally employed when division is needed although no sign correction is necessary. Subroutines usually spend the 66 minor cycles which have to be wasted in testing whether the quotient exceeds capacity. The method consists of picking out one of the partial quotients from 21_2 and testing this while the divider is still running.

33 digits of the quotient are produced by the divider during its 66 minor cycles and, since TCB is on, the first one produced is shifted off and lost from 21_2 in the last minor cycle. When the dividend is greater than the divisor this "extra" digit is different from its neighbour, the one which will end up as the sign digit of the quotient; however, if the dividend is less than the divisor, the quotient is a string of ones or noughts. The test on the validity of the division consists of picking out a partial quotient while the "extra" digit is still available and comparing it with its neighbour. For instance, if the 1 - 24 instruction is obeyed in minor cycle 25 the "extra" digit is the P_1 digit from source 21 in m.c.26, the P_2 digit in m.c.28, the P_{17} digit in m.c.26 of the next major cycle, etc. In the example given below the partial quotient is placed in 14 when it has reached the P_{17} position. It is shifted up 15 places in 14 so that the "extra" digit is in the P_{32} position, then double this number is placed in 15. The sign digit

* Or, exceptionally, $a = -b$ with b positive.

of the final quotient is the sign digit of C(15). As explained in I.12, source 26 will give a negative number when the two digits differ and a positive number when the digits are identical and the division invalid.

The following subroutine divides the number a in 14 by the number b in 16. The quotient is given in 21₂ to 30 b.p. and the validity of the division is tested. The quotient has to be shifted down one place, so the method of I.11, example 30, is used. TCB is put on by the 1 - 24 instruction and has to be put off by the subroutine after the division has ended.

15. Because 13 is not needed in the subroutine the link is placed in

$2_{18} \quad 30 - 21_2$
 $2_{21} \quad 14 - 21_3$
 $2_{23} \quad 1 - 24$
 $2_{26} \quad 21_2 - 14_{26}$
 $2_{27} \quad 15 - 1_{30}$
 $2_{30} \quad 24 - 14 \text{ (15 m.c.)}$
 $2_{20} \quad 24 - 15$
 $2_{22} \quad 26 - 27$
 $2_{24} \quad 7 - 24 \leftarrow$
 $2_{25} \quad 22_2 - 23_2$
 $2_{28} \quad 4 - 24$
 $1_{30} \quad \text{Link}$

Clear 21_2 - necessary for validity test.
 Set a in 21_3
 Division starts in m.c.25.
 in m.c.26 (33 m.c. later)
 Plant link
 in m.c.28 after division ends.
 TOB off

I.13 Logical operations

Source 25 and source 26 both give words which are digit-by-digit combinations of the words in TS 14 and TS 15. These combinations are known as logical operations since they are derived from Boolean algebra. Unlike arithmetical operations there is no carry between digits - the r'th digit of the word given by either source is a function of the r'th digits only of the words in 14 and 15. The use of either source does not affect the words in 14 and 15 - unless, of course, destination 14 or 15 is used.

If x and y are single binary digits, either 0 or 1, the three logical operations which can be performed on them are:

"or"	$x \vee y$	1 if x or y or both is 1; 0 if x and y are 0.
"and"	$x \& y$	1 if x and y are 1; 0 if x or y or both are 0.
"not equivalent"	$x \ncong y$	1 if x and y are not equivalent (opposite); 0 if x and y are equivalent (the same).

Alternatively the operations can be expressed by the following table:-

(x,y)	(0,0)	(1,0)	(0,1)	(1,1)
$x \vee y$	0	1	1	1
$x \& y$	0	0	0	1
$x \neq y$	0	1	1	0

It will be seen that the order of x and y does not matter; i.e. $x^*y = y^*x$ where * denotes any of the three operations.

For words of several binary digits the logical operations are performed digit by digit. Thus, if $a = 0101$, $b = 0011$,

$$a \vee b = 0111$$

$$a \& b = 0001$$

$$a \neq b = 0110$$

Sources 25 and 26 perform logical operations on the words in 14 and 15; source 25 gives $C(14) \& C(15)$; source 26 gives $C(14) \neq C(15)$.

Examples

In examples where logical operations are performed on two numbers we need consider only $a = 0101$, $b = 0011$, as no other combinations of 0 and 1 can occur. If three words were involved we should take $a = 01010101$, $b = 00110011$, $c = 00001111$.

Example 1 Replace $C(14)$ by the word which has 1 where it has 0 and 0 where it has 1. (This word, the ones-complement of $C(14)$, is called not $C(14)$ and written $\sim C(14)$.)

Let $C(14) = 010101\dots$

$$31 - 15 \quad 1111\dots$$

$$26 - 14 \quad 101010\dots \text{ to } 14.$$

Example 2 Place $C(14) \vee C(15)$ in 13. We shall take $C(14) = 0101$, $C(15) = 0011$.

$$26 - 13 \quad C'(13) = 0110$$

$$25 - 14 \quad C'(14) = 0001$$

$$13 - 15 \quad C'(15) = 0110$$

$$26 - 13 \quad C'(13) = 0111$$

An alternative method can be used since $a \& b$ (from source 25) and $a \neq b$ (from source 26) can have no digits in common.

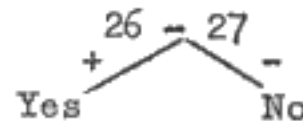
$$26 - 13 \quad C'(13) = 0110$$

$$25 - 25 \quad C'(13) = 0110 + 0001 = 0111.$$

Example 3 Interchange $C(14)$ and $C(15)$ without using another store.

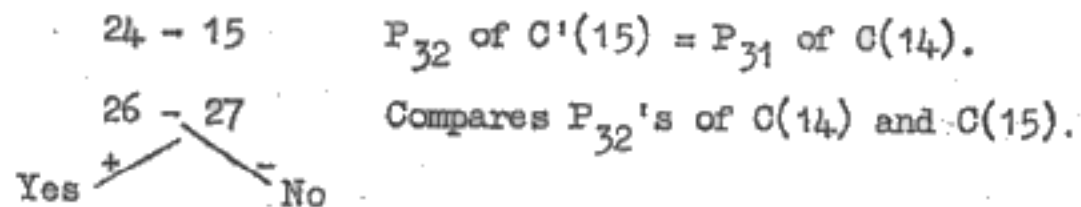
	$C(14)$	$C(15)$
	0101	0011
$26 - 14$	0110	0011
$26 - 15$	0110	0101
$26 - 14$	0011	0101

Example 4 Have $C(14)$ and $C(15)$ the same sign?



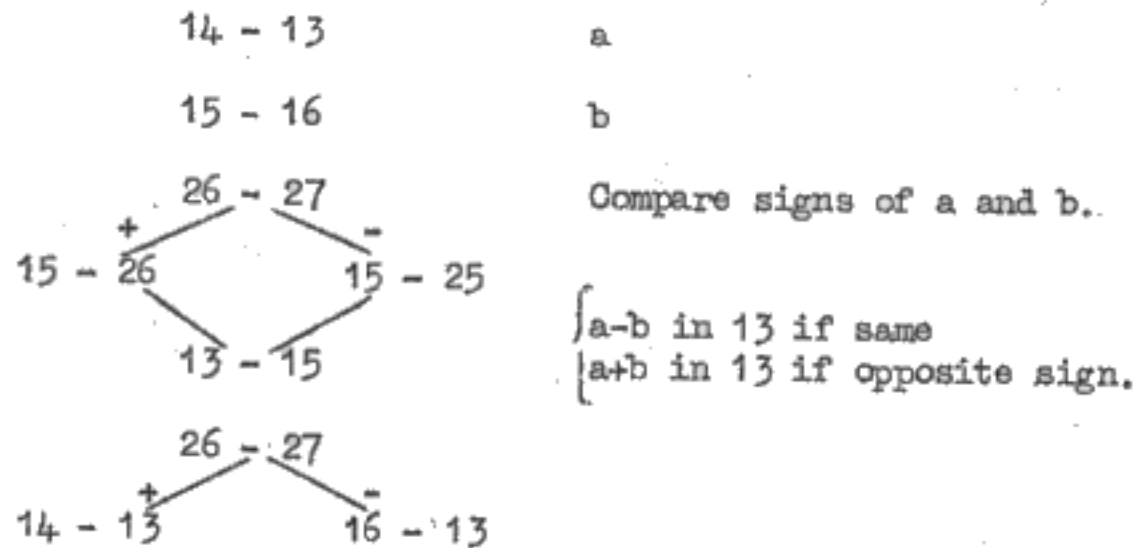
If they have the same sign they have the same sign digit. The sign digit of the word given by source 26 is therefore 0 and the word is positive.

Example 5 Are the top two digits of $C(14)$ alike?



Example 6 Place the numerically greater of $C(14)$ and $C(15)$ in 13.

Write $a = C(14)$, $b = C(15)$



This method is shorter than Example 10 of I.4 but more difficult to explain. Its chief disadvantage is the somewhat haphazard behaviour when the numbers are numerically equal: it should only be used when it does not matter which number is then chosen.

The only way to explain the method is to set out the eight possibilities when a and b are not numerically equal:

- (1) $a > b > 0$, so $a-b > 0$. $a \neq b > 0$, so $a \neq (a-b) > 0$, so a chosen
- (2) $b > a > 0$, so $a-b < 0$. $a \neq b > 0$, so $a \neq (a-b) < 0$, so b chosen.
- (3) $a < b < 0$, so $a-b < 0$. $a \neq b > 0$, so $a \neq (a-b) > 0$, so a chosen
- (4) $b < a < 0$, so $a-b > 0$. $a \neq b > 0$, so $a \neq (a-b) < 0$, so b chosen
- (5) $a > |b| > 0 > b$, so $a+b > 0$. $a \neq b < 0$, so $a \neq (a+b) > 0$, so a chosen
- (6) $|b| > a > 0 > b$, so $a+b < 0$. $a \neq b < 0$, so $a \neq (a+b) < 0$, so b chosen
- (7) $|a| > b > 0 > a$, so $a+b < 0$. $a \neq b < 0$, so $a \neq (a+b) > 0$, so a chosen
- (8) $b > |a| > 0 > a$, so $a+b > 0$. $a \neq b < 0$, so $a \neq (a+b) < 0$, so b chosen

It is easy to see that the correct number has been chosen in each case.

The positive side of the first discrimination is taken by cases (1)-(4), the positive side of the second discrimination by the odd-numbered cases.

Example 7 Place in 15 the wait number of the instruction in 14.

$$- 15 \quad P_{17-21} = 31P_{17}$$

$$25 - 15 \quad (N, S-D, Ch, W, T) \& (0, 0-0, 0, 31, 0 X) = (0, 0-0, 0, W, 0 X)$$

The constant $31P_{17}$ is used as a collating mark. (The & operation is also called collation.) Where it has zeros source 25 must give zeros: where it has ones source 25 gives the digits of 0(14).

I.14 Destination 'Triggers'

The name "destination triggers" is given to destination 24 because any n - 24 instruction sets an electronic trigger. The full list is

- 0 - 24 Start the multiplier
- 1 - 24 Start the divider
- 2 - 24 Discrimination on T.I.L.
- 3 - 24 Put on TCA
- 4 - 24 Put off TCB
- 5 - 24 Put on TCB
- 6 - 24 Put off alarm
- 7 - 24 Put on alarm
- 8 - 24 Clear output staticizer
- 9 - 24 Stop passing cards through reader or punch
- 10 - 24 Start passing cards through punch
- 12 - 24 Start passing cards through reader.

None of these operations involves the transfer of any number, so they do not fit easily into the Source-to-Destination form of DEUCE instruction. On the Pilot Model ACE (where there were fewer of them) a different destination was used for each and the source number did not matter. As this would be too wasteful on DEUCE all such instructions have the same destination (D 24) and different source numbers. Of course the source number in such an instruction has nothing to do with the usual source, e.g. 1 - 24 has nothing to do with DL 1.

No other triggers are used at present although 32 are available. Proposals for fitting magnetic tape units will use almost all the trigger sources from 16 upwards: at the moment the P_9 digit is ignored so that $n - 24$ and $(16 + n) - 24$ have the same effect.

11 - 24 was omitted because it would have called in the interlock when the magnetic drum was being used (see I.16).

Of the above twelve trigger instructions, triggers 0 and 1 were explained in I.12 and triggers 4 and 5 in I.11. Triggers 2, 9, 10 and 12 are used to operate the reader and the punch and will be explained in I.15.

TCA - Trigger 3

Trigger TCA provides one of the most curious and least used facilities - one which is used to avoid instructions which have to be modified.

TS 16 is normally employed to store a single word which circulates in the usual way and is available in every minor cycle. However, if TCA is switched on by the 3 - 24 instruction, the recirculation path is broken. Instead TS 16 is connected to DL 10. The word which comes out of DL 10 in m.c. n is always fed back into DL 10 to circulate again (unless, of course, a transfer to destination 10 is in operation). When TCA is on this word is also fed into TS 16. It takes one minor cycle to flow through TS 16 and so it comes out of TS 16 in minor cycle $(n + 1)$ just as $C(10_{n+1})$ comes out of DL 10. The word does not go anywhere from TS 16 unless source 16 is called for - it does not recirculate in TS 16.

The instruction

16 - 9₅ with TCA on

transfers a word from TS 16 to DL 9 in m.c.5. Since it comes out of TS 16 in m.c.5 it must have entered it from DL 10 in m.c.4, so the operation is

$$C'(9_5) = C(10_4). \text{ Of course } C'(10) = C(10).$$

Similarly,

16 - 9 (32 m.c.) with TCA on, starting in m.c.5, say,
puts $C'(9_5) = C(10_4)$, $C'(9_6) = C(10_5)$, $C'(9_7) = C(10_6)$, ... $C'(9_{31}) = C(10_{30})$,
so it transfers DL 10 into DL 9 with a delay of one minor cycle.

The instruction

16 - 10 (32 m.c.) with TCA on, starting in m.c.5, say,
puts $C'(10_5) = C(10_4)$, $C'(10_6) = C(10_5)$, ... and finally $C'(10_{31}) = C(10_{30})$.
It therefore moves all the words in DL 10 round by one minor cycle.

TCA is cleared when a word is sent to destination 16. The programmer, therefore, does not have to worry whether TS 16 will hold a word if one is sent to it: it is automatically switched back to behave as a TS when the word is sent.

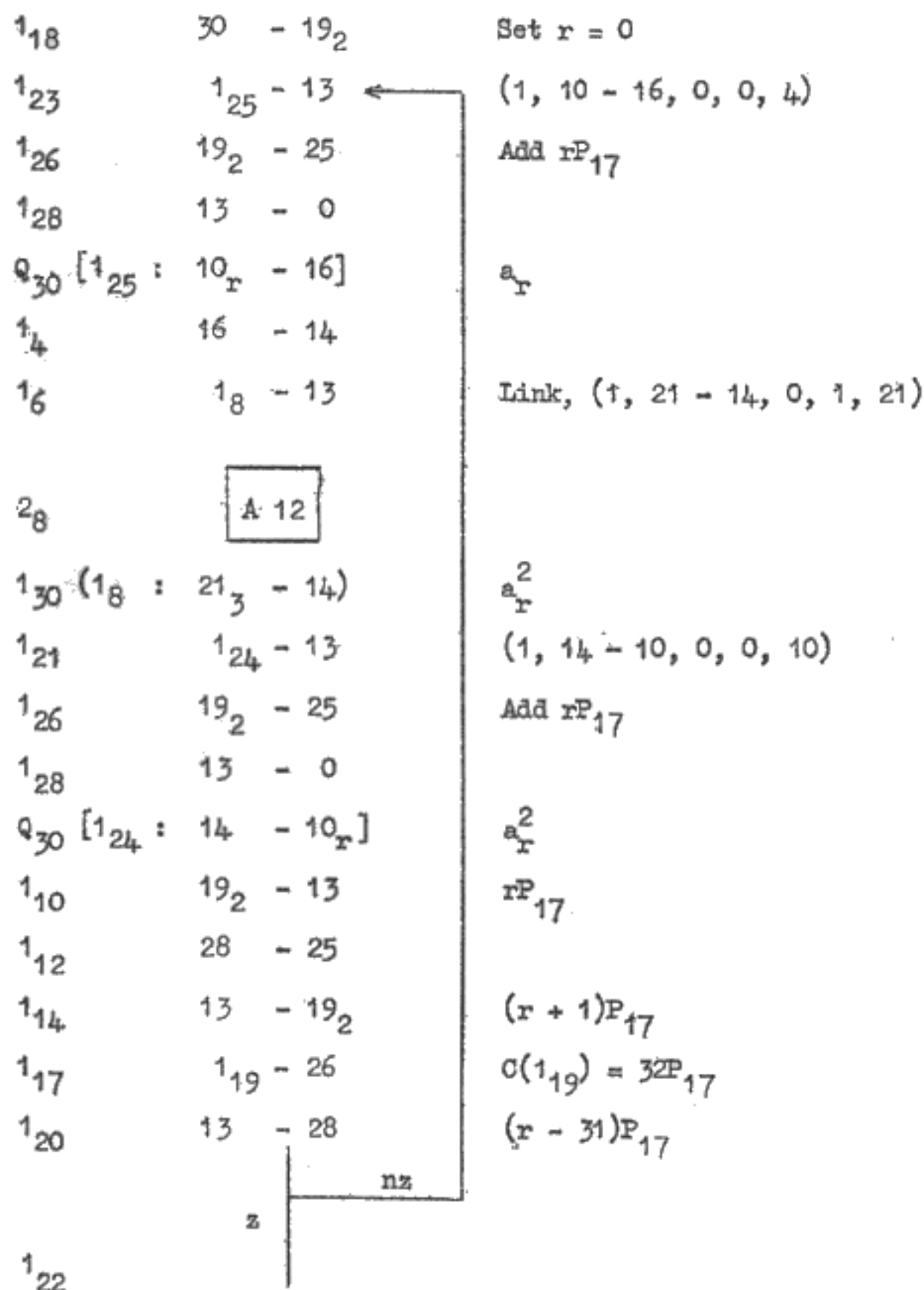
In general source 10 in minor cycle n gives the same word as source 16 in minor cycle $(n + 1)$ when TCA is on. The only exception is when TCA is switched off so that TS 16 shall hold a particular word from DL 10. In this case $10_n - 16$ must be used in preference to $16_n - 16$. The latter switches TCA off but leaves a hybrid word in TS 16: the former correctly places $C(10_n)$ in 16.

TCA can be used to avoid modifying instructions.

Example Replace all the numbers in DL 10 by their squares.

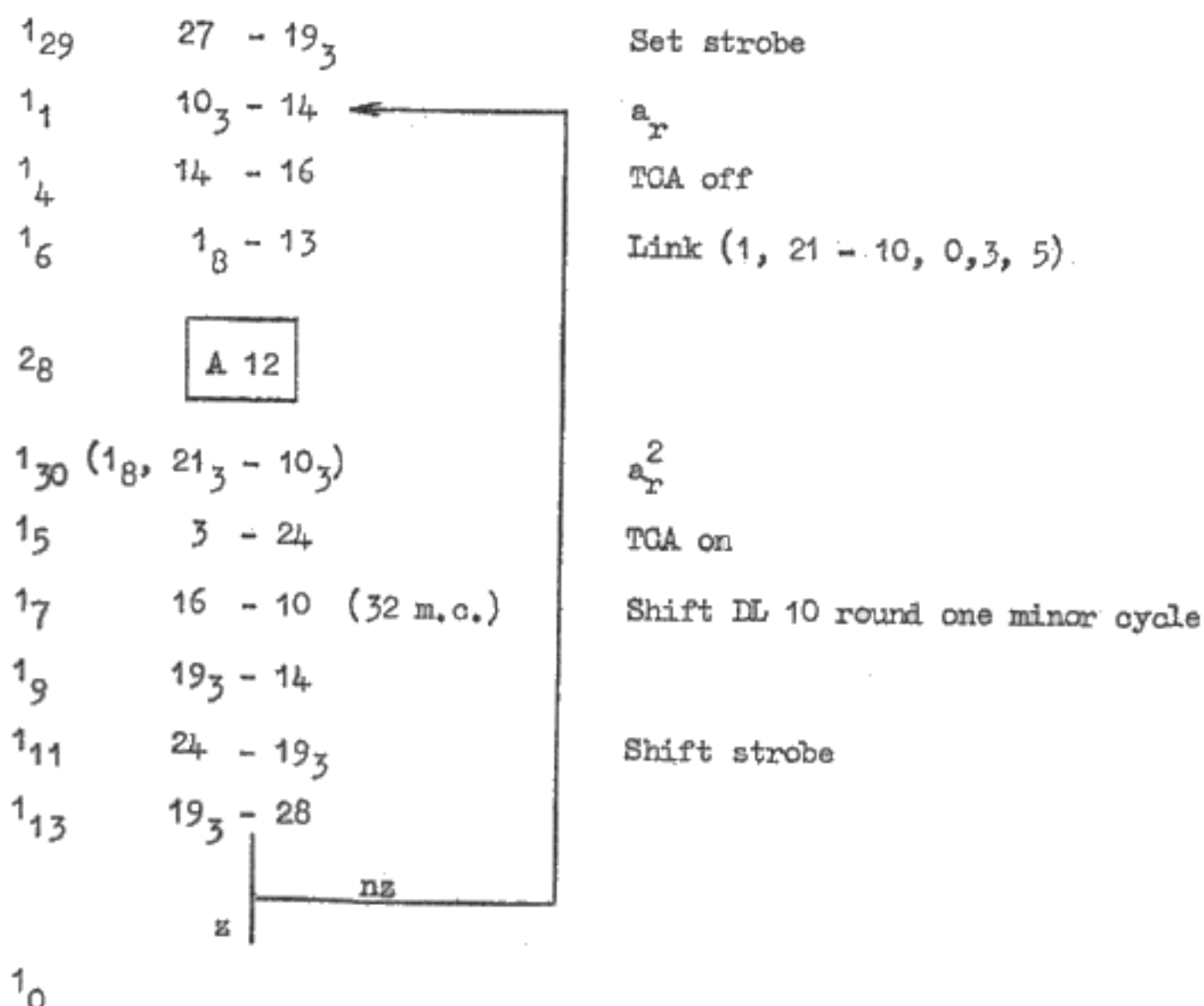
(The multiplication subroutine A 12 (No.221) will be used.)

(i) Using modified instructions.



This program needs 17 instructions and takes 208 major cycles. (It is rather crude - use of the spill-over technique can save 5 instructions and 34 major cycles. A.I.M. would save even more instructions.)

(ii) Using TCA



This program needs 11 instructions and 160 major cycles.

The numbers in $10_0, 10_1, \dots, 10_{31}$, are called $a_0, a_1, \dots, a_{31}$. Program (i) uses a variable order to pick out a_r (starting with $r = 0$), squares it and plants the result in 10_r with another variable order. It operates on $a_0, a_1, \dots, a_{31}$ in that order.

Program (ii) picks out a_3 , squares it and replaces it in 10_3 . It then uses TCA to move the contents of DL 10 round; the new contents will be $a_{31}, a_0, a_1, a_2, a_3^2, a_4, \dots, a_{30}$. Next time round the loop a_2 is in 10_3 and is replaced by a_2^2 . Then next move round of DL 10 gives $a_{30}, a_{31}, a_0, a_1, a_2^2, a_3^2, a_4, \dots, a_{29}$, so a_1 is the next term squared.

A strobe in 19_3 starts at P_1 and is shifted up one digit each time. After 32 loops it becomes 0. All 32 numbers have then been squared and a_3^2 , which started in 10_3 , has been moved through 32 minor cycles so that it is back in 10_3 .

This time the operations have been on $a_3, a_2, a_1, a_0, a_{31}, \dots, a_4$. The choice of a_3 as the first term was arbitrary, but working backwards through the numbers was compulsory. The method using TCA is less flexible than variable instructions, which could equally well work forwards or

backwards: the use of TCA makes working backwards essential. This means that where the sequence in which the numbers are treated matters (unlike the example above) arrangements must be made to have the first number in the latest minor cycle. This is probably the reason why TCA is so little used.

Alarm - Triggers 6 and 7

The alarm consists of a buzzer behind the main console and a red light on the panel. Below the light is a key marked "Clear Alarm".

When the instruction 7 - 24 is obeyed the alarm is put on: the buzzer sounds and the red light comes on. The instruction 6 - 24 is used to clear both. Alternatively the Clear Alarm key can be pressed down and released: the effect is just that of the 6 - 24 instruction. If the Clear Alarm key is left up the buzzer will not sound but the light still reacts to the 6 - 24 and 7 - 24 instructions.

The alarm is used chiefly as a programmed warning that an error has occurred; for example, a scaling error is detected when a division subroutine finds the numerator is greater than the denominator, or a card-punching error is detected when a subroutine reads decimally-punched cards. The most unimaginative failure instruction is 7 - 24 leading back to itself: how is the programmer to know which failure has occurred? Ideally, when a failure has been detected the program should sound the buzzer, place some word on the output staticizer (e.g. indicate the incorrect columns if a card is not properly punched in decimal) and stop; on receiving a single shot the program should clear the buzzer and the output staticizer and repeat the calculation where this is possible.

At R.A.E. both DEUCES have had an oscillator fitted to the alarm. This goes on and off with the red light, not with the buzzer as might have been expected.

Output staticizer - Trigger 8

There is a trigger instruction to clear the output staticizer but not to put a word on it. Destination 29 is connected to the punch through the output staticizer, so when the punch is not running, an instruction with destination 29 will cause the word to be displayed on the output staticizer lights. If there is a word already on the staticizer when a new one is sent to destination 29 the first word is not cleared: the two words are superimposed. It is therefore necessary to put in an 8 - 24 instruction to clear the staticizer before sending the second word to destination 29.

At the right hand end of the output staticizer lights on the console is a key which clears the output staticizer. Its effect is just the same as that of an 8 - 24 instruction.

The 8 - 24 instruction is also used with Ch = 1 to switch from the α -field to the β -field when punching (I.15).

I.15 The reader and the punch

The standard form of input and output of numbers from DEUCE is on Hollerith cards. Hollerith cards have 12 rows (labelled Y, X, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 reading from top to bottom) and 80 columns. Numbers are usually punched on them in decimal with not more than one hole in each column, but numbers may be punched in binary when the cards are to be used on DEUCE. If, for instance, the number -9876543210 is to be punched in decimal starting in column 17, holes are punched in rows X, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0 in columns 17, 18, ..., 27 respectively - the Y row indicates a + sign and the X row a - sign.

DEUCE has two Hollerith machines attached to it, a reader and a punch. In both of them the cards move along the length of the columns, i.e. all the holes in one row simultaneously come into position under the reading brushes or punch knives. The cards travel with the Y-edge leading - in this way they are unlike most of the ancillary Hollerith equipment not attached to DEUCE, e.g. the tabulator, on which the 9-edge of the card leads. The reader's speed of 200 cards per minute is twice that of the punch: how this affects the programming is explained in the "Hollerith timing" section below.

All twelve rows pass under the brushes or knives as the case may be, but only 64 of the 80 columns are used by DEUCE. The reader has brushes in columns 17 - 80 only, split into two fields, the α -field extending from column 17 to 48, while the β -field covers the remaining 32 columns. As a row comes into position, source 0 gives the 32-digit binary word punched on that row in one of the two fields, reading holes as "ones" and absence of holes as "noughts". How to switch from one field to the other is described below. (There is also a brush in column 1 whose use is explained later.)

The punch has knives in columns 17 to 80, similarly split into an α -field and a β -field. A 32 digit binary number sent to destination 29 when a row is in position under the knives is punched in one of the two fields: "ones" are punched and "noughts" are not punched. The punch also has knives in columns 5 to 16 for punching identification marks in decimal but, as these are not controlled by the program and cannot be read back into DEUCE, they are explained in section II.1 instead of here.

Because a 32-bit number is read or punched at one operation when a card is in position it is much easier to program the reading and punching of numbers in binary than in decimal. It is also more compact because a card can hold 24 numbers punched in binary but usually not more than six in decimal. This section is concerned with the programming of the input and output of binary numbers: decimal numbers will be read and punched by subroutines, but, for interest only, a sketch of the methods used is given at the end of this section. In practice, numbers should be read and punched in decimal: the loss of time is usually negligible, the use of subroutines saves any extra trouble in programming and the preparation of data cards and interpretation of results is far simpler.

Except for the obvious differences that the reader uses source 0 and the punch destination 29, and that certain triggers are numbered differently there is very little difference for programming purposes between reader and punch. The reader is therefore explained more fully than the punch to save much repetition: unless otherwise stated it can be assumed that the behaviour of the punch is similar to that of the reader.

The reader

The reader is called by the instruction 12 - 24. Provided it is in the "ready" state (i.e. the cards are not all in the top feed - see II.1) the cards then run past the reading brushes. Source 0 is switched from the Input Dynamicizer keys to the reading brushes.

When a row of a card is under the reading brushes source 0 gives the 32-digit word punched on that row in either columns 17 - 48 or 49 - 80. Normally it is the α -field (columns 17 - 48) that is being read. When the brushes are between the rows of a card (or between cards) source 0 gives rubbish. So that only sense shall be read the reader gives a single-shot when each row comes into position. The programmer makes the instruction to read the α -field of the row a "stopper", so that when the

program reaches the instruction it does not immediately read whatever is under the brushes: instead, it waits for the single-shot which is given when the next row is in position. This reads the α -field. It is then possible to have another "quick look" at the α -field. The exact timing specification is as follows:- If the first source 0 instruction is in minor cycle m , and has timing number t , the α -field is available until minor cycle $m + t + 6$ inclusive, and the β -field from $m + t + 22$ onwards. Both fields must be read within 2 major cycles of the single-shot for the row.

All instructions with source 0, except 0 - 24, 0 - 30 and 0 - 31 affect the α - β switching circuits, so spurious ones should be avoided whilst the reader is running. The waste instruction 0 - 0 in particular should be guarded against.

The instructions:

$1_{19} : 0 - 14 X$
 $1_{21} : 0 - 15 X$

will read the α -field of successive rows, not the same row twice, because the single-shot for any row is cut off as soon as it is used. In this example the β -field was ignored. The instructions

$1_{19} : 0 - 14 X$
 $1_{21} : 0 - 15$
 $1_{23} : 0 - 16 X$

have different effects according to the wait number of the instruction in 1_{21} . The two stopped instructions will definitely read the α -field of successive rows. If the instruction in 1_{21} has $W = 0$ it reads the α -field again, if W is between 3 and 17 we get rubbish, and $W > 18$ would give the β -field.

The programmer does have to make certain that not too much time is spent between the stopped instructions. At least 15 major cycles elapse between successive rows of a card coming under the reading brushes, so not more than 15 major cycles should be spent on computing (including reading the β -field) between rows. If more than 21 major cycles are spent between stopped instructions a row will pass the reading brushes and be missed: alternate rows will be read instead of successive ones. More time is available between cards than between successive rows of the same card.

In between the 9 row of one card and the Y row of the next up to 80 major cycles' computing time is available: the Y row will be missed if over 100 major cycles are spent between the instruction to read the 9 row α -field and the next stopped instruction.

The reader, being a mechanical device, cannot be kept to tolerances as strict as the rest of DEUCE, so that a single-shot is specified as being available between 15 and 21 major cycles after the previous one. Similarly, the time between cards is specified to be in the range 80 to 100 major cycles. No program should take a time which falls between the two limits - if it does, it may work some days and not others. In both the examples given the program must take less than the minimum specified time.

It is often useful to be able to distinguish the last row of a card from the others. This could be done by counting the rows as they pass, but

a simpler method is available because the reader gives out a signal called TIL (Twelfth Impulse Line) which lasts from 2 major cycles before the 9 row single-shot until 18 major cycles after it. The 2 - 24 instruction is a discrimination instruction and behaves like an instruction with destination 28: the next instruction is in m.c. $(m + T + 2)$ if the TIL signal is off and is in m.c. $(m + T + 3)$ if the TIL signal is on (i.e. round about the 9 row single-shot).

The 9 - 24 instruction is used to stop cards passing through the reader. Once it has been obeyed no new card will pass through the reader but if a card is already on the way through it will finish its journey without giving single-shots or TIL. This means that the 9 - 24 instruction stops the supply of single-shots and TIL from the reader, as would be expected, and when the reader is next called, the first row read will be the Y row of a new card.

When the reader is stopped between cards (this is more common than stopping it between rows of a card) the 9 - 24 instruction must come within 40 major cycles of the 9 row single-shot, otherwise an extra card may be slipped through the reader. Unless the 9 - 24 comes just before the Y row of the next card is in position, however, a lamp on the reader marked "Missed Card" comes on and no more cards are read*. On the other hand the 9 - 24 can come as near to the previous stopped instruction as one likes without affecting the accuracy of the reading.

The reader is also stopped when it finds a hole in Hollerith column 1. The machine behaves as if the instruction to read the row in which it occurred was immediately followed by a 9 - 24 instruction. In practice column 1 is punched only in the 9 row, but in any case this facility is not often used.

If 9 - 24 is used to stop the reader in the middle of a card there should be a gap of 2 major cycles before the reader is called again by a 12 - 24 instruction. This is not really a serious limitation; the reader is so rarely stopped except between cards. If the reader is switched off and on again too rapidly the card running through will continue to be read - single-shots and TIL will be available in the normal way - whereas one would expect to be reading the beginning of the following card.

The reader gives single-shots only when cards are actually passing the reading brushes. It is therefore perfectly safe to put only part of a pack of cards in the reader. When all the cards have been read, the machine halts on the stopped instruction which would read the α -field of the Y row on the next card. More cards are then placed in the reader and run in (see II.1) and the machine continues to read them.

The punch

There are only minor programming differences between the reader and the punch. The punch is called by the 10 - 24 instruction: cards run through the punch at the rate of 100 per minute and destination 29 is switched from the output staticizer lamps to the punch.

Instructions to punch on the α -field of cards are made "stoppers" and the punch supplies the single-shot when a row is in position under the punch knives. The binary word sent to destination 29 is punched in columns 17 to 48 of the card. To punch on the β -field we have a second instruction

* In several years at R.A.E. the only program to put on this lamp has been the one which is used to test it.

with destination 29. This one has no stopper. It must, however, be preceded by an instruction 8 - 24, 1. This clears the output staticizer and the characteristic of 1 effects the field switching. These three instructions must all be obeyed within 5 major cycles after the single-shot for the row. Because the punch runs more slowly than the reader up to 38 major cycles' computing time is available between the stopped instructions which are to punch on successive rows, but if more than 46 major cycles are spent on computing the instructions will punch on alternate rows as the intermediate row will be completely missed. Again the range 38 to 46 occurs because the speed of the punch cannot be guaranteed exactly.

In between cards up to 116 major cycles are allowed for computing. If more than 135 major cycles are spent after the 9 row single-shot the Y row will be missed and the X row will receive the next punching.

Because the reader and punch give single-shots to indicate that rows are in position it is impossible to use both simultaneously*. Therefore the 9 - 24 instruction is used to stop the punch as well as the reader. If the 9 - 24 comes between the rows of a card that card runs past the punch knives without giving single-shots or TIL and the punch stops with the cards ready for punching to start again on the Y row of the next card. The 9 - 24 should not be followed by a 10 - 24 within 2 major cycles or the single-shots and TIL from the card still running through will be made available, just as on the reader.

However the 9 - 24 instruction usually comes between cards. In this case it must come within 20 major cycles of the 9 row single shot or a blank card may slip into the pack of results.

The 9 - 24 instruction is the only way of stopping the punch. There is no equivalent of the hole in column 1 which stops the reader.

TIL is provided by the punch just as it is by the reader. It is on from 2 major cycles before the 9 row single-shot until 35 major cycles after it. During this time the TIL discriminator is on and the instruction following a 2 - 24 is taken from m.c. (m + T + 3).

If an insufficient number of blank cards have been placed in the punch the machine halts on the stopped instruction to punch the Y row of a card when the cards run out. If more blank cards are placed in the punch and run in (see II.1) the machine continues punching correctly.

Examples

Example 1 Read a, c, d punched in binary on the α -field of the Y, O and 1 rows of a card and punch 2a - c - d on the Y row of a card.

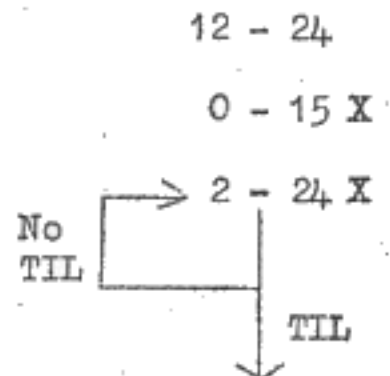
12 - 24	Start reader	
0 - 13 X	Obeyed when Y row in position.	$C'(13) = a.$
13 - 25 X	" " X " " "	$C'(13) = 2a$
0 - 26 X	" " 0 " " "	$C'(13) = 2a - c$
0 - 26 X	" " 1 " " "	$C'(13) = 2a - c - d$
9 - 24	Stop reader	
10 - 24	Start punch	
13 - 29 X	Obeyed when Y row in position.	Punch 2a - c - d
9 - 24	Stop punch.	

*The single-shot key must not give single-shots when either Hollerith machine is running. In fact it is disconnected when cards are running through reader or punch to prevent accidents.

If we are not interested in some row there is no need to make the stopped instruction one with source 0 or destination 29*. By making the 13 - 25 instruction a "stopper" we have ensured that the first 0 - 26 instruction is obeyed when the 0 row is in position. If the 13 - 25 were not stopped the program would calculate and punch 2a - b - c, where b is the number punched on the X row.

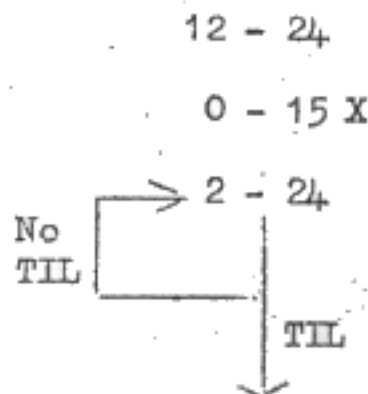
The 9 - 24 and 10 - 24 instructions are obeyed consecutively with less than 2 major cycles between them. This is correct because the 9 - 24 is stopping the reader: it is only if it is stopping the punch that it must not be followed immediately by a 10 - 24 instruction.

Example 2 Read the Y row of a card and "throw away" (i.e. ignore) the remainder of the card so that the reader is left running with the next single-shot coming from the Y row of the next card.



TIL is on when the 9 row is read but not when any other row is in position. The program stops on the 2 - 24 to read each row and only comes out of the loop after the 9 row single shot.

The loop in

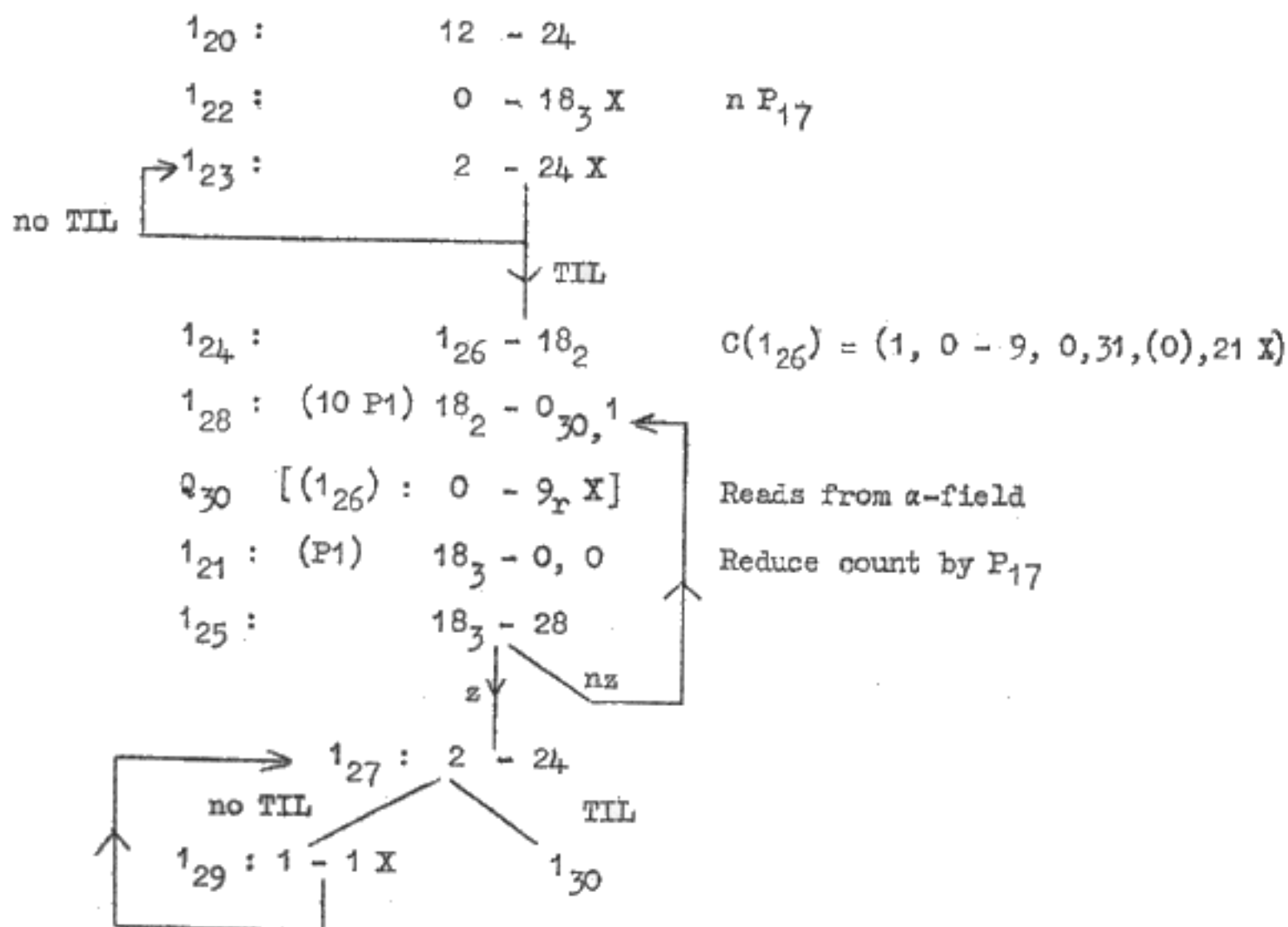


is left before the 9 row single shot.

Example 3 nP_{17} is punched in binary on the Y row of a card ($n \leq 32$). It is followed by n coefficients punched on the α -field of successive rows of cards starting with the Y row of the next card. Read nP_{17} and store it in TS 15, read the coefficients into DL 9 starting at m.c. 0 and leave the reader running so that the next row read is the Y row of the card following the coefficients.

/Program

* This applies to the α -field. The β -field can still be used if desired.

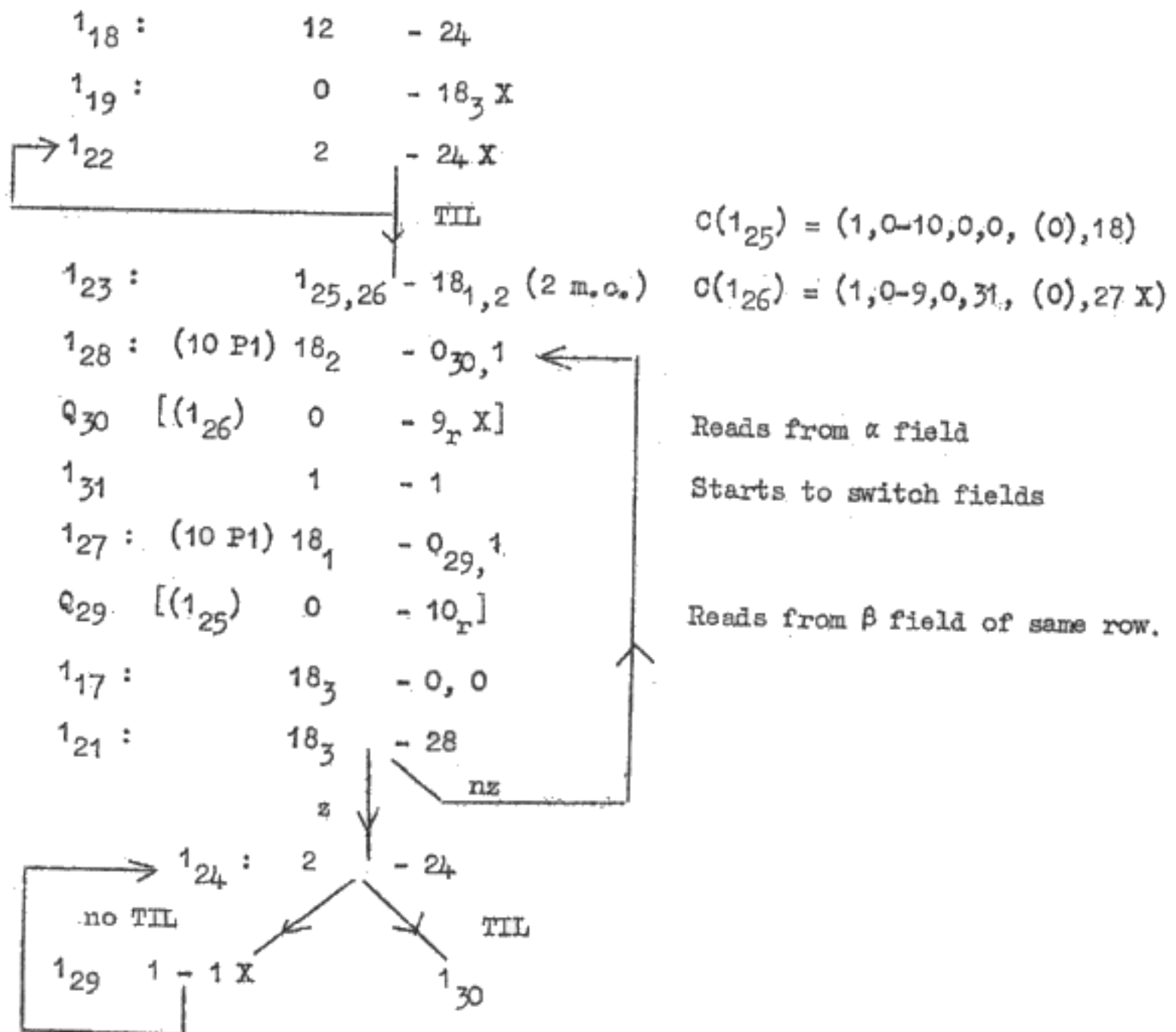


In this example the method of example 2 was used to "throw away" the remainder of the parameter card on which nP_{17} is punched. If this method had been used at the end of the program when n was a multiple of 12 an extra card would have been passed through the reader; this is the reason for the slightly more complicated method used.

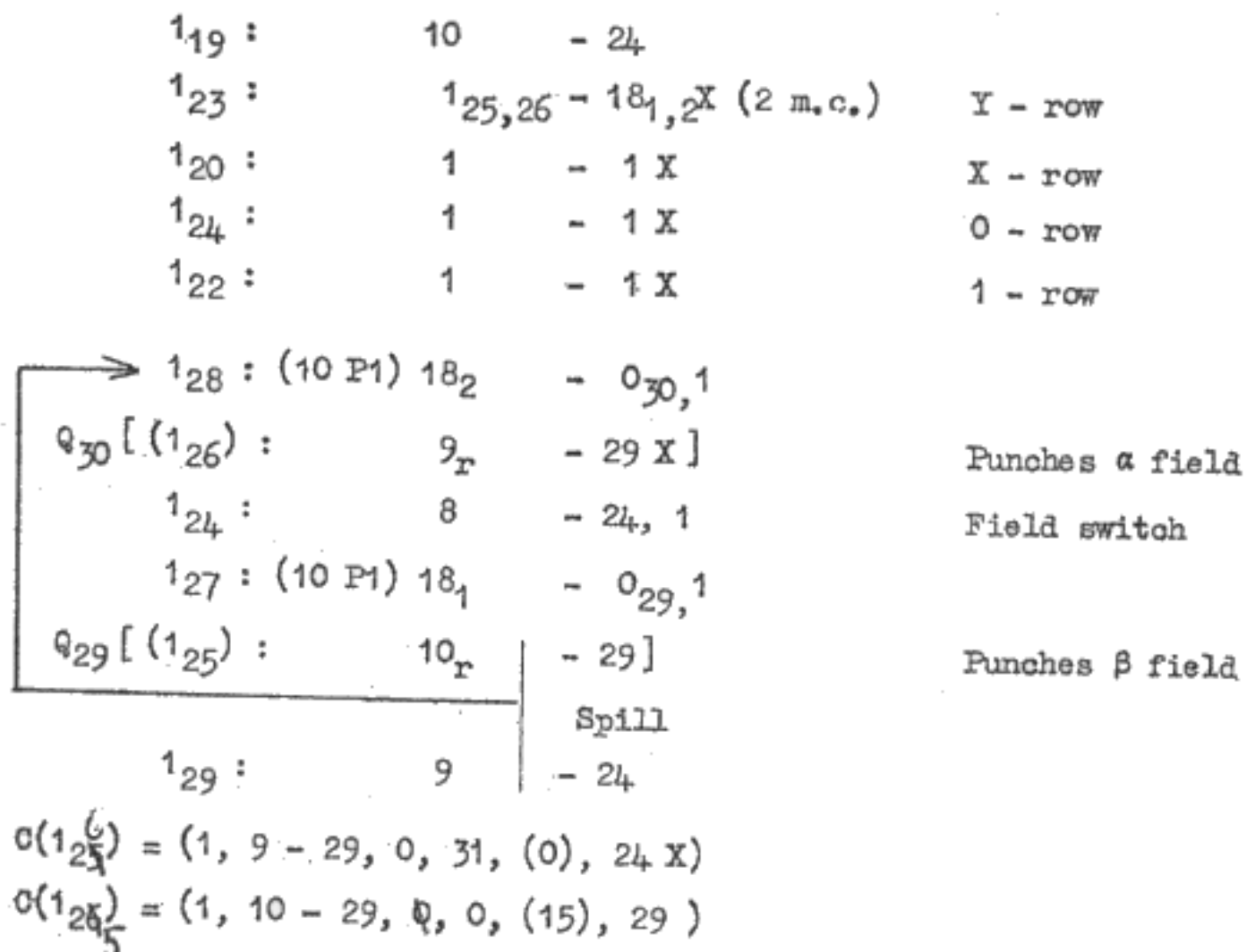
On this occasion the instruction in 1_{28} could have been written with $Ch = 0$ with $Y = 3P1$. The number n is lost.

Example 4. As above, except that there are n pairs of coefficients, punched on the α and β fields of successive cards, to be placed in parallel in DL's 9 and 10.

/Example



Example 5 To punch out in binary the contents of DL's 9 and 10 in parallel on the α and β fields of three cards, leaving the first four rows of the first card blank and stopping the punch at the end.



How decimal cards are read

It has been mentioned above that normally numbers should be punched in decimal. Yet the reader and the punch both will only accept binary numbers. The problem therefore arises of how to convert the sequence of binary words which a decimal card presents to the computer into the binary equivalent of the decimal number. A commonly used method is sketched below. For simplicity we shall assume that we are to read an integer punched in the first few columns of the α -field*.

From the Y and X rows we discover the sign of the number. There will be punching in only one of these rows. We shall build up the number positively and, perhaps, change its sign at the end. It is actually necessary to read only one of Y and X rows - reading both can be used as a check on accurate punching.

Suppose first that we are to read the single-digit integer 6. This has a single hole in the 6 row, in column 18. The first row that will be supplied by source 0, after the sign is determined, is the 0 row. We now add a P1 to a number (initially zero) until we come to a row with punching. In this case we add the P1 on arrival of the 0, 1, 2, 3, 4, 5 rows, i.e. 6 times in all, as is required. Notice that it is not necessary to read the 9 row, as a correctly punched card without punching in rows 0 to 8 has punching in row 9. Again, the check is useful.

For numbers of several digits we modify the above slightly. First of all, a "marker word" consisting of "ones" in those columns where there will be punching is used. Whenever a row is read it is compared with the marker word. "Ones" in the new row are used to remove "ones" from the marker. Next, we determine the contribution of this new row. We examine the digit of the marker word in the position of the most significant decimal digit. If this is zero, there has already been punching in this column: otherwise the most significant digit still has to contribute and we mark a P1. If there are further decimal digits, the contribution to date is multiplied by 10. The next digit is then examined, if necessary another P1 is added (the contribution is now one of 00, 01, 10, 11 - decimal numbers!), there is a further multiplication by 10, etc., until all the punched columns have been examined. This contribution is now added to the number that has so far been built up, and the next row can be read. The integer has been correctly converted to binary by the time the 9's row appears. Reading this can be used as a check against double punching, for the marker word after the 8's row and the punching on the 9's row should be identical.

This is a brief sketch of how integers are read. Fractions are read as if they are integers and finally divided by an appropriate power of 10.

The procedure for punching numbers is similar, except that it is easier to punch fractions than integers. Before an integer is punched, therefore, it is divided by a power of 10 and then punched as if it were a fraction.

Hollerith timing

The two Hollerith machines are electro-mechanical devices which run much more slowly than the electronic parts of DEUCE. Their speeds are not absolutely reliable - they would be reliable only if they could be guaranteed to a fraction of a microsecond, which is obviously impossible. Instead of

* It is not difficult to adapt the technique to deal with a number punched elsewhere, or to several numbers punched on the same card.

specifying a figure for a certain quantity (e.g. the time between single-shots from the reader is 18 major cycles) the engineers specify upper and lower limits for the quantity (in this example the time is at least 15 major cycles and not more than 21). This allows both for day-to-day variations in the performance of any one machine, and also for variations between the different DEUCES.

It is standard practice to write programs so that they can be used on any DEUCE. It is therefore essential that programmers should ensure that their programs do not, say, spend 16 major cycles on computing between rows when reading. This may work on the machine on which it is tested, but there is no guarantee that it will work on any other machine, or on the same machine later in the week.

The engineers have a set of programs to test that the various times between operations on the reader and the punch are within the specified limits. "DEUCE News" has published a list of the specifications, but, because this list was written as a list of what the programs were testing, it was not easy for the programmer to use. The list will be found in Appendix 4, where it has been re-written from the programmer's point of view.

The important times are the minimum ones between events. The programmer must treat these as his maxima.

I.16 The magnetic drum

I.16.1 General

In addition to the high speed (mercury) store DEUCE has an auxiliary store on a magnetic drum. This holds 8192 words. The drum may be used for storing data or instructions.

There are 256 tracks on the drum, numbered 0 - 255, each holding 32 words. Each track is therefore the equivalent of one DL of the high speed store. There are reading and writing heads for transferring information from the drum to a particular delay line (DL 11) of the high speed store and vice versa. The working of the reading and writing heads is entirely similar and in future we shall speak only of 'the heads': all remarks apply to reading and writing.

There are 16 heads numbered 0 - 15 mounted in a block that can move to one of 16 positions, numbered 0 - 15. Each head can therefore cover 16 tracks on the drum.

Choice and movement of the heads are controlled by destinations 30 and 31. Destination 31 is used for head position: the source gives the position number. Similarly, destination 30 is used for selecting the head and the source gives the head number.

The characteristic tells the machine whether to read or write. An even characteristic (0 or 2) is used for reading, an odd characteristic (1 or 3) for writing.

Example 1 The instructions

15 - 31 Head position

14 - 30 Head number

will shift the reading heads to position 15 and then read from head 14. (This is track 254 of the drum.)

Example 2

a - 31 (1)

b - 30 (1)

will write on track $(16a + b)$ of the drum. This is head b in position a . Often this track is written as track a/b : note that the head position comes first.

As the source and destination are both in use in instructions concerned with the drum, all transfers must be to a standard DL. This is DL 11 (not DL 12 as might have been expected).

Example 3 To write the contents of DL A on track a/b we have the sequence of instructions

A - 11 (32 m.c.)

a - 31 (1)

b - 30 (1)

If the heads are already in position a we can omit the second instruction.

Example 4. To clear all tracks of the drum.

$$1_{30} : \quad 30 - 11 \quad (32 \text{ m.c.})$$
$${}^1_{26}\text{Fe} : \quad {}^1_{24}\text{Mg} - {}^{17}_0\text{O}$$
$$1_{28} : (4P1) 17_0 \text{ f} - 0_0, 1$$

Adds P5 + P22

$$Q_0[(1_{24}) : a_1 - 31, 1]$$
$$1_{31} : \quad 1_{25} - 17_1$$
$$1_{27} : (4P1) \quad 17_1 - 0_1, 1$$

Add's P5 + P22

$$Q_{29} \quad [(1_{25}) : 1^6 - 30, 1]$$

Spill

 $1_0 : \Leftarrow$
$$C(1_{24}) = (1, 31 - 30, 1, 0, (15), 28)$$
$$C(1_{25}) = (1, 31 - 29, 1, 0, (0), 28)$$

The coding of the instructions in 1_{24} and 1_{25} may appear eccentric. 1_{24} is modified, before it is obeyed for the first time, to the more normal looking

1, 0 - 31, 1, 0, (0), 29 ,

as required. When the outer loop has been obeyed sixteen times this becomes

1, 16 - 31, 1, 0, (0), 30

and so takes 1_0 as the next instruction. Also the writing heads are removed to position 0.

The fate of 1_{25} is similar but simpler. It becomes

1, 0 - 30, 1, 0, (1), 28

before it is obeyed first, and later gets as far as

1, 15 - 30, 1, 0, (0), 29

when the inner loop is left and 17_1 is reset.

When we read from the drum we fill DL 11.

The reading and writing heads are independent and can be in different positions.

Movement of the heads and reading (writing) takes time. A head shift takes 35 ± 3 major cycles and reading (writing) takes 15 ± 1 major cycles. This time need not be wasted, but can be used for other computation. Once an instruction with destination 30 or 31 has been obeyed that operation proceeds independently of the rest of the machine. Thus, if a head shift is necessary, there is no reason for not initiating this some time before the information is required from the drum. No time is wasted if the heads are in the correct position already. Similarly, we can read from the drum some time before the information is required in the high speed store.

There is an 'interlock' associated with all magnetics instructions. This means that if one such instruction is being obeyed, say a head shift, when another enters TS Count, then the DEUCE waits until the first is completed before obeying the second. The interlock applies to destinations 30 and 31 and to DL 11. It is impossible to use DL 11 as either a source or destination while such a transfer is in progress. This, incidentally, is the reason why no instruction 11 - 24 exists - although the source here does not refer to DL 11 the interlock would still be effective.

I.16.2 The drum as a data store

There are several subroutines available to help fetch (store) data from (on) the drum. If we are always concerned with a whole DL a useful routine is B.02/1 (No.161), 'Magnetic Fetch and Carry'*. This is normally stored in DL 1 and requires a parameter in TS 15. The parameter is

$$N.P2 + m.P5 + \delta(P15 + P19) + t.P26$$

where N_t is the address of the instruction following the routine, m is the track number and $\delta = 0$ for reading and 1 for writing. B.02/1 includes a version of 'Fetch and Store' and the generally used 13 - 0 instruction in 1_{28} .

If we are concerned with single words only one of the most useful routines is B.06 (DEUCE Subroutine 200). This occupies 2 DL's for instructions and 2 DL's for data. It allows the fetching (storing) of consecutive words.

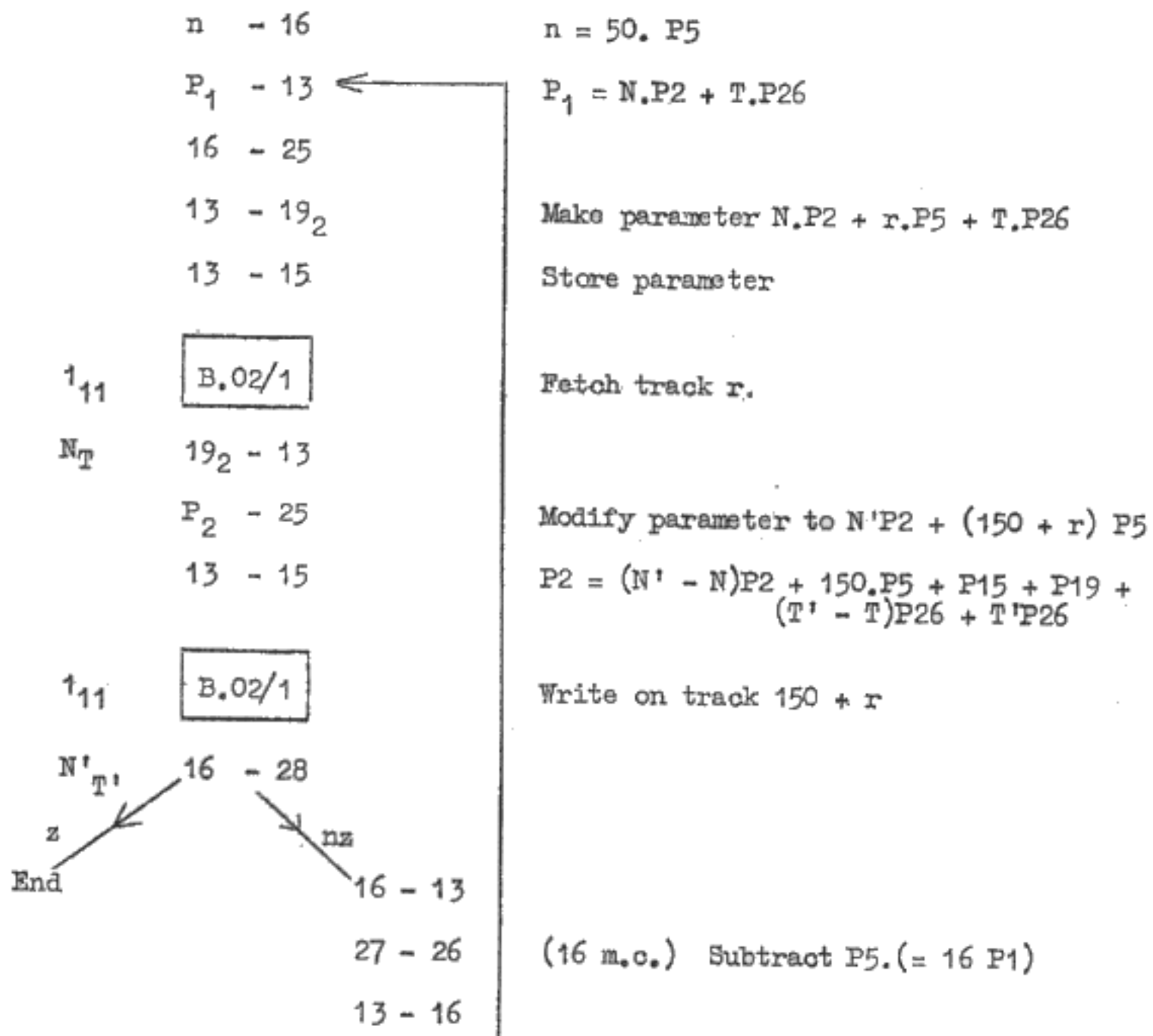
Both of these routines are of use only when we have to deal systematically with much data. If we wish to deal with only a few tracks

* Often abbreviated to M.F.C.

of the drum it is generally simpler and more economical to write out the required instruction.

Example 5 To transfer the contents of tracks 0 - 50 to tracks 150 - 200.
(This may be required in a matrix program.)

B.02/1 does not use TS 16 and is entered in 1₁₁. We might use the following sequence of operations.



The same 13 - 15 instruction before B.02/1 may, of course, be used on both occasions.

I.16.3 The drum as an instruction store

It frequently happens that a program is so large that it requires more space than is available in the high speed store. In such a case one stores as much as possible in the mercury store and has to call down the rest from the drum when required. There are two methods of doing this.

The first is of use mainly when the complete program is logically composed of only one block which is too big for the high speed store. We arrange the program in such a way that as much of it as possible stays in the mercury all the time and the remainder is stored on the drum. This part is then over-written whenever required by fresh instructions from the drum.

A good example is the auxiliary routine that will be required with a Runge-Kutta subroutine for integrating a set of differential equations. If

the set is at all complicated not all of the program can be kept in the high speed store. The auxiliary routine is required for evaluating the right hand sides of the given set of equations. It is obviously important to keep the master routine and the Runge-Kutta subroutine in the high speed store all the time, as well as much of the data that is required. The auxiliary, however, may be composed of several sections, say one for each equation, and these can be brought down one after the other. No time is wasted if the second section is being read while the first is being obeyed, etc. A useful way of ensuring this is to start each section with an instruction to read the next section from the drum. The last instruction of a section then transfers DL 11 to the appropriate place in the high speed store.

The second method is extremely useful when a large program breaks up into logically distinct sections that can occupy the high speed store in turn. Here the separate parts are treated rather like subroutines with a master routine. The master routine is written in such a way that the various component programs are brought into the high speed store one after the other, are entered at the correct point and return control to the master routine at the correct point. Various special control programs are available to facilitate the writing of large programs. One of the most useful and flexible of such programs is the Program Changing Routine, RAE 180, described in IV.2.

I.17 Input of programs

A program either is or is not short enough to be stored entirely in the high-speed store. If it is not short enough some of it will have to be stored on the magnetic drum, and it has become the general rule at R.A.E. to store the entire program on the drum. This section therefore falls into two parts, input to delay lines and input to drum: a combination of the two is possible but is not recommended.

Input of program to high-speed store

A delay line holds 32 words. If these are punched in binary they will take one row each, so 3 cards will be needed for a delay line and 4 rows will not be needed for the contents. Invariably the 32 words are punched on the last 32 rows of the triad of cards and the first 4 rows are used for other purposes. It is possible to punch a program on to these rows which will make the machine read the next 32 rows into the delay line. The method depends very largely on the effect of the zero word as an instruction. It is the instruction 0, 0 - 0, 0, 0, 0 X. Source 0 (the reader) is sent to destination 0 (TS Count) in minor cycle (m + 2) if the instruction is stored in m.c. m, and, as the instruction is stopped the machine waits for a row of a card to come into position under the reading brushes.

We first describe the technique of reading only the α -field and then discuss simultaneous input to two delay lines using both fields.

Input to delay lines 1 to 8, reading only the α -field

Suppose that the machine has just reached the instruction in 1_0 , that this instruction is 0, 0 - 0, 0, 0, 0 X and that the reader is called. (This is almost the state in which the machine is left when the Initial Input key on the reader is pressed: the exception is explained below.) A triad of cards is run into the reader and the first card has the following punching on the first 4 rows:

Y row: 0, 0 - 0, 0, 0, 0 X (i.e. nothing punched)

X row: 5, 0 - 5, 1, 26, 25 X

0 row: 5, 0 - 5, 0, 30, 31 X

1 row: 1, 0 - 5, 0, 30, 31 X

The following instructions are obeyed:

1_0	0 - 0_2 X	Y row to TS Count in m.c.2.	
0_2	0 - 0_4 X	X row to TS Count in m.c.4.	
0_4	0 - 5 X (32 m.c.)	0 row to all m.c. of DL 5, ending in m.c.31.	
5_{31}	0 - 5_{31} X	1 row to 5_{31} only.	
5_0	0 - 5_0 X	2 row to 5_0	} Read 32 words into DL 5.
5_1	0 - 5_1 X	3 row to 5_1	
	$\vdots$	$\vdots$	
5_{31}	0 - 5_{31} X	9 row of 3rd card to 5_{31}	
1_0	0 - 0_2 X		

The instruction punched on the X row is obeyed, quasi-4, when the 0 row is in position. It puts the instruction 5, 0 - 5, 0, 30, 31 X in m.c. 0 - 31 of DL 5 and immediately obeys m.c.31. The effect of the instruction in m.c. n is 0 - 5_n X with next instruction in 5_{n+1} . The instruction in 5_{31} therefore is 0 - 5_{31} X (replacing itself by 1, 0 - 5, 0, 30, 31 X from the 1 row of the card) and the next instruction is in 5_0 .

The next 31 instructions, in $5_0, 5_1, \dots, 5_{30}$, all replace themselves by words punched on the successive rows of the cards. When 35 of the 36 rows have been read the machine has reached 5_{31} , which is 0 - 5_{31} X leading to 1_0 because the 1 row of the first card of the triad was placed in it. The 36th row is read to 5_{31} and the next instruction is taken from 1_0 , not 5_0 .

When all 36 have been read the words punched on the last 32 rows of the triad of cards have been stored in the 32 minor cycles of DL 5 (in minor cycles 0, 1, ... 31 in that order) and the program ends in 1_0 with the reader called.

Obviously a very similar process will fill any other of delay lines 1 to 8, the only changes being that the 5's punched in the X, 0 and 1 rows of the first card will become the number of whichever delay line is to be filled. Since the program to fill the delay line begins and ends in 1_0 and does not affect the word in 1_0 (unless the delay line to be filled is DL 1 the triads to fill the various delay lines can be put together into a pack. The order in which the delay lines are filled is unimportant except that 1_0 must be left clear; so either DL 1 must read the zero word into 1_0 or (much more frequently) DL 1 is the last one filled. Delay lines 9 to 12 cannot be filled in this way because the machine cannot obey directly instructions stored in them, so a different method used to fill them is explained below.

If the instruction on the 1 row were changed to N, 0 - 5, 0, 30, T X (only the NIS and timing numbers have been changed) the filling of DL 5 would have proceeded in the same way until the second instruction in 5_{31} were reached. The only change would be that the next instruction would be taken from N_{T+1} instead of 1_0 . This is used to enter the program when all of it has been read into the store. The last delay line (usually DL 1) has N and $(t - 1)$ punched in the NIS and timing number positions on the 1 row (boxes are printed on every instruction card in these two places) and the program is entered in N_t .

It is worth remarking that every instruction obeyed in filling the delay line was stopped; so it does not matter whether the Normal/Stop key is at Normal or Stop.

Some old library subroutines were issued in a form where the filler instructions assumed that 8_0 was clear. This can easily be changed to 1_0 by punching a P_2 in the 1 row of the first card where it has not already been punched. For safety's sake this change should always be made as soon as a copy of the subroutine is made.

Input to delay lines 9 to 12, reading only the a-field

In the previous example to fill DL 5 instructions were placed in that delay line and later obeyed. The same cannot be done in delay line 9, say, so we shall put instructions in DL 2 each of which fills one minor cycle of DL 9.

On the top four rows of the first card of the triad we punch

Y row: 1, 0 - 2, 1, 29, 28 X
 X row: 2, 0 - 9, 0, 30, 31 X
 0 row: 2, 0 - 2, 0, 27, 28 X
 1 row: 1, 0 - 9, 0, 30, 31 X

and the instructions obeyed are:

1_0	0 - 0_2 X	Y row to TS Count in m.c.2.	
0_2	0 - 2 X (32 m.c.)	X row to all m.c. of DL 2.	
1_0	0 - 0_2 X	0 row to TS Count in m.c.2.	
0_2	0 - 2_{31} X	1 row to 2_{31} only.	
2_0	0 - 9_0 X	2 row to 9_0	} Read 32 words into DL 9
2_1	0 - 9_1 X	3 row to 9_1	
	⋮	⋮	
	⋮	⋮	
	⋮	⋮	
2_{31}	0 - 9_{31} X	9 row of 3rd card to 9_{31}	
1_0	0 - 0_2 X		

The reader should have no difficulty in understanding how this works. This method is used to fill DL 9, 10, 11 and 12: it could be used for the first eight delay lines but the method described above is normally employed.

Again, if the instruction on the 1 row is changed in the NIS and timing numbers so that it becomes $N, 0 - 9, 0, 30, (t - 1) X$ the filling of the delay line is unaffected but the next instruction is taken from N_t instead of 1_0 . However it is not common to enter a program after filling one of delay lines 9 to 12 last: the usual order of filling the delay lines is 12, 11, 10, ... 2, 1. Delay line 2 must be filled after delay lines 9 to 12 because the "filler instructions" for these delay lines put instructions into DL 2, and DL 1 is filled last because 1_0 has to be clear when any delay line is being filled. The program is therefore entered after DL 1 has been read.

The method of filling delay lines 9 to 12 again uses only stopped instructions - it works just as well with the machine "stopped" as on Normal.

There may be one or two subroutines in the library, issued in the days when 8_0 , not 1_0 , was used as the blank minor cycle, which fill some of delay lines 9 to 12. These will use DL 1 instead of DL 2 for the filler instructions, so they do not leave 1_0 clear. These filler instructions must be changed to the ones now current as soon as a copy of the subroutine is made.

Input to delay lines using both fields

It is possible to fill two DL's at the same time by using the α - and β -fields of a card. Since the β -field is read by an instruction with a 'Go' digit it is essential that the Normal/Stop key is at Normal (i.e. in the 'up' position).

A uniform procedure is possible this time for filling any combination of delay lines. Separate filler instructions are, of course, needed for DL's 9, 10, 11, 12 but in no case is it necessary to have punching on the Y row of the first card of a triad.

To fill DL 2 and DL 3 the first four rows are punched as follows:

	α -field	β -field
Y row:	Blank	Blank
X row:	1, 0 - 2, 1, 27, 26	3, 0 - 2, 0, 30, 30 X
0 row:	2, 0 - 3, 1, 28, 27	2, 0 - 3, 0, 30, 31
1 row:	3, 0 - 3, 0, 30, 30 X	1, 0 - 3, 0, 30, 31

The following instructions are obeyed (1_0 is assumed clear):

1_0 :	$0 - 0_2 X$	Y row to TS Count in m.c.2
Q_2 :	$0 - 0_4 X$	X row (α) to TS Count in m.c.4
Q_4 :	$0 - 2$ (32 m.c.)	X row (β) to DL 2, ending in m.c.0.
1_0 :	$0 - 0_2 X$	0 row (α) to TS Count in m.c.2.
Q_2 :	$0 - 3$ (32 m.c.)	0 row (β) to DL 3, ending in m.c. 0.
2_{31} :	$0 - 2_{31} X$	1 row (α) to 2_{31}
3_{31} :	$0 - 3_{31}$	1 row (β) to 3_{31}
2_0 :	$0 - 2_0 X$	2 row (α) to 2_0
3_0 :	$0 - 3_0$	2 row (β) to 3_0
.....		
2_{31} :	$0 - 2_{31} X$	9 row (α) of 3rd card to 2_{31}
3_{31} :	$0 - 3_{31}$	9 row (β) of 3rd card to 3_{31}
1_0		

The contents of DL 2 came from the α -field and the β -field goes into DL 3.

If the DL in the β -field is the last to be filled and control is to go to N_t , the next instruction source and timing number in the β -field of the 1 row become N and t - 1, as before.

Below we give the necessary filler instructions for other possible combinations of DL's. In each case the first named is in the α -field. As usual, 1_0 is assumed clear.

DL 9 and DL 10 (Fillers in DL 2 and DL 3)

	α	β
Y row:	Blank	Blank
X row:	1, 0 - 2, 1, 27, 26	3, 0 - 9, 0, 30, 30 X
0 row:	2, 0 - 3, 1, 28, 27	2, 0 - 10, 0, 30, 31
1 row:	3, 0 - 9, 0, 30, 30 X	1, 0 - 10, 0, 30, 31

DL 3 and DL 9 (Special fillers in DL 2)

	α	β
Y row:	Blank	Blank
X row:	1, 0 - 3, 1, 27, 26	2, 0 - 3, 0, 30, 30 X
0 row:	3, 0 - 2, 1, 28, 27	3, 0 - 9, 0, 30, 31
1 row:	2, 0 - 3, 0, 30, 30 X	1, 0 - 9, 0, 30, 31

DL 9 and DL 3 (Special fillers in DL 2)

	α	β
Y row:	Blank	Blank
X row:	1, 0 - 2, 1, 27, 26	3, 0 - 9, 0, 30, 30 X
0 row:	2, 0 - 3, 1, 28, 27	2, 0 - 3, 0, 30, 31
1 row:	3, 0 - 9, 0, 30, 30 X	1, 0 - 3, 0, 30, 31

All these filler instructions lead back to 1_0 after their work is complete.

Initial card

When the Initial Input key is pressed the mercury store is cleared, i.e. every word is zero. For some reason connected with the circuitry TS Count is not completely cleared; instead the word 0, 0 - 0, 0, W, T, X is left in it. There is no reason for calling any particular minor cycle m.c.O.

It is easy to get the zero word into TS Count. Doing this decides m.c.O, but we must then decide whether m.c.O is an even minor cycle or not. The machine can always distinguish between even and odd minor cycles because 0 - 24 and 1 - 24 must be obeyed in odd minor cycles. It is also desirable that TCB should be off at the beginning of a program.

One card, called an "Initial Card" is placed in front of the pack of program cards which are to be read into the delay lines. The initial card normally used is punched in the α -field as follows:

Y row:	1, 1 - 1, 0, 0, 28 X
X row:	Blank
0 row:	1, 4 - 24, 0, 0, 28 X
1 row:	Blank
2 row:	1, 31 - 21, 1, 0, 28 X
3 row:	Blank
4 row:	1, 27 - 22, 0, 0, 28 X
5 row:	Blank
6 row:	1, 21 - 28, 1, 0, 28 X
7 row:	Blank
8 row:	1, 30 - 21, 1, 0, 28 X
9 row:	Blank

Suppose that 0, 0 - 0, 0, W, T X is in TS Count in m.c. m when the Y row is read. Delay lines 1 and 8 are clear. The first three instructions obeyed are:

If $W = T$

$Q_m \quad 0 - 0_{m+T+2} \quad X \quad Y \text{ row to TS Count.}$
 $Q_{m+T+2} \quad 1 - 1 \quad X$
 $1_{m+T} \quad 0 - 0_{m+T+2} \quad X$

 If $W \neq T$

$Q_m \quad 0 - 0 \quad X \text{ Ineffective.}$
 $Q_{m+T+2} \quad 0 - 0_{m+T+4} \quad X$
 $Q_{m+T+4} \quad 0 - 0_{m+T+6} \quad X$

These three instructions get the zero word into TS Count whatever was left there after the store was cleared. We shall put $n = m+T$ in the first case, $n = m+T+4$ in the second; the instructions, from the third onwards, are

$0 - 0_{n+2} \quad X$	$0 \text{ row to TS Count}$
$Q_{n+2} \quad 4 - 24 \quad X$	TCB off
$1_n \quad 0 - 0_{n+2} \quad X$	2 row to TS Count
$Q_{n+2} \quad 31 - 21 \quad X \text{ (29 m.c.)}$	"Ones" to DS 21
$1_n \quad 0 - 0_{n+2} \quad X$	4 row to TS Count
$Q_{n+2} \quad 27 - 22_{n+4} \quad X$	Add P_1 to DS 21
$1_n \quad 0 - 0_{n+2} \quad X$	6 row to TS Count
$Q_{n+2} \quad 21 - 28 \quad X \text{ (29 m.c.)}$	Test DS 21
$\begin{array}{c} \nearrow n \\ \searrow nz \end{array}$	
$1_n \quad 0 - 0_{n+2} \quad X$	$1_{n+1} \quad 0 - 0_{n+3} \quad X$
$Q_{n+2} \quad 30 - 21 \quad X \text{ (29 m.c.)}$	$Q_{n+3} \quad 30 - 21 \quad X \text{ (29 m.c.)}$
$1_n \quad 0 - 0_{n+2} \quad X$	$1_{n+1} \quad 0 - 0_{n+3} \quad X$
	8 row to TS Count
	Clear DS 21
	Y row of next card to TS Count

If n is even the instruction $27 - 22_{n+4} \quad X$ adds P_1 to 21_2 giving 0 in DS 21 as TCB is off. If n is odd the instruction adds P_1 to 21_3 , giving 0 in 21_3 but not in 21_2 . Therefore the zero side of the discrimination is taken if n is even and the non-zero side if n is odd. In either case the last instruction (the one which reads the Y row of the next card) is in an even minor cycle of delay line 1. For then onwards this is m.c.0.

TS Count now contains a completely blank word, TCB is off and m.c.0 is even. The behaviour of DEUCE is now completely predictable.

If an initial card came between two triads of cards, so that 1_0 was already clear and the zero word was in TS Count, it would have no effect except to clear DS 21. It would not matter if the other minor cycles of DL 1 or the whole of DL 8 had been filled. This is the reason for the instruction on the Y row of the initial card.

Once again every instruction obeyed is stopped. This means that if the program is run in with the machine stopped the reading will not be affected but the program will stop on its first instruction - unless there are still cards in the reader to give single-shots. This useful facility is not available when the program is stored on the drum.

If the initial card is omitted minor cycle 0 may not be even and the word in TS Count may not be zero. Occasionally it does not matter if m.c.0 is even (an example is the "Clear Drum" program mentioned below): in such cases the first row read must be 0, 0 - 0, 0, 0, 0 X so that the zero word gets into TS Count. This is the reason why the input to DL's 1 to 8 has a blank top row.

Input of program to drum

When a program is read and stored on the magnetic drum, four cards precede the program proper. These are:

Card 0 Program R.A.E.353. Clear Drum.

Card 1 Program R.A.E.354. Set m.c.0 and clock track.

Cards 2,3 Program R.A.E.136. Input to Drum.

When the magnetic drum is used by a program it is usual to clear it at the start, as this makes testing much easier if the program is incorrect. The one card program R.A.E.353 does this in $5\frac{1}{2}$ seconds. It does not matter whether m.c.0 is odd or even to this program. At the end the program calls the reader and returns to 1_0 (which it leaves clear) to read the rest of the program pack.

The next card, R.A.E.354, has two functions. It sets an even minor cycle as m.c.0 - the method it uses is rather different from that given above, but the effect is the same - and it writes a clock track on track 15/15 of the drum. This clock track has the instruction 1, 0 - 0, 0, 0, 1 in every minor cycle except m.c. which has 1, 30-1, 1, 16, 15, 14 (P31). If the clock track is brought down from the drum into DL 1 and entered in any minor cycle it ends within 4 major cycles on 0, 0 - 0, 0, 0, 0 X in 1_0 with DL 1 clear. Nearly all organisational programs assume the clock track is stored on track 15/15, so it must be included.

The next two cards form R.A.E.136. When they have been read into DL 1 the program is entered and triads of cards are read and stored on the drum. On the Y row of the first card of a triad is punched $P_{15} + bP_{17} + aP_{21}$ if the contents of the triad are to be stored on track a/b (head position a, head number b). If nothing is punched on this Y row the program stores the contents of the triad on the track immediately following the last triad. When a P_{31} is punched on the Y row of a triad its contents are stored on the drum* but no further triads are read to the drum: instead the program ends on 0, 0 - 0, 0, 0, 0 X in 1_0 .

After the program proper is placed one further card, R.A.E.144. On this card the programmer punches details of how the program is to be entered. When the card is read track a/b is brought down from the drum, transferred to DL N and entered in minor cycle t - the instructions from N_t onwards must bring down the rest of the program from the drum. The programmer punches

aP_5 on the 4 row

bP_5 on the 7 row

and $NP_2^{\wedge}$, NP_{10} , $(t + 1) P_{17}$ and tP_{26} on the 9 row of R.A.E.144.

* If only the P_{31} is punched the contents of the triad are stored on the track following the previous triad, not on track 0/0.

$\wedge$ If N = 8 the NIS is left blank.

If the instruction on the 9 row is made a stopper the machine will halt just before entering the program.

R.A.E.144 brings down the clock track and "synchronises" the mercury store with the magnetic drum. (This is the term used to denote that minor cycle 0 is discovered from the clock track.) As it also has a blank top row, this card can be run in with initial input without doing any harm to the program. The advantage of this occurs when the program is being tested. If something has gone wrong and the program tester wishes to restart his program it is necessary to read only the last program card: it is not necessary to read the entire program pack.

Two programs often useful when programs are being tested are described in II.5. They are R.A.E.429 (Post Mortem) and R.A.E.391 (Correct Drum Instruction). When some error is suspected R.A.E.429 is used to punch out the contents of parts of the store. When an error is detected R.A.E.391 is used to correct instructions stored on the drum. These programs are fully described in Part II and are only mentioned here because they both use the clock track written by R.A.E.354. The first card of R.A.E.391 may have other uses. When read in with initial input it brings down the clock track and synchronises the two stores, ending on 0, 0 - 0, 0, 0, 0 X in 1_0 .

R.A.E.136 reads program from the α -field only. R.A.E.481 should be used if the program is punched on both the α - and β -fields and it can be used if the program is punched on the α -field only. This program performs the tasks of the three programs R.A.E.353, 354 and 136, for it clears the drum, sets a clock track on track 15/15 and reads triads of program to the drum. If the Y row of the first card, and the 32 information rows of the β -field of a triad are all blank, that triad will be treated exactly as it would have been by R.A.E.136. If not, the β -field will be treated as though it were the α -field of the next triad read by R.A.E.136.

Input to drum - Summary

The order of the cards when a program is stored on the drum are:

- | | | | |
|-------|--------------|--|----------------|
| (i) | R.A.E.353 | - Clear Drum | } or R.A.E.481 |
| (ii) | R.A.E.354 | - Set m.c.O and Clock Track | |
| (iii) | R.A.E.136 | - Input to drum | |
| (iv) | Main program | | |
| (v) | R.A.E.144 | - Synchronise and enter program - on which the user punches details of the entry point | |
| (vi) | Any data | | |

Data should never be interleaved with the program.

Input to drum - Program Changing Routine

The Program Changing Routine, R.A.E.180, is explained in IV.2. The order of the cards when it is used is:

- (i) R.A.E. 353 - Clear drum
- (ii) R.A.E. 354 - Set m.c.0 and Clock Track
- (iii) R.A.E. 136 - Input to drum
- (iv) Main program
- (v) R.A.E. 180 - Program Changing Routine - last track to be stored on drum
- (vi) R.A.E. 181 - Fill 12₂₉₋₃₁
- (vii) R.A.E. 138 - Enter R.A.E. 180 - parameter on 9 row punched by user.

All these programs except (iv), of course, have been collected together and can be found as R.A.E. 321.

If working with both fields, R.A.E. 353, 354 and 136 should be replaced by R.A.E. 481, and R.A.E. 181 and 138 can be replaced by R.A.E. 482.

This information is placed here solely for reference.

Availability of programs

When a programmer wants an initial card he takes a copy from a drawer kept for this purpose. It is not necessary to copy the card on the reproducer.

This drawer also contains cards punched in the top four rows with the "filler" instructions for the various delay lines. There are several copies for every delay line some unpunched in NIS and timing number positions (these are marked "last tank") and some with 1 and 31 punched in these places (these are marked "not last tank"). The programmer takes filler cards for all the delay lines he uses, only one of them (usually DL 1) being a "last tank" card, and punches the contents of m.c. 0 to 7 of these delay lines on to the cards. Minor cycles 8 to 31 are punched on to blank cards.

The various R.A.E. programs are kept in the R.A.E. library drawer (see III.1). A copy must be made of any program which is needed and the library copy replaced in the drawer.

Most of the R.A.E. programs mentioned also exist in older versions with rather fewer facilities. R.A.E. 134, 135 and 137 are earlier versions of R.A.E. 353, 354 and 180. The first three were also issued as DEUCE programs but their later versions have not been published.

PART II PROGRAM TESTING

II.1 The DEUCE console

This section explains the uses of the various keys and lamps on the console of DEUCE. A working knowledge of the most important knobs is essential if DEUCE is to be operated, let alone programmed: more of them must be known if programs are to be tested.

When the operator faces the main bank of switches on the console he has 3 keys and 13 lamps on the reader to his left: 101 keys, 88 lamps and

a telephone dial on the left hand panel facing him: 2 cathode ray tubes, 20 lamps, 9 buttons and 5 knobs on the right hand panel facing him: and 3 keys and 4 lamps on the punch to his right, with 12 numbering switches and 2 further keys lower down. They are illustrated in Figs. 2 and 3.

It is almost unnecessary to use the knobs on the two main panels when a working program is being operated; the only necessary keys have been placed on the reader and punch.

The Reader

There are three keys on the reader, with 7 lamps, all labelled, on the left, and 6 more lamps, arranged vertically and numbered from 1 to 6, on the right.

There are six card positions in the reader: each has a lamp which is lit when a card is in that position. Position No. 1 is the card feed at the top, position No. 6 is the stacker at the bottom from which cards are removed. These are the only positions which ought ever to have more than one card.

The cards are actually read between positions 3 and 4. Once a card has passed beyond position 3 it cannot stop until it has reached position 6, the stacker; so when the reader is not running lamps 4 and 5 will be off. Cards can sit in positions 2 and 3 indefinitely. DEUCE can only read cards if position 3 is filled and the Ready light is on.

The three keys on the reader are marked Initial Input: Stop and Run In (above and below the same key): and Run Out. All three are spring-loaded to return to their normal positions. The Initial Input has to be pulled out before it can be pushed down. The middle key can be pushed up or down. The Run Out can only be pushed down.

The Initial Input key is used only when a new program is read. It clears the entire high-speed store, calls the reader and runs the cards into positions 2 and 3 so that the machine starts to read the program. Initial Input should never be used when data cards are put in the reader.

The Stop/Run In key is generally used for running in data cards. When pressed down cards are fed into positions 2 and 3 and the Ready light comes on. Should there be a card in position 3 already, the cards do not move but the light comes on.

If this key is pushed up the Ready light goes off. There is no card movement, but if the cards are running through the reader when it is pushed up to Stop, no further cards will run through but reading of the current card will be completed.

The Run Out key is generally used only when something has gone wrong. If there are cards in positions 2 and 3 which need to be taken out of the reader, press Run Out down and they will run down into the stacker. So will any other cards in the feed, hence the feed should be emptied before Run Out is used.

The seven lights to the left of the switches are nearly self-explanatory. "Ready" comes on when the cards are ready for reading; "Running In", "Running Out" and "Reading" when these conditions occur. The "Called" light comes on after a 12 - 24 instruction and goes off after a 9 - 24 instruction. For a card to be read by DEUCE the "Ready" and "Called" lights must be on.

The "Not Ready" light is misleadingly labelled. It is not the converse of "Ready"; in fact, both can be on together! When "Not Ready" is on, the Initial Input key is disconnected and no new programs can be read. The Run In key is not affected and more data can be read by the existing program. "Not Ready" comes on when any of five keys on the console are on: they are Cont TCI, Cont TT, Cont TIL, Discrim, and Ext.Tree/Request Stop. It is unfortunate that Augmented Stop does not have the same effect.

The "Missed Card" light comes on if the reader is switched off by a 9 - 24 instruction between 40 and 80 major cycles after the last row of a card is read. In these circumstances the next card will have passed into the stacker without being read. It goes off when the Stop key is raised, and more cards are read when the same key is depressed to "Run In"; but usually the missed card together with the two cards in positions 2 and 3 will have to be recovered by the operator and replaced at the head of the pack of cards in the card feed. Missed cards are the result of bad programming. The missed card lamp has not been seen on at R.A.E. except when being tested.

The Punch

The punch has five card positions of which number 1 is the pile of blank cards and number 5 is the stacker containing the punched cards. The punch knives cut the cards between positions 2 and 3. In general therefore positions 3 and 4 contain punched cards which will not be taken out when the stacker is emptied. Unlike the reader, the punch does not have lamps to indicate cards in the various positions.

The punch is "Called", and the top one of the four lights comes on, following a 10 - 24 instruction: the call is cancelled by the 9 - 24 instruction. Punching takes place only if the punch is "Called" and "Ready"; then the "Punching" light also comes on as each card is punched.

The "Stop" key, the left hand of the three, can only be pressed down. It puts the "Ready" light off and so prevents any further cards from being punched, but punching of the current card is completed.

The punch cannot be ready unless there are cards in position 1 (the pile of blanks). This being so, it is made ready by pressing down the Run In key, which puts the "Ready" light on: and then if necessary position 2 is filled when the punch is called.

When a program has finished, the last two cards punched will be in positions 3 and 4. The operator will want these cards at the end of the pile in the stacker, so the Run Out key must be pressed down and released. Two blank cards are fed into the punch so that positions 1 to 4 all contain blanks and the last two cards fall into the stacker. This must be done at the end of every program.

There are now two blank cards in positions 3 and 4 which will be the first two cards into the stacker when the new program starts punching. The results of almost every program are therefore preceded by two completely blank cards (not punched even in the card number columns) which the operator should remove.

Below the stacker the punch has a panel with columns of four and eight circular handswitches. The eight in the right hand column all have eleven positions, 0, 1, ... 8, 9 and OFF. These eight switch settings are copied on to columns 5 to 12 of every card punched.

The column of four switches is copied on to columns 13 to 16 of the card. They are usually set to 0000 to start with: they then count the

cards punched, carrying automatically from one switch to the next, and so number the cards consecutively. These card numbering switches can be set to any initial value required: the least significant digit should be set on the bottom switch, then the next least significant digit on its switch, and so on. These switches cannot be turned backwards, and there may be carry from one switch to the next if they are turned from 9 to 0 by hand.

There are also two keys under the card numbering switches marked "Count Off" and "Reset". When Count Off is left down there is no punching from the switches, whether parameters or card numbers: this facility is rarely used. If Reset is held down the four counting switches return to 0000.

The Main Console - Panel I

Top Row

The order of the keys in the top row is:

- (i) Cont. TCI.
- (ii) Cont. TT.
- (iii) Cont. TIL.
- (iv) Discrim. On/Off.
- (v) Prog. Display - with a lamp above it.
- (vi) Release.
- (vii) Normal/Stop/Aug.Stop - with a lamp above it.
- (viii) Single-Shot.
- (ix) A telephone dial comes here.
- (x) Clear Store.
- (xi) Marginal Check - with a lamp above it.
- (xii) Clear Sync. Alarm - with a lamp above it.
- (xiii) Also a lamp marked "Mag. Alarm".

Continuous TCI is not normally used by programmers. It supplies the T.C.I. signal continuously, thus allowing a stream of instructions to enter TS Count.

Continuous TT supplies the T.T. signal continuously. In normal working this signal is on for the duration of the transfer determined by each instruction. This key is used with the Ext. Tree key to obey instructions manually.

Continuous TIL supplies the T.I.L. signal to the TIL discriminator when it is down. It is used when testing a program instruction by instruction. If a 2 - 24 instruction is used to leave a loop, the programmer will eventually want to follow the on side of the discrimination. To do this he stops the machine on the 2 - 24, puts the Cont. TIL key down, gives a single-shot and returns the key to its normal position. The Cont. TIL key should in general not be put down when the machine is running normally as it may interfere with the reading and punching of decimal cards,

but some large programs use it to enable the operator to interfere with the running of the program, e.g. in an emergency to punch out sufficient data to restart the program.

The Discrimination key has a similar function to Cont. TIL. It has three positions:

- up is marked "Off";
- horizontal (the normal position) is unmarked;
- down is marked "On".

When the program-tester wants to force the machine to take the unnatural side of a discrimination, e.g. to test the behaviour of the program when a failure occurs, he stops the machine on the discrimination instruction, puts the Discrimination key to "Off" or "On" as the case may be, gives a single-shot and returns the Discrimination key to normal. The Discrimination key should never be used when the machine is running normally as it will override all discriminations.

The Program Display key is used for punching each instruction in binary just before it is obeyed. Full instructions for using the Program Display facility are given in II.4.

The Release key is used in conjunction with Program Display. Details of its use are given in the same section.

The Normal/Stop/Augmented Stop key is usually up in the Normal position so that the machine can run at full speed. If it is put in the horizontal position marked "Stop", the machine treats every instruction as being stopped and waits for the single-shot signal before obeying the instruction. This facility is frequently used in program testing - too frequently users are seen with this key at Stop and their hand on the single-shot key obeying instructions one by one and wasting enormous quantities of time. If tempted this way they should use Program Display (see II.4).

When the Stop key is down to "Augmented Stop" every instruction is still treated as a stopper and single-shots are necessary to obey them. Instructions which are already stoppers make the red light above this key come on, and no pressing of the single-shot key will make the machine obey the stopped instruction. Instead the Release key must first be pressed: the red light goes out and a single-shot will now make the machine obey the instruction. This facility is normally, but not exclusively, used with Program Display.

The Single-Shot key, if pressed down, gives one single-shot causing a stopped instruction to be obeyed. It cannot give a second single-shot until it has swung back to its normal position. If this key is lifted up a series of single-shots is given at the rate of 10 per second.

The Telephone Dial gives a specified number of single-shots; e.g. dialling 6 gives 6 single-shots. 0 should be labelled 10 because dialling 0 gives 10 single-shots. Dialling 9 then 8 gives 17 single-shots.

The Clear Store key has to be pulled out before it can be pressed down. It clears the whole of the mercury store (except that it may not put $W = T = 0$ in TS Count) but in no way does it affect the magnetic drum.

The Marginal Check key is used only by the engineers. Programmers should always have it in the horizontal position: the red light above it will then be off.

The Clear Synchronism Alarm key is used when the red light above it comes on and a buzzer sounds. This indicates that the magnetic drum store is out of synchronism with the rest of the machine. Usually the only remedy is to stop the program, press this key repeatedly until the light goes off, and then start right at the beginning, as the machine will attempt to read from the drum or write on it even after the alarm is given.

This should not affect programs which do not use the magnetic drum but the conditions which caused the drum to go out of synchronism are likely to cause errors in the arithmetic or in the store: results should be viewed with suspicion.

When this alarm is given during a program which does not overwrite the data it stores on the drum, it may be possible to use program number R.A.E.160 to "resynchronise" the drum with the rest of the machine. Details are given in II.5.

The red light marked Magnetics Alarm comes on if the engineers use certain switches placed near the magnetic drum. It should be off before the machine is handed over to the programmers.

Second Row

There are four keys in the second row:

- (i) Read.
- (ii) Single Read.
- (iii) Punch.
- (iv) Clear Alarm.

Only the second does not have a light above it.

(On Daisy there are three keys to the left of these four. On Gert the three keys appear in the third row. They will be described there.)

Pressing down the "Read" key has the same effect as 12 - 24; pressing it up has the effect of 9 - 24. When the reader is running it is better to use the "Stop" key on the reader first; otherwise the 9 - 24 may come in the middle of a card and the user will not know how much of the card has been read.

The lamp over the Read key comes on when the reader is called and goes off when it is cancelled. It will always be in the same state as the Called lamp on the reader.

Pressing down the Single Read key is supposed to call the reader and cancel it when one card has been read. It will not work with decimally punched cards as the subroutine to read them will call the reader again.

The Punch key has the same effects on the punch as the Read key on the reader. The lamp above it will always be in the same state as the Called lamp on the punch.

The Clear Alarm key, when pressed down, turns off the buzzer and the red lamp above the key. They do not come on again unless another 7 - 24 is obeyed. If the Clear Alarm key is left in the up position the buzzer will not sound but the lamp will go on and off under the control of the program. When the key is in the horizontal position both buzzer and lamp will be controlled by the program.

Both R.A.E. DEUCES have an oscillator and loudspeaker attached to the alarm. They behave like the lamp - not like the buzzer as might be expected.

Third Row

This consists of a row of 14 lights, known as the I.S. lights, with 3 further lights, TCA, TCB and Go, some way to the right.

The Instruction Staticizer lights are divided into groups of 4, 5 and 5 by lines ruled underneath them: the groups are labelled N, S and D. They show the Next instruction source, Source and Destination of each instruction just before it is obeyed. When the machine is working at full speed it will not be possible to see every instruction, but the way in which the lights flicker may give some indication whether the program is working correctly. They are more often used when instructions are being obeyed singly. Reading them has been made easier on both R.A.E. DEUCES by making the Source lights green, the NIS and Destination lights white.

The TCA and TCB lights are on when the respective triggers are set.

The Go light is on when instructions are obeyed at full speed, but it goes off when the machine waits on a stopped instruction. It will be off when the Stop key is at Stop or Augmented Stop: it may flicker during reading and punching.

Fourth Row

This consists of:

- (i) 14 external tree keys;
- (ii) 1 characteristic key; (Characteristic 0, 1, 2 only);
- (iii) the Ext. Tree/Request Stop key;
- (iv) 3 request stop keys - on Gert only. On Daisy these appear at the beginning of the second row.

As long as the Ext.Tree/Request Stop key is in its normal horizontal position the other keys in this row do not affect the working of the machine. Application of External Tree and Request Stop are explained more fully in II.4.

The External Tree is used only with the machine stopped. Then, if the Ext. Tree/Request Stop key is down in the Ext. Tree position, a single-shot causes the machine to obey a hybrid instruction whose NIS, Source, Destination and Characteristic are set on the first 14 keys of this row, and whose Wait and Timing numbers are those of the instruction in TS Count. Thus the P1 - 16 digits of the instruction in TS Count are superseded by the external tree keys but the P17 - 32 digits are still used. If the Cont.TT key is used instead of a single-shot the instruction is obeyed continuously - which is a disadvantage if it is, say, 21 - 43 - but control does not move on to another instruction: the old one stored in TS Count can still be used.

When the Ext. Tree/Request Stop key is up at Request Stop the Characteristic key is ignored. If any of the three switches to the right of the Ext. Tree/Request Stop key on Gert (to the left of the Read key on Daisy) are not down, the corresponding sections NIS, S and D of the external tree keys are ignored. So if 2, 13 - 1 is set on the external tree keys and the NIS switch is up while the S and D switches are down, the machine stops just before obeying any 13 - 1 instruction, whatever its NIS. The Go lamp

is on. The Stop key must be put to stop; then the Request Stop can be put to normal and the machine stops on the next instruction.

The usual cycle of operations when using the Request Stop facility, starting with the machine stopped is as follows:

- (i) Set instruction of external tree switches, set request stop switches.
- (ii) Ext. Tree/Request Stop key up to Request Stop.
- (iii) Normal/Stop key up to Normal.
- (iv) Wait for Request Stop.
- (v) Normal/Stop key to Stop.
- (vi) Ext. Tree/Request Stop key to Normal.
- (vii) Then back to (i), or (ii) if the same instruction is used again.

As it is fatal to interchange (ii) and (iii), or (v) and (vi), it is much better to use one hand only for these operations. Using both hands generally leads to error.

Fifth Row

The 32 output staticizer (O.S.) lights. Rulings on the console between this row of lights and the next divide the 32 digits into the parts of the instruction word. There are also 3 screws whose heads divide the digits into 4 groups of eight - useful for recognising Scheme B code words, (IV.6).

The key at the end of the row clears the output staticizer on being pressed down and released. If it should stick down the O.S. cannot be cleared and all punching will be incorrect.

Sixth Row

The 32 input dynamicizer (I.D.) lights. The rulings and the screws between the O.S. and the I.D. lights can be used in reading either. It is the I.D. lights, not the keys underneath them, which represent what the machine may read when an instruction has source 0.

The key at the end of this row is independent of the I.D. It is a single-shot key put on to both R.A.E. DEUCES for short-armed programmers* who want to watch the monitor tubes while giving single-shots.

Seventh Row

The 32 input dynamicizer keys. Putting these down or up puts the corresponding I.D. lights on or off. The key at the end of the row clears all the I.D. lights if pressed down: if it is left in the up position the I.D. lights will stay on only as long as the keys are depressed.

Eighth Row

A set of 32 switches with one key to their right, connected to the I.D. They are used chiefly by the engineers who set a standard pattern on

* Programmers of normal size?!

them: If the key is put up (marked +) the number set on the switches is placed on the I.D.; if the key is put down (marked -) the ones-complement of the number on the switches goes to the I.D. Thus if the key is put up and down every I.D. light is put on.

The Main Console - Panel II

Top of Panel

At the top of the panel are four pairs of lamps, red and green; from left to right they are labelled, Fan, Heaters, H.T., Hollerith. The green lamp indicates that this part of the machine is switched on, the red lamp that it is switched off. The switches used are the eight buttons under the lamps marked Off and On. (The two middle Off buttons are also used for resetting fuses.) To the right of these are 12 red lamps which light according to electrical failures.

The switching-on procedure is:

- (i) Fan On.
- (ii) Heaters On.
- (iii) Wait for one minute.
- (iv) H.T. On.
- (v) Hollerith On.

This should normally be done by an engineer.

The switching-off procedure is:

- (i) Hollerith Off.
- (ii) H.T. Off.
- (iii) Heaters Off.
- (iv) Fan Off.

The lazy man's way is:

- (i) Fan Off.

In no case need any other buttons or switches be pressed. This applies to the buttons on the fan's own switch-boxes.

If a fuse* blows part of the machine will be switched off and one of the 12 red lights comes on. Make a note of which fuse has blown, press the appropriate "Off & Reset" button and then try to switch on again; but an engineer should be called first if one is available.

Bottom of Panel

The lower half of Panel II contains two monitor tubes, each displaying rows of 32 dots. The left hand tube has four single rows, then three pairs of rows and finally two sets of four rows at the bottom. These display the contents of 13, 14, 15, 16; 19, 20, 21; 17, 18. The right hand tube displays 32 rows and can show the entire contents of one delay line.

Each tube has two knobs marked Focus and Brilliance underneath it. The Brilliance knob changes the contrast between the ones and noughts

* "Fuse" is the term always used, but in fact they are resettable cut-outs.

(brighter and fainter spots) in the display. When the Focus knob has to be adjusted it is better to turn the brilliance up first.

The left hand tube has a three position switch below it. In the left hand position the display is split between P_4 and P_5 , P_9 and P_{10} , P_{14} and P_{15} , P_{16} and P_{17} , P_{21} and P_{22} , P_{25} and P_{26} , and P_{30} and P_{31} . This enables instructions to be recognised more easily. In the centre position there is no splitting of the display. In the right hand position the display is split between P_5 and P_6 , P_{10} and P_{11} , P_{16} and P_{17} , P_{20} and P_{21} , P_{25} and P_{26} and P_{30} and P_{31} to enable binary words to be read more easily.

The right hand tube has a 14-position rotary switch between the Focus and Brilliance knobs. This enables the user to choose which of the 12 delay lines to display. The remaining two positions are TS Count and Off. TS Count is given a 32 word display in which the NIS, Source, Destination, Characteristic and Joe digits are always the same, but each line has Wait and Timing numbers one less than the row above. If the instruction entered TS Count from a delay line, the original form will be displayed on the same line as the instruction in its delay line. Digits 1, 16, 25 and 31 of TS Count are not displayed.

One button labelled "Minor Cycle Slip" is placed in the line of knobs between the two monitor tubes. Every time it is pressed the display in the right hand monitor slips down two rows, and the display of QS 17 and 18 in the left hand monitor also slips two rows. The reason for this facility is that there is no guarantee that the even minor cycle chosen to be minor cycle 0 when the program is read in is the even minor cycle displayed at the top of the right hand monitor. It is useful to make it so; therefore the program tester presses the button until it is in the right place. This can often be done while the magnetic drum is cleared or while the program is read in by R.A.E.136. Later one can use the 13 - 0 instruction so often stored in 1₂₈ or the fact that most links are planted in 1₃₀. Once the monitors have been "lined up" the form of the instruction shown in m.c.30 of the TS Count display has absolute wait and timing numbers: the wait number is the number of the minor cycle in which transfer starts and the timing number is the number of the minor cycle of the next instruction.

II.2 Common errors in programs

It is not possible to list all the errors that can occur in the writing of programs. Mistakes, however, do occur only too frequently and some account of the more common ones is necessary. They may be divided into four classes.

(a) Incorrect punching This should never happen (ideally). If programs are punched in binary they should always be checked with the 'Check Detailed Coding' program (see II.3). It is only too easy to punch in the wrong column of a card. A very common fault is to punch 13 instead of 11 (1011 instead of 1101) and vice versa. The same applies to 26 and 22.

Cards that are out of order can cause much heartbreak and frustration. The order should always be checked - all cards should be numbered, by punching on the cards, not just in writing.

(b) Coding errors Many of these can be avoided by careful checking and by use of the detailed coding program (see IV.1 and Ref.7). Even so, some mistakes do occur.

Very often loops are made to return to the wrong instruction. The machine often appears to go into a (small) loop of instructions or will omit sections of the program.

Many errors that occur are connected with the use of subroutines. They are almost always due to careless reading of the operating instructions. Links may be coded incorrectly, especially with second and higher order subroutines. The link may be in the wrong place before the subroutine is entered. This is spotted because the link does not appear in 1₃₀ (or wherever else it should be). Links are sometimes forgotten altogether and the same can happen with a subroutine. In particular, with higher order subroutines it is not difficult to forget to insert the low order subroutines. Subroutines often require parameters, such as number of decimal digits or terms of a power series. These must be inserted in the program in the program according to the specification of the subroutine.

Many subroutines assume the existence of an instruction like 13 - 0 in 1₂₈. The programmer must make sure that the subroutine's assumption is correct.

There is a general convention that TCB is to be off when entering a subroutine. This should always be checked. When a subroutine uses TCB it must always be cleared before leaving the subroutine.

Frequently when instructions concerned with fetching or storing in consecutive minor cycles are modified, the reference to the variable minor cycle is not correct. Calling for the wrong minor cycle when fetching a link or a parameter also occurs, but is not so common.

Discriminations need careful checking. The two alternative routes must begin in consecutive minor cycles. Long transfers present another trap for the unwary. It is very easy to make a mistake and make the transfer go on for too long, or too short, a time. If a double length shift is to be for 2n minor cycles, starting in an even minor cycle, then it must end in an odd minor cycle. This is often forgotten when repairing a program, as instructions are moved willy-nilly to convenient spaces with little or no thought for the possible effect on other parts of the program.

Fetching information from the drum can lead to many mistakes if not enough care has been taken. It is easy to fetch the wrong tracks, especially if only one or two are to be brought down. The reason is generally that the reading heads are in the wrong position - they have been moved and left there, instead of being brought back to where they are wanted. (The same remarks apply to the writing heads.) Mistakes of this nature, especially when concerned with reading fresh instructions can often be spotted fairly quickly. If the program 'falls out', i.e. encounters a zero instruction during the program, at a time when new instructions are supposed to be read from the drum, DL 11 is generally empty showing that a zero track has been read.

Less common, but still worthy of mention, are errors concerned with the multiplier or divider. These must never be started in even minor cycles and both must be allowed sufficient time to complete their jobs. Mistakes due to calling for a product or quotient before it is complete are rarely found by testing with the machine 'stopped' as the multiplier/divider then has enough time. It can happen that the result is taken from the wrong half of DS 21.

Too much computing between cards or between rows while the reader or punch is running is noticed reasonably easily, because faulty reading or punching of cards occurs. Either information never reaches the correct part of the machine or is not punched on the cards.

(c) Logic and book-keeping These are among the most troublesome to find and correct.

Mistakes of logic, when operations have been done in the wrong order, generally require wholesale revisions of the program.

A common source of error is counting. Loops are frequently traversed once too often or once too few times. The lessons of I.5 and I.8 must be learned.

Careless reading of operating instructions for subroutines leads to some book-keeping errors. Subroutines in general require some working space - quantities that are required for further use should not be left in positions that are destroyed by the routine.

It is easy to have a discrimination the wrong way round. The two sides may, in general, be interchanged quite simply but care must be taken if other instructions can lead to these. The programmer should also remember that zero is a positive number - a downward count goes once more if the test is with destination 27 than it does with destination 28.

(d) Scaling These faults are due to quantities becoming more than single length. Symptoms are failures in division subroutines, attempting to take square roots of negative numbers, complete rubbish when punching out results, etc. It is not easy to correct such errors and, strictly speaking, they are not programming faults. The mistake lies in the numerical method of dealing with the particular problem. One way out of the difficulty is to use a systematic method of scaling, such as floating point arithmetic, discussed in IV.3.

The types of mistake given above are not exhaustive. All of them, however, do occur and are made by most programmers at some time or other. For the beginner let it be said that subtlety is out of place until the art of programming has been mastered. It is better not to tinker with library subroutines to make them do something slightly different - the results are often disastrous.

As experience of programming increases the incidence of mistakes drops considerably. Recognition of symptoms and their interpretation comes only with practice. There are many aids to program testing which can lighten the task of error correction. These are described in succeeding sections.

II.3 The use of checking routines

Check detailed coding

When the detailed coding of a program has been made and punched in binary it is essential to check both operations. They are very frequent sources of error: to keep them down to one arithmetic mistake and one wrong punching in a delay line is the exception not the rule.

A program has been prepared to read a program punched in binary and punch each instruction in decimal with some extra information. The program is DEUCE library program ZPO6/1 (No.213), R.A.E.148. On each card it punches out the following information for one instruction:

The delay line in which the instruction is stored.
 The minor cycle in which the instruction is stored.
 The P1 digit of the instruction.
 The Next Instruction Source delay line.
 The minor cycle of the next instruction, $m + T + 2 \pmod{32}$.
 The Source number.
 The Destination number.
 The Characteristic.
 The Wait number.
 The Joe number.
 The Timing number.
 The P31 digit.
 The Go digit.
 The first minor cycle of transfer, $m + W + 2 \pmod{32}$.
 The length of transfer.

A special board has been plugged to list the cards on the Hollerith 805 machine and special ruled sheets for this listing have been printed with various column headings.

The program reads in three cards with one delay-line's worth of instructions. The number of the delay line is taken from the X row of the first card and the 32 cards it punches are numbered from 0 to 31, giving the minor cycle numbers. It can be seen that the program punches out every digit of the instruction in decimal so that a comparison between the listing and the original detailed coding sheet should pick up any mistakes in the binary punching. In addition, however, the program calculates the first minor cycle of transfer, the length of transfer and the minor cycle of the next instruction, and it also punches these in decimal. This means that it is possible to reconstruct the flow diagram from the listing. This is a very good check: an error very rarely slips through it, and the programmer quite frequently notices mistakes in the logic of his program at this stage instead of wasting time finding them by trial and error on the machine.

The check detailed coding program has one further facility. The machine stops on the instruction 1, 0 - 14 X before reading each triad of cards. The user sets a word on the I.D. and the machine reads this when given a single-shot. It then reads the 3 cards and punches out 32. If the Pr light on the I.D. is on the program treats the word in minor cycle r as a binary number instead of an instruction (P32 is used for minor cycle 0), and the number is split into groups of 5 digits and punched in the scale of 32. Numbers are easily recognised in the listing because the length of transfer column is blank in their minor cycles. This facility provides a check on the punching of binary numbers.

Check decimal punching

This program, R.A.E.22 - not issued in the DEUCE library - is intended to be used for checking the data cards punched for a working program, but it

is probably best described here.

It reads one parameter card and then checks all the data cards it is given. On the parameter card the user punches:

Y row: Columns to be checked.

X row: Columns which should have a sign punched.

O row: Columns which should have a decimal digit punched.

The possibilities which the program can check are:

- (i) No punching in a column. Punch Y on parameter card.
- (ii) Sign only in a column. Punch Y, X on parameter card.
- (iii) Digit only in a column. Punch Y, O on parameter card.
- (iv) Sign and digit in a column. Punch Y, X, O on parameter card.

The program ignores all columns where the Y row of the parameter card is unpunched.

If all the data cards are correct the program will read them all and punch nothing. If it finds an error it immediately stops the reader, punches one card and then starts reading again*. The card punched gives information about the type of failure.

On the Y row it has the number of the card in binary - the program numbers the cards it reads from 0 upwards. The rows between X and 4 inclusive are punched in the columns where errors occur in the following way:

X row: where two signs are punched.

O row: where one or two signs are punched and none expected.

1 row: where no sign is punched and one is expected.

2 row: where two or more digits are punched.

3 row: where one digit is punched and none expected.

4 row: where no digit is punched and one is expected.

II.4 Machine aids to program testing

This section deals with facilities available for program testing insofar as these are built into the machine. The next section is concerned with programmed aids.

* If the instruction in 38 is made a stopper the program will stop 1, 12 - 24 X after reading an incorrect card and punching the failure card. The user can then pick out the card in error, as it will be the card just read. A single-shot will cause the program to continue reading.

The two monitor tubes on the console display the contents of the temporary stores (i.e. TS's, QS's, DS's) and any one of the delay lines or TS Count. In general, lengthy inspection of the monitors is to be discouraged and avoided. Intelligent use of the display facilities, on the other hand, can make for rapid diagnosis of faults. The DL monitor display does not necessarily have minor cycle 0 at the top. It is generally advisable to press the minor cycle slip button repeatedly until what is known to be in minor cycle 0 is on the top line of the monitor. Naturally, any other minor cycle can be used to line up the display. It is easiest to use something which should be either at the top or bottom - lining up on a not easily recognisable instruction in the middle of a DL is troublesome. Often 13 - 0 in 1₂₈ provides a convenient instruction - it should be the fourth line from the bottom of the DL 1 display. The clear drum and write on drum programs are quite useful for lining up. 'Clear drum' leaves one or two minor cycles clear at the top of DL 1 (it is read before even minor cycles have been found and hence there may be some ambiguity), 'write on drum' which comes after even minor cycles have been fixed leaves one blank minor cycle at the top of DL 1.

The simplest and worst way of testing a program is to stop the machine and then proceed with single-shots from one instruction to the next. This method wastes much time and not much is accomplished during a period of program testing. Naturally, the method can be used if only a few (less than six, say) consecutive instructions are to be checked. The machine is made to halt on a stopper instruction. Any instructions that are made stoppers for the purposes of program testing should be clearly marked, so that these stoppers may be removed before the program is declared correct.

A convenient method that allows inspection of what has happened is 'Program Display'. The key marked 'Normal-Stop-Augmented Stop' is put in the augmented stop position and the key 'Program Display' is then depressed. The amber lamp above the 'Program Display' key comes on and the machine punches out, in binary, the instructions as they are obeyed. This punching stops when the machine encounters a 'stopper' instruction. A red lamp is then alight on the console. To continue, press the 'Release' key, which switches off the red light, and then press the 'Program Display' key again.

Sometimes the program display stops when the instruction is not a stopper. Examination of the console shows that the punch is no longer called and the red light is off. The last instruction of the display will be found to have been 9 - 24. Program display may be continued by pressing the 'Punch' key. Neither 'Program Display' nor 'Release' is necessary. Occasionally there is a complication with TIL on program display. If a program looks for TIL (from a card or from the console) at some point, this may be supplied by the last row of a program display card passing through the punch.

Program display should not be used in some multiplication and many division subroutines as they depend on extracting a partial result from DS 21 while multiplication (division) is in progress. This also applies to TO1/2 (DEUCE Subroutine 195). The specifications of the subroutines will state whether the subroutine can be used with the machine stopped. (For these purposes using Program Display counts as stopping the machine.) The operation of the machine is slowed up considerably by program display (20 instructions per second instead of a normal rate of some 8000 or more instructions per second) and multiplication (division) will have finished before the next instruction arrives.

For ease of reading, program display should either be done on instruction cards or copied on to instruction cards. A cheaper alternative is to use a 'mask', which is an instruction card with holes punched everywhere

in the DEUCE field. The mask is placed above the program display cards. The 32-bit words are then conveniently split up into instruction words by the rulings on the mask. Note that program display only gives the instructions that have been obeyed, never the contents of any stores.

A useful facility is the 'Request Stop'. This may be used when large parts of a program are known to be working. The particular instruction, on which it is desired to stop, is set up on the external tree keys, the NIS, S, D request stop keys are put down and the External Tree/Request Stop key is put up. The machine will stop when this instruction is encountered. When the machine stops, the 'Normal/Stop' key is put in the stop position and only then is the request stop key put back to normal. The machine will now be on the instruction following the request stop. A common mistake is to set up the address of the instruction on the NIS keys, instead of the address of the following instruction. If desired the request stop can be used whenever a particular NIS, Source, Destination or any combination of these occurs. For example, when the NIS key is the only one depressed, the source and destination on the external tree are ignored. This facility is useful when it is desired to test a new subroutine - the program is stopped the first time the next instruction comes from a particular delay line. Another example is when discrimination instructions are being examined. The external tree destination keys are set at 27 (or 28 according to circumstances) and the destination key is depressed. The machine will stop every time D 27 (or D 28) occurs.

Request stop should not be used with 0 - 24 and 1 - 24 instructions. On these particular instructions the machine continues to multiply (divide) and gets the wrong answer*.

During testing it is essential to be able to examine both possible routes after a discrimination instruction. The 'Discrimination' key on the console can assist in this. In its normal position it has no effect; when it is up, i.e. in the 'off' position, every discrimination will take the natural (i.e. earlier) side regardless of the number that has been sent to D 27 or D 28. Similarly, when the key is down the later instruction is always taken as the next instruction. The Cont TIL key does not affect the discrimination key. Care is needed when using the discrimination key together with Request Stop on a discrimination. Remember that the machine proceeds to the next instruction when the External Tree/Request Stop key is put back to normal. By then it is too late to force the discrimination one way or the other. The discrimination key must be set before the Request Stop is cancelled.

It is often useful to be able to re-enter a program that has fallen out, or to go to a particular place in a program. If the monitor display is correctly lined up, examination of TS Count and use of the External Tree key can help. The following procedure is helpful.

(i) Stop the machine. Put all the External Tree keys level, and External Tree on. Set wait and timing numbers of 29 on the I.D. Now give single-shots until the top line of TS Count has $W = T = 28$. Overshooting does not matter, just continue until the wait and timing numbers are correct.

* For curious reasons of timing in TS Count, the machine starts obeying all 'Trigger' instructions just before the beginning of the appropriate minor cycle. Request Stop suspends the operation of every instruction but only starts at the beginning of the minor cycle. MULT and DIVIDE are therefore started before Request Stop becomes operative and do not go off at the end of the 65 or 66 minor cycles. The same is true for all other Trigger instructions, but with them no harm is done by using Request Stop.

(ii) Any instruction that is now set up on the I.D. keys will be obeyed Q_{30} . If one wishes to re-enter at N_T , set $N.P2 + T.P26$ on the I.D. (Set $W \neq 0$ if $T = 0$). Give a single-shot with external tree on. The instruction on the I.D. is now in TS Count. It also appears on the I.S. lights.

(iii) Set external tree to normal and give a single-shot. The machine is now at the instruction in N_T .

This method can be used to insert constants or new instructions into particular minor cycles but is not advised, as it is easy to make a mistake and there is no record of what has been done.

Examination of tracks of the drum is somewhat troublesome. Here the external tree keys are used together with the Cont TT key. To examine track a/b the following procedure is used.

(i) Stop the machine.

(ii) External tree on. Characteristic 0. Set

a - 31

on the keys. (The NIS is irrelevant.) Depress the Cont TT key.

(iii) Set

b - 30

on the keys and depress Cont TT.

The contents of a/b are now in DL 11. A similar procedure is used to write on a given track, except that the characteristic must be 1. If a program is to be re-started the heads must be returned to the positions they were in before the tracks were examined. As no single-shots have been given during this examination the machine is still at the same place in the program. In the above procedure DL 11 is lost. Should the contents be required, they must first be written on the drum on an unused track.

The Cont TT key may be used to insert quantities into every minor cycle of a store. Thus

(i) Stop the machine.

(ii) External tree on. S - D on the keys and depressing the Cont TT key has the effect of transferring the whole of S to the whole of D. For example, 28 - 10 would have the effect of placing a P17 in every minor cycle of DL 10. The instruction 11 - 10 would transfer every minor cycle of DL 11 to the corresponding minor cycle of DL 10.

Here again we may re-start the program at the place where it stopped because no single-shots have been given. Cont TT will not work with any instructions with D 24.

II.5 Programmed aids

Some routines have been made to help in the testing of programs, chiefly at R.A.E. The ones described in this section all assume that there is a clock track on track 15/15 of the drum of the type planted by DEUCE program ZP 15 (No. 37), R.A.E.135, or by R.A.E.354.

Post Mortem

There are two versions of this program, R.A.E.345 and R.A.E.429. The only difference is that R.A.E.429 punches information on both fields, whereas R.A.E.345 only uses the α -field. Only R.A.E.345 will be described here. The Post Mortem, R.A.E.345*, was formerly known by the vague title of "Program Testing Aid". It is used to punch out the contents of sections of the mercury and magnetic stores after the program has "died", whether by "falling out" or by being stopped at a strategic point by the tester.

It is usual to clear the magnetic drum before reading in a program to be tested. This has the advantage of ensuring that everything punched by the Post Mortem routine has been stored by the program being tested. In addition the Post Mortem routine is arranged to ignore blank tracks, thus saving much time.

The most awkward part of the operation of this program is reading it in. The initial input key on the reader cannot be used because this clears the high-speed store, yet one of the objects of the exercise is to punch the contents of this store. Instead the first nine operations in the following table are needed.

- (i) Stop key to Stop.
- (ii) Clear Input Dynamicizer.
- (iii) Set 0, 0 - 0 with characteristic 1 on the external tree keys, (i.e. all these 14 keys horizontal).
- (iv) Ext.Tree/Request Stop key to Ext.Tree.
- (v) Single-shot.
- (vi) Ext.Tree/Request Stop key to normal.
- (vii) Stop key to Normal. (This is easy to forget.)
- (viii) Press "Read" key on the console.
- (ix) Run in the 8 program cards of R.A.E.345.
- (x) The machine reads 5 cards and stops on 1, 0 - 15 X.
- (xi) Set P_{32} on I.D. if short stores are to be punched; set any combination of $P_1, P_2, \dots, P_{12}$ to show which delay lines are to be punched.
- (xii) Single-shot.
- (xiii) Machine punches out contents of requested stores; then reads last 3 program cards and stops on 1, 0 - 14X.
- (xiv) Clear I.D. If non-zero tracks between a/b and y/z are to be punched set $bP_1 + aP_5 + zP_{17} + yP_{21}$ on I.D.
- (xv) Single-shot.
- (xvi) Machine punches out contents of all non-zero tracks in the required range and stops on 0, 0 - 0 X in 1_0 with the reader called.
- (xvii) If the user wishes, the last 3 cards can be read again, steps (xiv) to (xvi) repeated and a further section of the drum store punched.

*An earlier version, R.A.E.145, which differed only in the section for punching the magnetic drum, was published as DEUCE library program ZP 26(Nb.155)

If P_r is set on the I.D., delay line r is punched with rP_5 on the Y row, three blank rows and the 32 words in their usual positions on a triad of cards. Any or all of the 12 delay lines can be punched.

If P_{32} is set on the I.D. the short stores are punched before the delay lines. The first 4 rows are blank, then come the contents of 13, 14, 15, 16, 17₀, 17₁, 17₂, 17₃ on the first card, 18₀, 18₁, 18₂, 18₃, 19₂, 19₃, 20₂, 20₃, 21₂, 21₃ and two blank rows on the second card and a blank third card.

The contents of tracks of the magnetic drum are also punched on triads of cards in the usual positions. Track p/q has $qP_1 + pP_5$ punched on the Y row of the first card and every track has a P_{17} punched on the X row to distinguish them from the contents of delay lines.

If none of the high speed store is to be punched the first two cards of the routine must still be read (though the initial input key can now be used). It is simpler to read the first 5 cards, set nothing on the I.D. at step (xi) and give a single-shot. The routine will go on to step (xiv) without punching anything.

The routine destroys one word in D.L.1, making it zero so that the rest of the program can be read in. Which word is lost cannot easily be forecast because it depends on the position of the instruction at which program-testing stopped.

The routine also destroys the contents of 2 tracks of the drum because it has to store DL 1 and 11. As the cards are punched the routine uses tracks 0/0 and 0/1, but, if these tracks are used by the program being tested, the user can change them to any two tracks under the same head position, a/b_1 and a/b_{11} , say. The changes needed are:

Change source 0 to source a in card 0, row X and card 1, row 8.

Change source 0 to source b_1 in card 0, row 4 and card 4, row Y.

Change source 1 to source b_{11} in card 0, row 2 and card 4, row X.

Correction of instructions on drum

This routine, R.A.E.391, was made to facilitate the corrections necessary after a mistake had been found with the Post Mortem routine. It was later realised that it could also be used to set parameters in a program during input. The latter use is explained at the end of this subsection.

This routine is intended for use with programs whose instructions are all stored on the drum before the program is entered. It assumes that the usual clock track has been stored on track 15/15 by R.A.E.354 or ZP 15. (The Post Mortem routine, R.A.E.354 only disturbs tracks 0/0 and 0/1 on the drum, so providing no instructions are stored here, the programme will be intact on the drum and ready to be re-entered.)

If some instructions in the program have to be altered the four cards of this routine are read in with the initial input key. The first card merely finds the location of minor cycle 0; the next three read the routine into DL 1 and start to obey it. The routine then reads a series of data cards, each card causing one instruction on the drum to be modified, until it reads a data card with any punching on row 9. After making the modification specified by this card the routine "falls out" in 1₀ with delay line 1 clear

and the reader called. The last cards of the original program, the ones which call down the program and enter it, can now be run in and the modified program will be entered.

The correction cards are punched on the first three rows only, except that the last card has anything punched on the 9 row, as already explained. The Y row specifies which word is to be modified, the X row which digits of the word are to be changed, and the O row gives the modification.

Two kinds of modification are possible: specified digits of the existing word can be replaced by the corresponding digits of the word on the O row, or the word on the O row can be added to the existing word. The actual operation of the routine is that it reads the word specified by the Y row from the drum zeroes the digits specified by the marker in the X row, adds the O row and replaces the word on the drum. An example will make this clear.

If the X row has the wait number digits, digits 17 to 21, punched and the O row has only P₁₈ punched, the existing word N, S - D, Ch, W (J) T has its wait number replaced by the wait number of the O row and becomes N, S - D, Ch, 2(J) T.

If the X row is blank and the O row has P₁₈ punched the existing word has the O row added to it and becomes N, S - D, Ch, W + 2, (J) T.

If the X row has every digit punched and the O row has P₁₈ the existing word is replaced by the word on the O row and becomes just P₁₈. Complete replacement of a word is a common correction; a P₃₂ can be punched on the Y row instead of every digit of the X row being punched in such a case.

The specification of the data cards is

- Y row Punch $aP_5 + bP_{10} + rP_{17}$ if minor cycle r on track a/b is to be modified. (Note the order of a and b.) Punch also P₃₂ if the word on the O row is to replace the existing word completely.
- X row Punch those digits to be replaced by the corresponding digits of the O row. (Ignored if P₃₂ punched on Y row.)
- O row Punch the modifying word.
- 9 row Punch anything to indicate last correction card.

Other rows are ignored.

This method of making corrections to a program already in the store has the twin advantages that it is quicker than reading in the whole program again and that a record exists of the corrections made. It is not necessary to make the corrections in the cards of the program after each period of program-testing time on the machine. Instead, a copy of the routine* can be placed in the existing pack of program cards immediately after the program has been read to the drum and before the cards which cause the program to be entered. When corrections have been made to the program stored on the drum

* The first card of the four is not strictly necessary in this case but no harm is done in including it.

the correction cards can be inserted after the copy of the routine among the program cards so that, next time the program is read into the machine these corrections to it are made automatically. The machine time wasted is negligible.

When this is done it is recommended that a special card punched only on the 9 row should be kept handy. This card adds 0 to m.c.0 of track 0/0, so its only noticeable effect is to end the reading of correction cards. When this card is used no other correction card need be punched on the 9 row and so the possibility of leaving the routine before all correction cards have been read can be avoided.

If the routine is included in the pack of program cards the programmer can save himself the trouble of punching parameters into sub-routine cards: he can use this routine instead and "correct" the program by inserting the necessary parameters from correction cards each time the program is read.

Resynchronise drum

When the synchronism alarm goes the only remedy, in most cases, is to start the program all over again. This can be most infuriating.

In a program which computes for a long time it may well be worth adding a piece to the program so that, every so often, all the data necessary for restarting the calculation is punched out. (This also involves another piece of program to read the data back in if the program does have to be stopped.) Such punching (which would probably be in binary) would occur at some fixed interval, perhaps every 20 minutes. It is convenient to make it possible to call for this punching at other times, e.g. by setting a P₃₂ on the I.D. keys.

There is a program, R.A.E.160, which can be read after the drum has gone out of synchronism. It uses the clock track on track 15/15 to find out how much the drum has gone out of synchronism, and then re-writes every track of the drum in synchronism. It ends on the instruction 0,0-0,0,0,0 X in 1₀ with the reader called.

However it is difficult to know where to restart. If the program overwrites some of its data it will be impossible to start any later than the part of the program which stored the first data there. The high-speed store is cleared when R.A.E.160 is read in.

There is also the possibility that the magnetic drum store may have failed at the same time as it went out of synchronism. R.A.E.160 should therefore be used only as a last resort.

PART III AVAILABLE PROGRAMS

III.1 The DEUCE library

The groups which operate the various DEUCES each have their own "library" of programs and subroutines. When a new program has been constructed it is given a serial number; e.g. R.A.E.301 at Farnborough N.P.L. 75 at Teddington. When it has been fully tested a copy of the cards should be added to the drawer containing all local programs. In the case of programs this should also be the time when the programmer writes a set of operating instructions.

Usually subroutines are of more general use than complete programs. They are commonly tested in a version stored in DL 2 for a subroutine using only one tank: DL 2, 3, 4, ... are used for longer subroutines. When fully tested the subroutine is copied with DEUCE library program No.4 so that versions are made starting in DL 3, 4, ... After this process of "copying into all tanks" the specification of the subroutine is written. The specification and the cards are then sent to the English Electric Company at Stafford for publication. They allot two numbers to the subroutine, a serial number and an alphanumeric classification: for instance, R.A.E.142 became DEUCE subroutine No.148, P11D/1. The specification, flow diagram and detailed coding of the subroutines are printed in a standard form. Copies of these and of the cards are sent out to all groups operating DEUCES, and the subroutine has been published in the DEUCE library of subroutines.

There is also a DEUCE library of programs, though of course a smaller proportion of complete programs are of sufficiently general use to warrant publication - most DEUCE library programs are either engineers' test programs or linear algebra bricks. They also have a serial number and an alphanumeric classification. The difference between the alphanumeric classification of subroutines and programs is that the former start with one letter (e.g. D 01) the latter with two (e.g. CW 04). As the same serial number can apply to a library subroutine and a library program which are entirely independent, it is better to refer to the alphanumeric classification and add the serial number in brackets: e.g. "D 01 (subroutine No.34)".

Since not all subroutines and programs are eventually published there is a residue left in the local library - this assumes that someone has the responsibility for reading out the published routines from this library - and there will also be several routines in it which are awaiting publication. It is usual to send copies of routines to other DEUCE users at the same time as they are sent to English Electric so that they can be used in the period between the end of testing and publication. For this reason there are four libraries at R.A.E.:

DEUCE library subroutines.

DEUCE library programs.

R.A.E. library.

N.P.L. library.

DEUCE library subroutines

The classification of a subroutine consists of

- (i) a single letter specifying the class of the subroutine;
- (ii) a two-digit number which is the serial number in that class;
- (iii) possibly one, two or three more letters;
- (iv) possibly a stroke, followed by one more digit.

For example P.11D/1 indicates that the subroutine is in class P (punch subroutines) and is the eleventh of the class; the D shows that the subroutine operates on double-length numbers and the terminating 1 that it is the first modification of P.11D.

The classes of subroutines now in use are:

- A Programmed arithmetic: chiefly with floating-point numbers.
- B Continuous Store.
- D Division
- E Exponentials and Logarithms.
- F Fractional powers: entirely square roots at the moment.
- G Differential equations.
- J Miscellaneous functions other than E and T; chiefly Bessel functions.
- L Linear algebra and vector operations.
- M Multiplication.
- N Numerical methods.
- P Punch.
- R Read.
- S Summation of series.
- T Trigonometrical and Hyperbolic functions.
- V Three-dimensional vectors.
- Z Miscellaneous.

Two classes have been allotted letters but have not yet been used:

- C Checking.
- Q Quadrature.

After the first letter comes the two-digit serial number in the class. It is usual to quote the non-significant zero: D 01 rather than D 1.

The letters which may follow this number are:

- D Double-length numbers.
- H High accuracy: numbers more than double-length.
- F Floating point numbers.
- C Complex numbers.

The first two are incompatible, but it is conceivable that three letters could follow the number.

A subroutine should be issued with a terminating digit only if it performs the same function as the original subroutine using less storage space or taking less time; preferably both. This has not been strictly obeyed, a bad example being the issue of subroutines to punch a fraction and an integer, completely different processes, as P 01 and P 01/1.

As mentioned earlier, every library subroutine also has a serial number: e.g. P 01 and P 01/1 are subroutines No.12 and 133.

Over 200 library subroutines have now been issued. To help those who have not used DEUCE subroutines before, the conventional methods of describing them and using them are given in III.2. Section III.3 gives a short list of recommended subroutines.

The documentation of a subroutine consists of

- (1) cards giving the information listed in section III.2, and
- (2) sheets giving the flow diagram and detailed coding as well as the information on the cards.

Fifteen copies of the cards and of the full documents are held in R.A.E. It is impossible for every programmer to have a complete set of either variety but both sets are available in Mathematical Services Department, the full documents in order of serial number, the cards in class order.

Specifications of subroutines and programs in the R.A.E. list are kept in the same place, as are the specifications of those N.P.L. routines which have been written up.

DEUCE library programs

The classification of programs starts with a pair of letters. The first letter, like the first letter of the subroutine classification, gives the class of program; the second letter sub-divides that class. A complete list of classes can be found in DEUCE News 1. Those most commonly occurring are:

- L: Linear algebra, including all bricks.
- T: Engineers' test programs.
- V: Demonstration programs for visitors.

The use of the second letter depends on the class; for instance, C following L indicates a check, whereas C following X (statistical programs) indicates a correlation program.

The two letters are followed by the serial number of the program in that sub-class. There are no further letters like F or C, nor is an extra digit attached following a stroke.

R.A.E. Library and N.P.L. Library

These both include subroutines and programs. The only classification is by serial number. A number is obtained for routines made at R.A.E. by filling in the title in the note book kept for this purpose and taking the next number vacant.

DEUCE News

This is an occasional publication which also comes from English Electric. It is chiefly used to issue corrections to published subroutines and programs, but it also has lists of routines awaiting publication, and occasionally some useful programming trick is explained.

III.2 DEUCE library subroutines

Subroutine specifications

When a subroutine is issued in the DEUCE library, its specification has the following items:-

- (i) The heading consisting of:
 - (a) Classification, alphanumeric, as described in III.1.
 - (b) Title.
 - (c) Serial number.
- (ii) Description: Order of the subroutine.
Amplification of title.
- (iii) Data: Number of binary places in data.
Ranges of values allowed.
- (iv) Results.
- (v) Error
- (vi) Uses: A list of all the lower order subroutines required.
- (vii) Failures: A list of failure indications and their causes.
- (viii) Instructions for Use. These are
 - (a) Stores used; Contents at entry; Contents at exit.
 - (b) Occupies: the delay line storage needed for the subroutine.
 - (c) Entry: the entry point or points.
 - (d) Time.
 - (e) Parameters.
 - (f) Waste Instructions.
 - (g) Useful Constants and Instructions.
- (ix) Notes, if any.
- (x) Method, if it warrants description.

The description will give the order of the subroutine, state the form of the numbers if not single-length (e.g. double-length, floating point) and describe the function performed.

The error should always be stated, even if it is less than $\frac{1}{2} P_1$. Regrettably, it is frequently omitted: e.g. only 14 out of 41 published subroutines of classes E, J and T specify it.

In "Uses" mention should be made if the instructions 1,14-25,0,0,0 in 126 and 0, 13-0,0,0,0 in 128 are used. Any subroutine is entitled to expect to find them there.

When a subroutine uses other subroutines the delay line numbers of these subroutines will have to be punched into the cards. It is conventional to leave 0 in the cards in all places where parameters have to be punched. The list of subroutines used will give their delay lines as B, C, D.... and the list of parameters will specify what has to be punched on the cards by using these letters.

The tabular layout of Stores used is extremely convenient; it shows which short stores are used, where the data (and the link) should be placed and where the results are to be found. A typical example might be:

Stores used	13	14	15	16	20	21 ₂	21 ₃
Contents at entry	Link	a	-	b	-	-	-
Contents at exit	c	-	-	b	-	0	2c

c is the letter conventionally used for the result of the calculation: the equation defining c in terms of a and b is given under "Results".

This table shows that

- (i) a and b, the two numbers needed by the subroutine, are placed in 14 and 16;
- (ii) the link is planted in 13;
- (iii) the result is to be found in 13;
- (iv) in addition, because of the working of the subroutine, 2c is left in 21₃ and there is zero in 21₂;
- (v) b is preserved in 16 but a is lost from 14;
- (vi) the numbers in 15 and both halves of 20 are lost;
- (vii) because they are not among the stores used, the numbers in 17, 18 and 19 are preserved.

"Occupies" specifies the storage space in the delay lines needed for the instructions of the subroutine: it does not normally include the space taken by any lower order subroutines. The subroutine's working space in the short stores is specified in the "Stores Used" section.

"Entry" gives the location of the first instruction of the subroutine or, in the case of a subroutine which computes alternative functions (e.g. sine and cosine), the alternative entry points. If there is only one entry point, it is usually in m.c.28 of the first delay line of the subroutine.

When a subroutine occupies less than one delay line the specification might be:

Occupies	m.c.0, 3, 5 - 29
Entry	m.c.28.

For a subroutine spread over two delay lines the specification might be

Occupies	2, 3 ₀₋₁₅
Entry	2 ₂₈ .

This does not mean that this subroutine is only available in DL 2 and 3: there will be copies of the subroutine in the library for every pair of consecutive delay lines up to DL 7 and 8. What is meant is that the subroutine occupies all of one delay line and m.c. 0 - 15 of the next, and it is entered in m.c. 28 of the first delay line.

A few short subroutines with not more than 16 instructions (chiefly for multiplication and division) have been coded compactly to fit into the top half or the bottom half of a delay line. When such a subroutine is published the detailed coding will have the "upper" version coded in the early minor cycles of the delay line and the "lower" version coded in the later minor cycles of the same delay line. It is confusing that the "upper" version is stored in the lower-numbered minor cycles. Cards of the "upper" and "lower" versions of these subroutines are issued for each delay line so that the user can pick, say DO5(U) and MO4(L) as his division and multiplication subroutines and store them in the same delay line.

The Time should always be given, but, like the error, it is frequently omitted. When the time the subroutine takes depends on the data upper and lower limits should be given. This is given so that the time taken over any part of a program can be found: it is frequently essential to estimate the time the program needs between cards when the reader or the punch is running.

Parameters are chiefly preset parameters, i.e. ones which the programmer fixes when first making the program, such as the delay line in which lower order subroutines are stored. They have to be punched into the cards of the subroutine being described. For instance, in the description of T 01 (subroutine No.29), it is stated that the subroutine uses

S 06 (No.15) in D.L. B,

M 12 (No.14) in D.L. C,

and coefficients $a_0 \dots a_5$ in D.L. D, m.c. t to $(t + 5)$.

The entry under "Parameters" is:

m.c. 9 Add CP_2

m.c. 10 Add BP_2

m.c. 8 Put $IP_5 + tP_{17}$.

When a subroutine has a program parameter, one which is set in a short store every time the subroutine is entered, this may also be described under "Parameters".

Many subroutines have a waste instruction which leads to the link, often because the link cannot be placed on one side of a discrimination in the main loop. In such cases the location of this waste instruction is given so that the programmer can change it to any instruction he prefers: e.g. 9 - 24 at the end of a read or punch subroutine. The waste instruction is usually 0 - 0 with zero wait number and non-zero timing number. It must not be N, 0 - 0, 0, 0, 0 because this has the fatal effect of taking the word set on the I.D. as the next instruction. An alternative sometimes used in read and punch subroutines is N, 8 - 8, 0, 0, 0 because this can easily be changed to 9 - 24.

Useful constants and instructions include the locations of any constants like P_{31} , P_{19} or 10^{-2} and any destination 0 instructions which the

programmer may use in his master routine. The entry for a destination 0 instruction should make clear which minor cycle the next instruction is obeyed in.

Use of subroutine cards

When a subroutine has been published copies of it can be found in the drawer of library subroutines punched on brown DEUCE instruction cards. A subroutine which spreads over two delay lines will have copies for DL 2 and 3, 3 and 4, 7 and 8; these will be preceded by tab cards labelled 2, 3, ... 7. Brown instruction cards must never be fed into DEUCE (still less should pink ones which are used for the master copies of routines), so they must be copied on to green or blue ones on the Hollerith reproducer and the brown cards replaced in the library. As soon as possible the parameters should be punched into the (green) copy of the subroutine, reference being made to the subroutine specification, as incorrect and missing parameters are very common sources of error.

III.3 List of recommended subroutines

There are now over 200 subroutines in the DEUCE library. Among these are many instances of several subroutines to perform almost identical functions. This section contains some recommendation on which subroutines to pick: it does not give their full specifications. This list of subroutines must be a list of personal preferences. There is no reason to suppose that a subroutine must not be used if it does not appear in the list.

It is not possible to name a single subroutine for each operation because some subroutines use others. It is certainly better always to work with 30 binary place numbers (this phrase is explained at the end of III.5), so that the recommended subroutines for multiplication, division and summation of series are A.12/1 (No.222) in which three subroutines are stored in one delay line; but these were issued so recently that no second order subroutines use them. If the programmer needs to take exponentials he will probably use E.03 (No.79) with its auxiliaries M.12 and S.06 (Nos.14 and 15); so in this case M.12 will be the multiplication subroutine to use.

Multiplication, division, summation of series

These are the three most common operations.

Recommendation: A.12 (No.221), multiplication and division only.

A.12/1 (No.222), all three operations

Alternatives: M.12 (No.14), multiplication

S.06 (No.15), sum series

D.05/1 (No.56), division.

All these subroutines work with 30 b.p. numbers.

Both subroutines in A.12 are "zero order": their links, planted in 13, are obeyed Q₃₀ by using the 13 - 0 instruction in 1₂₈, is so that links in 1₃₀ are undisturbed. A.12/1 is A.12 with a first-order sum series subroutine added and it uses this property of the multiplication subroutine.

M.12 and S.06 are used by a number of subroutines later in this list. They are designed to fit into one delay line. S.06 is second

order, using M.12. When they are used D.05/1 is the suitable division subroutine.

Square roots

Recommendation: F.04/1 (No.136)

Alternative: F.01 (No.20)

F.04/1 takes half the time of F.04 and needs 23 instructions as against 14. Normally it is better to save time than storage space.

The description of F.01 is an example of muddled thinking. The actual operation is to take the square root of a, both a and $\sqrt{a}$ being to n binary places. The description assumes that a is an integer and introduces a parameter n without explaining its object. This is not the only subroutine description which writes about integers when the numbers involved are fractions.

Trigonometrical functions

Recommendations: T.01/2 (No.195), $\sin \frac{\pi x}{2}$ and $\cos \frac{\pi x}{2}$

T.05/3 (No.242), $\frac{2}{\pi} \tan^{-1} \frac{x}{z}$

All these subroutines work with 30 b.p. numbers.

The unit used for storing angles is the right angle, not the radian. This takes care of overflow. For instance, the angles $\pi/4$ and $5\pi/6$ are represented by the numbers 1/2 and 12/3 stored with 30 b.p. If these numbers are added in DEUCE the result is not $21/6$ but $21/6 - 4 = -15/6$: the angle represented is not $13\pi/12$ but $13\pi/12 - 2\pi = -11\pi/12$. These trigonometrical functions of these two angles are identical.

Thus, if θ is an angle measured in radians, we normally store $2\theta/\pi$ to 30 b.p. and do not worry about overflow. The only exception would be where θ occurred explicitly as well as in trigonometrical functions, e.g. in

$$\ddot{\theta} + k\dot{\theta} + K \sin \theta = 0,$$

the equation of the simple pendulum with a circular spring attached. In this case $2-r2\theta/\pi$ would be stored with r so chosen that overflow does not occur.

T.05/3 must be used in preference to T.05 and T.05/1 - these two subroutines give wrong results when the angle should be $\pm\pi/4$ and should not be available for use. It is a second-order subroutine, using A.12/1 (No. 222).

T.01/2 is a first-order subroutine which is faster than any other routine for its purpose.

Hyperbolic functions

Recommendations: T.07 (No.113), $\frac{1}{2} \cosh x$.

T.08 (No.114), $\frac{1}{2} \sinh x$.

T.11 (No.128), $2^{-4} \sinh^{-1} x$ or $2^{-4} \cosh^{-1} x$.

No alternatives exist.

T.07 and T.08 are both second-order subroutines using an unrecommended multiplication subroutine, M.04/1 (No.64). This can multiply quantities with any fixed number of binary places: for the purpose of these subroutines the number must be fixed at 31.

T.11 is fourth order, using E.06, M.12, S.06 and F.02 (Nos.99, 14, 15 and 43 respectively).

Logarithms

Recommendations: E.11 (No.248) $\log_e X$

E.06 (No.99) $\log_2 X$

Alternative: E.07 (No.100) $\log_2 X$

E.11 is a second-order subroutine using A.12 (No.222). It calculates $2^{-4} \log_e X$ to 30 b.p., given X to 30 b.p.

E.06 is third order, using M.12 and S.06 (Nos.14 and 15); E.07 is second order, using M.12 and S.05 (No.25). The latter is rather faster, but S.05 is rarely used as a sum-series subroutine because it cannot be stored in the same delay line as M.12. Both these subroutines find $2^{-4} \log_2 2^q x$, given x; datum and result to 30 b.p.

Exponentials

Recommendation: E.12 (No.253), e^x

E.03 (No.79), 2^x with scaling factors

Alternative: E.04 (No.80), 2^x with scaling factors

E.12 is a second-order subroutine to find $2^{-r} e^x$ to 30 b.p., given $2^{-q} x$ to 30 b.p. It uses A.12 (No.222).

The only difference between E.03 and E.04 is that between E.06 and E.07 (see Logarithms): E.03 is third order, using M.12 and S.06 (Nos.14 and 15), whereas E.04 is second order and uses S.05 (No.25) only.

The identical descriptions of E.03 and E.04 are as difficult to understand as any yet issued. They say "... subroutine forming $2^{r(x-1)}$ where x is given to 30 b.p. and $r = 2^q$ ($q > 0$). Both r and q are inserted as parameters". They are incorrect in saying $r = 2^q$: this is not necessary. A better description is "... subroutine forming $2^{-r} 2^x$ to 30 b.p. given $2^{-q} x$ to 30 b.p., where q and r are non-negative parameters of the subroutine. r must be chosen at least as great as the integer part of x to prevent overflow".

It can now be seen that if e^x is wanted, x must first be multiplied by $\log_2 e$ or half this quantity.

Quadrature

Also called numerical integration.

The only existing subroutine - N.02 F, No.141 - cannot be recommended. It is also in the wrong class of subroutines.

There is a need for subroutines using Gaussian methods of integration. They would have to use an auxiliary routine, like differential equations subroutines, and plant the link inside themselves.

Input These recommendations cover only routines reading the α -field. The file of subroutine cards should be consulted for subroutines to read off both fields.

<u>Recommendations:</u>	R.01/3 (No.228), read one fraction on a card	} α field only
	R.02/1 (No.171), read two fractions on a card	
	R.03/1 (No.108), read three fractions on a card	
	R.04/2 (No.189), read four fractions on a card	
	R.07/2 (No.211), read eight integers on a card	
	R.13 (No.149), read integers on a card in any layout	
<u>Alternatives:</u>	R.01/1 (No.84), read one integer from a card	} α field only
	R.02 (No.93), read two integers from a card	
	R.03 (No.48), read three integers from a card	
	R.04 (No.105), read four integers from a card	
	R.08 (No.18), read one integer from the I.D.	

There are 34 read subroutines to choose from at the time of writing. The variety occurs because three factors have to be considered:

- (i) Layout on the card.
- (ii) Integers or fractions.
- (iii) Checks on decimal punching.

The list of recommendations is long because the user may have different needs in different programs. It seems likely, however, that he will usually want to read fractions, so the somewhat simpler subroutines to read integers have been listed as alternatives. He will certainly want subroutines which check that the cards have been decimally punched because incorrectly punched data cards give so much trouble if not noticed immediately: the only subroutines in this list which do not check the punching are R.13 (a later version has been made which incorporates a check) and R.07/2.

The beginner will certainly find it easiest to read one number per card with R.01/3.

R.02/1, R.03/1 and R.04/2 have a table of scaling factors of the form $10^{-D} 2^{p+30}$. The user first chooses where the decimal points will come on the cards: D is the number of decimal digits below the point. He takes the values of p and $10^{-D} 2^{p+30}$ from this table and punches the latter into the cards of the subroutine. The results will be given to p binary places.

R.13 is an exceptionally flexible routine. A single parameter specifies how many numbers are punched on a card, the spaces between numbers and how many digits are punched in each number - they do not all need to have the same number of digits. This parameter is a binary word in which each bit corresponds to a column of the card. It is zero for columns which are not punched and for the columns containing the last digit of each number; it is one for the columns containing the signs and the other digits of the numbers.

The two disadvantages of R.13 are that it reads integers (so that a multiplication by a scaling factor is needed if fractions are read) and that

most of its work is done after the last row of the card has been read. This may mean that cards cannot be read consecutively because too much computing would take place between cards. In such cases the reader will have to be switched off after every card. The specification of R.13 gives the time conversion takes: the user has to work out whether there is enough time in his particular case;

Output Here again these recommendations cover single field punching ~~reading~~ only.

<u>Recommendations:</u>	P.01 (No.12), one fraction	} a-field only
	P.02/1 (No.91), two fractions	
	P.03 (No.22), three fractions	
	P.04 (No.23), four fractions	
	P.14 (No.130), fractions in variable layout	
<u>Alternatives:</u>	P.01/1 (No.133), one integer	}
	P.09 (No.38), eight integers	
	P.16 (No.147), integers in variable layout	
	Z.01 (No.46), display one integer on monitor.	

There are nearly as many punch subroutines as read subroutines for the same reasons, but, since punching fractions is simpler than integers, there is not the same range of subroutines for punching integers.

The first five subroutines in the list are described as if they can be used with a range in the number of binary places. This is misleading because the numbers are punched with only one decimal digit before the point: they have to be less than 10, so there are dangers in using them for numbers with 27 b.p. or less.

P.14 is the output equivalent of R.13. It is, however, a little less flexible in that it must punch the same number of decimal places every time: all subroutines for punching fractions to D decimal places add a round-off constant of $\frac{1}{2} 10^{-D}$ before starting the conversion. This disadvantage can be ignored if the programmer is prepared to add the round-off constants in the master routine.

With P.14 the layout of the numbers on the card is as flexible as with R.13. The layout parameter is the exact converse of that used by R.13; it has a one for columns which are not punched and for the final digits of numbers, and a zero for columns where the signs and the other digits are punched.

P.16, which is the integer equivalent of P.14, has an identical layout parameter. The disadvantage of both these subroutines is that it may not be possible to run the punch continuously as they both do all the computing before starting to punch. P.14 can just punch five 6 digit fractions per card and P.16 may manage four 7-digit integers. The total time increases with the number of values punched; e.g. the subroutines cannot manage 6 5-digit fractions or seven 4-digit integers with the punch running continuously. As with R.13, the user must read the specification and work out whether in his particular case there is sufficient time.

Floating-point arithmetic

Recommendation: A.13F, A.14F, A.15F, A.16F (Nos.236-239)

Alternative: A.11/1 (No.192).

Floating point numbers (a, b) represent $a \cdot 2^b$ - see IV.3. The first four subroutines store a (a fraction) and b (an integer) in a pair of words, the last one stores a and b in the same word. It is slower and considerably less accurate, but it takes considerably less space for storing numbers. Subroutine A.13F contains Add, Subtract, Multiply and prepare. The A.14F, A.15F and A.16F subroutines add on Divide, square root and sum series respectively.

Other floating-point functions

Recommendations: R.19F/1 (No.269), read floating decimal vector and store in floating binary.

P.10F/2 (No.220), punch floating binary number in floating decimal.

E.05F/1 (No.235), exp x.

E.10F/1 (No.209), $\log_e x$.

Alternative: P.13F/3 (No.202), punch floating binary number in floating field.

Double-length arithmetic

Recommendations: M.10D (No.36), multiplication.

D.12D (No.66), division.

F.03D (No.67), square root of integer.

S.03D (No.44), sum series.

R.11 (No.69), read integer.

P.06D (No.21), punch fraction.

P.15D (No.132), punch integer.

Alternative: R.13 (No.149), can be used for reading double-length integers as well as single-length ones.

All these subroutines have double-length arguments and results.

Continuous store

Recommendations: B.01 (No.19), fetch or store in delay lines.

B.02/1 (No.161), fetch or store tracks.

B.06 (No.200), fetch or store consecutive words on drum.

Alternatives: B.03 (No.181), fetch consecutive words from drum.

B.04 (No.182), store consecutive words on drum.

B.05 (No.199), modify consecutive words on drum

Also B.09 (No.210), B.10 (No.251), B.11 (No.266) and B.12 (No.282)

To the programmer, one of DEUCE's chief disadvantages is the lack of a large continuous store. Words have to be split into blocks of 32, whereas Pegasus, for example, has been carefully designed so that it is realistic to address the words on the drum consecutively. Subroutines of this class are designed to make it possible to treat DEUCE's mercury and magnetic drum stores as continuous.

B.01 (generally known as "Fetch and Carry") makes the mercury store continuous. It is given an instruction in TS 13 which, if obeyed unmodified and Q30, would fetch or store word number 0 of the series. The number of the word to be fetched or stored is placed in TS 14 and the subroutine is entered at the "Fetch" or "Store" entry points. The subroutine modifies the instruction in 13 suitably and then obeys it. If the instruction placed in 13 is 2, 6 - 16, 0, 0, 12 the number in 14 is 42P₁₇ and "Fetch" is entered, the instruction obeyed is 2, 7 - 16, 0, 10, 12. The consecutive words in the store are 6₀, 6₁, ... 6₃₁, 7₀, ... 7₃₁, 8₀, ...

With B.01 the unmodified instruction should always refer to m.c.0. If the instruction were 2, 6 - 16, 0, 2, 12, the consecutive words would be 6₃, 6₄, ... 6₃₁, 6₀, 6₁, 6₂, 7₃, ... 7₃₁, 7₀, 7₁, 7₂, 8₃

B.02/1 will fetch one track of the drum to DL 11 or store DL 11 on a track. It is given a single parameter (in TS 15) which specifies both the number of the track and whether "fetch" or "store" is being used. This subroutine includes B.01.

With B.01 and B.02/1 the programmer can refer to words in any sequence he chooses. More frequently he will want to pick out words consecutively. B.06 is a first-order subroutine with five entry points to enable successive words on the drum to be fetched or to be stored; only two entry points are used regularly. The programmer first enters the subroutine at the points "Prepare fetch" and "Prepare store" with parameters which specify the locations of first words of the sequences to be fetched and stored. In the main loop the entries "Fetch" and "Store" can be used as often as the programmer requires to handle the successive words. At the end the entry point "Complete Store" is used to write the last track on the drum.

Simplified versions of B.06 have been published for the cases where storage is unnecessary (B.03), fetching is unnecessary (B.04), and where the programmer is fetching and storing the same sequence of numbers (B.05).

WARNING More than most of this note, this list is likely to become out of date. All the authors can hope to do is to recommend subroutines from those available in March 1959. If some of their suggestions for new subroutines are taken up, or if new versions of existing subroutines are made to use the recommended subroutines instead of unrecommended ones, they will be only too pleased to see the list become outdated.

III.4 Specifications of programs

The programmer should write a set of operating instructions for a new program as soon as it has been fully tested. In practice, however, this is usually left until ten minutes before somebody who has never seen the program has to operate it (alone, on an evening shift), and the chaos that results may take days to unravel.

The operating instructions should first give the minimum information necessary if all the data cards have been prepared and the program works perfectly:

- (i) The name or number of the program.
- (ii) The order of the cards. (The whole pack of program cards should be read before any parameter or data cards.)
- (iii) What to set on the I.D. - if anything. This is better avoided.
- (iv) The setting of the numbering switches on the punch.
- (v) How to recognise the end of the program - specify what appears on the I.S. lights.
- (vi) If known, how many cards will be produced.
- (vii) Any checks the operator should perform immediately - e.g. running the program twice and comparing results.

Be brief, and do not expect the reader of this section to have any intelligence.

These should be followed by details of what happens if the program does not work perfectly.

- (viii) The location on the monitors or the O.S. of any counters which change steadily, so that the operator can see whether the program is working correctly. (This is particularly necessary in programs which compute for long periods without reading or punching.)
- (ix) How to make an "emergency stop" - if one has been written into the program.
- (x) A list of all the failure indications.
- (xi) How to restart the program, particularly after an emergency stop. (Say here whether reading the last one or two program cards with initial input is sufficient to bring the entire program down from the magnetic drum.)

This section will have to assume some intelligence from the operator.

Finally, for experienced operators only,

- (xii) Exact details of the punching of data cards and any parameter cards.

When these operating instructions have been written it should be possible for the program to be used without any further reference to its maker. It is well worth showing a draft of the operating instructions to someone else, preferably to a colleague who has not been concerned in writing the program, to get advice on the English as well as to find any ambiguities or omissions. At R.A.E. the operating instructions must be shown to a Vetting Committee. This committee also makes recommendations about the organisation of programs.

When a program has to be written up for publication in the DEUCE library the operating instructions can form a basis for the specification. Some items, e.g. (iv), (vi) and (vii) may have to be omitted because they depend on the particular job. Others, such as an outline of the method, will be inserted and the various sections placed in a different order. The specification of a program can be more flexible than that of a subroutine. The following will probably be the main headings:

Summary.

Data. How punched. Limits on values.

Results. How punched. Number of cards expected.

Operating Instructions. Some of (ii), (iii), (v), (vii), (viii), (ix), (x) and (xi) of the operating instructions.

Description of the methods used.

PART IV SPECIAL TOPICS

The nine sections that follow are intended for the more advanced programmer. Knowledge of the topics discussed is helpful, but ignorance is not disastrous.

Accounts have already been published of the Detailed Coding Program, the Program Changing Routine, the NPL General Interpretive Program, the Alphacode and the RAE Translation Routine for a single address code. Nevertheless, it has been thought desirable to include brief summaries of these programs in this Note, since the publications are not always easily accessible and anyway cater only for the experienced programmer.

As far as the authors are aware, no publication concerned with DEUCE has yet appeared dealing with floating point arithmetic, as used on DEUCE. The specifications of subroutines are not a satisfactory introduction to the subject.

No readily available account of interpretive programs has appeared. What literature exists is scattered in many journals and books. The short article in IV.4 describes some of the reasons for having such programs and their use.

The three DEUCE conventions on matrix algebra are briefly explained and compared in IV.7. The most important of these is 'Scheme B', which is used in connection with the General Interpretive Program.

The simplified programming system known as "Alphacode" is described in IV.8. Ease of programming is bought at the expense of time on the computer.

The ninth and last section concerns Logical Design. This comprises the considerations that go into designing a computer to perform certain functions. An understanding of what goes on inside the computer is often useful, besides being interesting in its own right.

IV.1 The detailed coding program (R.A.E.415)

R.A.E.415 is a program constructed to assist in the detailed coding necessary for DEUCE programs after allocation of storage has been completed. The program is simple to use, is rapid in action and caters for all the whims of the programmer. Facilities are available for including library subroutines, decimal and binary constants in a program. There are also some checks on the consistency of the information presented to DEUCE. A complete description of R.A.E.415 has already been published (Ref.7) and the operating instructions are also available separately (Ref.8).

Instructions are punched, in decimal, directly from the flow diagram, with one instruction per card. The location of the instruction is punched on each card. It is not, in general, necessary to specify the position of the next instruction as this is the place where the next instruction is stored and is therefore punched with the instruction itself. There is a facility for breaking the sequence, as may happen at the end of a loop when there is a jump to an instruction that has already been stored. There is a special facility to allow the coding of links and 'quasi-instructions', as these are not obeyed from the minor cycle in which they are stored. The maximum length of a program that may be coded is 168 IL's. Decimal constants may be either fractions or integers with a scaling factor.

The output consists of the coded program. There is also a failure card which records the number of elementary blunders of particular types. These errors make the routine place distinctive markers in the program and are easily located.

IV.2 The program changing routine (R.A.E.180)

The object of this particular program is to facilitate the construction of large programs. As a complete description of the routine is available (Ref.9) we content ourselves with only a brief summary.

Programs too large for the high speed store can often be split up into logically self-contained blocks that are sufficiently small for the high speed store. These blocks of program may often be of use in their own right in other programs, rather like subroutines. One is therefore led to the idea of constructing a complete program from several sub-programs linked by a master routine. Every time a sub-program is to be used it is necessary, before entry, to plant a link that will return control to the correct place in the master routine after the sub-program. Slightly more is necessary, however, if the programs are large. In general, it is impossible to keep the master routine always in the mercury store. We therefore adopt a system whereby either the master routine or a sub-program is 'downstairs'. It is therefore necessary for the master routine to know where each of the sub-programs is to be entered - we must give a 'cue' - and also where it has been stored on the drum.

It is desirable to be able to specify the position on the drum and size of the master routine and sub-programs in as compact a form as possible, in a form rather akin to the parameter required for Magnetic Fetch and Carry. A special routine is then required to fetch and enter the master routine or the appropriate sub-program. This is the role of the Program Changing Routine.

This routine has four entry points. The master routine normally enters it at 1_2 with a cue and a link (described below) in 21_2 and 21_3 . Both are stored on the drum. The cue is then interpreted and the appropriate sub-program brought down and entered. Generally the sub-program ends by bringing the Program Changing Routine into the high speed store and entering it at 1_1 . The link is now interpreted and obeyed, so that the master routine is brought down and entered at the correct point.

The other entries are much less frequently used. They are described later in this section.

The cue and the link are alike in form. The only difference is that one, the cue, refers to the sub-program whereas the other, the link, refers to the master routine. Both cue and link must be supplied by the master routine.

For convenience, sub-programs are made to end in a standard way, so that they can easily be fitted together. The last three minor cycles of DL 12, i.e. $12_{29,30,31}$ are used to store three instructions which will fetch and enter the Program Changing Routine. These instructions are transferred to DL 1 at the end of the sub-program, by the instruction

$$1, 12 - 1 \text{ (32 m.c.)}$$

leading to 1_{30} . There is a special one-card program, R.A.E.181, for planting these three instructions in DL 12.

Sometimes a sub-program itself is too large for the high speed store. It must then be split into two or more parts that use the high speed store in turn. One entry to R.A.E.180 allows for this possibility by adding a quantity given by the sub-program to the cue so that the next part is called down. Thus if the whole sub-program consists of two parts and the necessary cues, if the parts were used separately, would be C_1 , C_2 , we first give R.A.E.180 the cue C_1 . This will allow the first section to be obeyed. The cue itself is stored. For the second section we arrange that the first section leaves $(C_2 - C_1)$ in the correct place and enters R.A.E.180 at the appropriate entry point. The previously stored cue, C_1 , is added to $(C_2 - C_1)$ and the resulting cue, C_2 , is stored and interpreted, calling down the second part. After this section we re-enter the Program Changing Routine at 1_4 as usual, interpret the link and return to the master routine.

There are several such sub-programs already in existence which can be used in a variety of programs. If they are to be of general application they should not contain a cue which depends on the position of the sub-program on the drum. So the first section of the sub-program cannot give the cue C_2 directly to Program Changing Routine to call down the second section because C_2 contains the number of the track on which the first delay line of the section is stored. On the other hand the second section will always be stored adjacent to the first section, so $(C_2 - C_1)$, which contains the difference between two track numbers, does not depend on the position of the sub-program on the drum. It can therefore be stored in the first section of the sub-program and planted in the correct place for the Program Changing Routine.

The Program Changing Routine has been assembled with the various administrative programs into one pack, called R.A.E.321. This comprises

- (i) Clear Drum (R.A.E.353).
- (ii) Set even minor cycles and clock track (R.A.E.354)
- (iii) Write on drum (R.A.E.136).
- (iv) R.A.E.180
- (v) Plant instructions in $12_{29,30,31}$ (R.A.E.181).
- (vi) Enter master routine (R.A.E.138).

The 'private' program is inserted between items (iii) and (iv).

R.A.E.138 is a one-card program which allows the first entry to the master routine. The link has to be punched on the 9's row of the card*.

If working with double field programs, (i), (ii) and (iii) above can be replaced by R.A.E.481 and (v) and (vi) by R.A.E.482.

The cue and link are of exactly the same form. The cue is always planted in 21_2 , the link in 21_3 .

The specification of R.A.E.180 is given below. For a fuller description of the routine and its use the reader is referred to Ref.9**.

Specification of R.A.E.180, the Program Changing Routine

When the parameter $NP_2 + mP_5 + tP_{17} + (r - 1) P_{22} + tP_{26}$ is interpreted the routine will first read the r tracks from m to $(m + r - 1)$ from the drum and store them backwards in delay lines r to 1. Control will then be sent to N_t .

The parameter interpreted depends on the minor cycle of entry and the content of TS_{13} on entry as follows:-

Entry 1_2 with cue and link in 21_2 and 21_3 ; cue and link stored and cue interpreted.

Entry 1_4 with TS_{13} clear; previously planted link interpreted.

Entry 1_4 with P_{32} in TS_{13} ; previously planted cue modified by the addition of 21_2 , then stored and interpreted. The link is unaffected.

Entry 1_1 - previously stored link interpreted.

Entry 1_{21} - parameter just planted in TS_{14} interpreted.

(These last two entries are kept because the previous version of the routine had them. In new programs the entries at 1_4 should be used in preference to these.)

If any of the parameters interpreted contains a P_{32} the machine will stop on $N_{11} - 1$ immediately before entering the program at N_t . This facility is useful when testing the program.

The routine occupies DL_1 and should be transferred there before entry. It should be stored on track 254 of the drum, for it rewrites itself on to that track each time a new cue or link is given, thus preserving these parameters for later use. The cards are punched so that the routine will be read to this track of the drum by R.A.E.136. Normal input to the delay lines will be resumed after the input of this routine.

*In an older version the link is punched on the 2's row. For the current version, if there is punching in column 1 of the 8's row the parameter can be set up on the I.D. keys. The machine will stop on 1, 0 - 14 X to read this parameter.

**There is an error on p.10 of this reference. The first two instructions should read

$$1_4: 1_{14} - 22_2$$

$$1_7: 1_{13} - 21_3$$

The routine uses Ts 13, TS 14, TS 15, DS 21 and DL 11. No other stores are disturbed except those delay lines into which new program is transferred.

Insertion of parameters for sub-programs

It is often necessary to leave parameters in a particular place for use by one of the sub-programs. Many programs are arranged so that they expect to find the necessary parameters in QS 18. A useful systematic method of planting parameters and the cue and link is the following one, shown schematically.

<u>DL A</u>		<u>DL (A + 1)</u>	
A, (A + 1) - 18, 1, 30, 1		parameters	18 ₀
Link } for sub-program 1		for	18 ₁
Cue } for sub-program 1			18 ₂
1, A - 21, 2, 28, T		sub-program 1	18 ₃
.....			
A, (A + 1) - 18, 1, 30, 1		parameters	18 ₀
Link } for sub-program 2		for	18 ₁
Cue } for sub-program 2			18 ₂
1, A - 21, 2, 28, T-4		sub-program 2	18 ₃
.....			
etc.			

In this particular arrangement the master routine has R.A.E. 180 as its DL 1. The timing number T (above) is chosen to lead to 1₂. The program is faster if R.A.E.180 is incorporated in the master routine if this is possible, but this is not essential as the difference in time is very small.

IV.3 Floating point arithmetic

The difficulties of scaling all problems so that numbers occurring stay within the capacity of the machine have already been mentioned. A systematic method of scaling is very desirable; as far as possible it should be easy to use. Floating point arithmetic meets these demands.

Use of floating point arithmetic is rather akin to using logarithms. The emphasis is on numbers of significant digits, not on the total number of digits. This brings its own difficulties, but these are matters of numerical analysis not programming.

Instead of storing the number x we store two numbers associated with it, a mantissa and an exponent or scaling factor. We write x as

$$x = a.2^b$$

where b is an integer and $\frac{1}{2} \leq |a| < 1$. (There is some ambiguity about x = 0 and the end-points as this stands - this is resolved below.) We call

and

a	the mantissa
b	the exponent.

For each number x we now store the pair (a, b) which will generally take two words. On DEUCE this means that $|b| < 2^{31}$ and so we can handle numbers in the range

$$10^{-6} \cdot 10^8 < |x| < 10^6 \cdot 10^8$$

This range is so vast that most ordinary needs are easily met.

If a lies in the range $\frac{1}{2} \leq |a| < 1$ we can store an exact power of 2 in one of two ways. Suppose $x = 2^d$; then we could write

$$x = 2^d \cdot 1 \quad \text{or} \quad 2^{d+1} \cdot \frac{1}{2}$$

and both the possible a 's lie in the range. We therefore restrict the range by removing two end points and say that a must satisfy

$$\text{either} \quad -1 \leq a < -\frac{1}{2} \quad \text{or} \quad \frac{1}{2} \leq a < 1.$$

(In either case the numerically smaller end is included.) The number 2^d would then be stored, unequivocally, as

$$x = 2^{d+1} \cdot \frac{1}{2},$$

where the mantissa is $\frac{1}{2}$ and the exponent $d + 1$.

One exception occurs to the rule with the mantissa. It is impossible to represent zero in the form $a \cdot 2^b$ with $|a| \geq \frac{1}{2}$. Therefore, by convention, we put

$$0 = 2^0 \cdot 0,$$

i.e. store 0 as the pair $(0, 0)$.

So far the remarks about floating point arithmetic have been applicable to all machines. For the rest of this section we shall be concerned with the floating point arithmetic available on DEUCE.

The mantissa is always stored to 30 b.p. With the convention about the range of a this means that the three most significant digits of a are either

$$100 \quad (a > 0) \quad \text{or} \quad 011 \quad (a < 0).$$

A number that has been expressed in the above form is said to be in 'standard floating form' or 'standard floating binary' (s.f.b.). For ordinary floating arithmetic we store (a, b) in consecutive minor cycles, with b in the even minor cycle and a in the odd one. (There are some other schemes, where the exponent and mantissa are stored in a different form. These are mentioned below.)

All the floating point subroutines include a section called 'Prepare' which puts a number into standard form. To enter Prepare the number x must be given as

$$x = a \cdot 2^b$$

where a is to 62 b.p. in DS 21 and the exponent is in TS 13.

Example 1 To put the integer n into s.f.b. This is achieved by

(n) - 21 (2 m.c., e.o.)

(62 P1) - 13

(Enter Prepare)

Example 2 If the integer n is known to be less than 2^{31} in modulus a faster method is

30 - 21₂

(n) - 21₃

(30 P1) - 13

(Enter Prepare)

Prepare will now shift up the number in DS 21 until the top three (binary) digits are either 100 or 011 and count the number of shifts required. This means that a is brought to standard form and the exponent adjusted accordingly.

A convention with floating point subroutines is that the link should be placed in 19₂. Unfortunately this is not always observed.

In the examples given below we shall use the subroutine AO_{3F}/3 (DEUCE subroutine 214), which contains six sections, called Add, Subtract, Multiply, Divide, Square Root and Prepare. It is normally stored in DL's 2, 3 and 4 and its entries are

Add	421	Subtract	416	Multiply	230
Divide	23	Square Root	40	Prepare	225

The link is always in 19₂, the operands in DS 20 and DS 21, the result in DS 21 and also in TS 13 (exponent) and TS 14 (mantissa). The only exit from the subroutine is via 'Prepare'.

/Example 3

Example 3 Given x_1 in $17_{0,1}$ and x_2 in $17_{2,3}$ place $y_1 = \frac{1}{2}(x_1 + x_2)$ in $18_{0,1}$ and $y_2 = \sqrt{x_1 x_2}$ in $18_{2,3}$.

$17_{0,1} - 20$ (2 m.c.) x_1

$17_{2,3} - 21$ (2 m.c.) x_2

Link - 19_2

Add

Puts $x_1 + x_2$ into DS 21

$1_{30} 27 - 26$

Exponent is in TS 13 at end

$13 - 21_2$

Reduce exponent by 1, i.e. $\frac{1}{2}(x_1 + x_2)$

$21 - 18_{0,1}$ (2 m.c.)

Store $\frac{1}{2}(x_1 + x_2) = y_1$

$17_{0,1} - 20$ (2 m.c.) x_1

$17_{2,3} - 21$ (2 m.c.) x_2

Link - 19_2

Multiply

$x_1 x_2$ in DS 21

1_{30} Link - 19_2

Square Root

$\sqrt{x_1 x_2} = y_2$

$1_{30} 21 - 18_{2,3}$ (2 m.c.) Store y_2 .

All our arithmetical operations now have to be done by subroutines. Often there is considerable loss of time, especially in 'Prepare', as the results of all operations have to be put into standard form. When doing many additions this loss of time can be quite serious - multiplication is quite fast because one shift at most is needed to bring the result to standard form. Addition and subtraction, however, require that the exponents of the two numbers are the same, so that some shifting down is required and then quite a lot of shifting up may be needed at the end. In general fixed point arithmetic is reckoned as being about 10 times faster than floating point. On the other hand, there is a considerable gain due to the avoidance of scaling.

There are occasions when more accuracy is required than that given by the 32 binary digits of the mantissa, (about 10 decimal digits). To this end some double-length and even triple-length floating point subroutine exist. The mantissa now needs two (three) words and the whole number takes up three (four) consecutive words. The use of such subroutines is very similar to the ordinary single-length floating arithmetic.

On the other hand the range of exponent is often much more than is required and frequently fewer than 32 binary digits will suffice for accuracy. There are some so-called 'packed-floating' point subroutines which store the exponent and mantissa in one word. The eight least significant digits are used for the exponent (to give x a range of roughly 10^{-40} to 10^{40}) and the remaining 24 digits are the mantissa. Full instructions for their use are given with the subroutine specification sheets.

Input and output of floating numbers is achieved by subroutines which convert from floating decimal to floating binary and vice versa.

IV.4 Interpretive and translating programs

Programmers in general are dissatisfied with the machine at their disposal. Whatever facilities are supplied by the engineer, the mathematician invariably demands more. Changing an existing machine is not an easy matter and much time has therefore been spent on constructing programs to make machines behave as if they had different facilities available. This procedure is useful only if the writing of programs in the artificial instruction code is considerably simpler than writing the corresponding program in the natural code. Such 'artificial programs' will require much less testing than normal programs, because the new code will be much more powerful than the actual machine code.

Two approaches are now possible. The first method is to translate every artificial instruction into the corresponding sequence of natural instructions at input. This means that before the program of artificial instructions is obeyed a complete program is written by the machine. This machine-language program is then obeyed in the normal manner. Programs that operate in this manner are called 'translating programs'. They may be very ambitious and prepare a program from a code that is closely allied to ordinary mathematical notation. Such programs are often referred to as 'Compilers' or 'Generating Programs'. On the other hand, the translating program may be quite small and do a 'word for word' translation. An example of such a program is R.A.E.424, described in IV.5. This routine allows DEUCE to be programmed as if it is a one-address machine.

The second possibility is to provide the machine with a dictionary and translate every artificial instruction as it is encountered. This will be much slower because no record is kept of what has been converted into actual instructions. This means that the instructions of a loop have to be translated every time the loop is traversed. Such programs are called 'Interpretive Programs'. An example of such a program developed for DEUCE is the NPL General Interpretive Program, described in IV.6.

It is not the intention to give a full account of such programs here but rather to introduce the reader to a fascinating topic.

On machines which have fixed point arithmetic the scaling of numbers to keep them within the capacity of the machine is often a source of trouble. Floating point arithmetic, introduced in IV.3, presents a way out of the difficulty. However, it is tedious to use because every operation has to be programmed or done by subroutines, which will all require links that must be planted correctly. A very common form of interpretive program is one to convert a fixed point machine into a floating point machine*. An instruction code has to be made up and the floating point subroutines are

* Paradoxically, on the Ferranti Mercury Computer, which normally works in floating point arithmetic, an interpretive program is necessary to do fixed point arithmetic.

incorporated in the interpretive program. The instructions in the new code are then interpreted one by one and with each instruction the appropriate subroutines are called by the interpretive routine. Generally the artificial instructions are obeyed sequentially (unlike ordinary DEUCE instructions), because the time taken to obey each one is so long that nothing will be gained by trying to place them in a better order. At present there is no such floating point interpretive scheme for DEUCE.

A second very common type of interpretive program concerns matrix algebra. Here, programs to perform operations on matrices, such as addition and multiplication, take the place of the corresponding operations on numbers. The code is arranged so that a matrix is defined by the address of its first element and an instruction might take the form

$$a + b \rightarrow c,$$

where this is to mean that two matrices, starting at a and b, are to be added and the result is to be placed at c. Often the available functions are numbered, but codes do exist where mathematical symbols, like +, may be written in the instructions. Such matrix interpretive programs allow one to construct extremely complicated programs with very little effort. The time taken in interpreting artificial instructions is now very small compared to the time taken by the actual matrix operations: it is rarely worth the time and effort to make a rapid master routine to link the matrix programs if the whole program is to be used only a few times.

The NPL General Interpretive Program, described in IV.6, can be used for any operations, although it is generally only used for matrix algebra.

IV.5 The R.A.E. translation routine for a single-address code (R.A.E.424)

One of the reasons why DEUCE is an awkward machine for which to make small programs is its two-address (Source and Destination) code: another is the optimum coding which follows the writing of the flow diagram. Certainly the simplest type of code is one with a function and one address in which instructions are stored in the order in which they are obeyed. A routine R.A.E.424 exists so that DEUCE can be programmed as if it were a one-address machine. It translates each "instruction" into DEUCE instructions almost as soon as it is read, and only the DEUCE instructions are stored while the program is obeyed.

This section will not describe the translation routine in every detail, since this has already been done by its author (Ref.5): only the salient points will be brought out.

The "instructions" are punched in binary - 32 instructions take 3 cards, just like ordinary instructions - with the function F in the P10 to P14 digits and the single address (A,B) in the P1 to P4 and P5 to P9 digits respectively. The address is written as a pair of numbers because it usually refers to minor cycle B of delay line A. Table 1 gives the effect of the "instruction" in delay line d, minor cycle n: unless otherwise stated the next instruction is in d_{n+1} .

/Table 1

TABLE 1

<u>F</u>	<u>Effect</u>
0	Next instruction in A_B , not in d_{n+1} .
1	$C'(13) = C(A_B)$. $C(A_B)$ unchanged.
2	$C'(14) = C(A_B)$. $C(A_B)$ unchanged.
3	$C'(15) = C(A_B)$. $C(A_B)$ unchanged.
4	$C'(16) = C(A_B)$. $C(A_B)$ unchanged.
5	$C'(13) = C(13) + C(A_B)$. $C(A_B)$ unchanged.
6	$C'(13) = C(13) - C(A_B)$. $C(A_B)$ unchanged.
7	$C'(A_B) = C(13)$. $C(13)$ unchanged.
8	$C'(A_B) = C(14)$. $C(14)$ unchanged.
9	$C'(14) = 2^B C(14)$. (A has no effect).
10	$C'(14) = 2^{-B} C(14)$. (A has no effect).
11	Next instruction $\left\{ \begin{array}{l} \text{in } d_{n+1} \text{ if } C(13) > 0 \\ \text{in } d_{n+2} \text{ if } C(13) < 0 \end{array} \right\}$ (A and B have no effect)
12	Next instruction $\left\{ \begin{array}{l} \text{in } d_{n+1} \text{ if } C(13) = 0 \\ \text{in } d_{n+2} \text{ if } C(13) \neq 0 \end{array} \right\}$ (A and B have no effect)
13	The machine stops. No other effect.
14	Not used.
15	Dummy instruction.
16 to 31	Used for subroutines. Explained below.

The program, R.A.E.424, takes 32 of these words and stores them in a delay line. It then takes them one by one, translates each into a DEUCE instruction and stores the DEUCE instructions. Having translated 32 words it writes them on the magnetic drum and reads the next 32 words. Table 2 gives the dictionary which the translation program uses for the word in minor cycle n of delay line d .

/Table 2

TABLE 2

Function	DEUCE Instruction					
	N.	S.	D.	Ch.	W.	T.
0	A,	13 - 13,		0,	0,	B-n-2
1	d,	A - 13,		0,	B-n-2	31
2	d,	A - 14,		0,	B-n-2,	31
3	d,	A - 15,		0,	B-n-2,	31
4	d,	A - 16,		0,	B-n-2,	31
5	d,	A - 25,		0,	B-n-2,	31
6	d,	A - 26,		0,	B-n-2,	31
7	d,	13 - A,		0,	B-n-2,	31
8	d,	14 - A,		0,	B-n-2,	31
9	d,	24 - 14,		1,	32-B,	31
10	d,	23 - 14,		1,	32-B,	31
11	d,	13 - 27,		0,	32-B,	31
12	d,	13 - 28,		0,	32-B,	31
13	d,	13 - 13,		0,	32-B,	31 X
15	d,	13 - 13,		0,	32-B,	31

It is suggested that a number of function 13 instructions should be inserted in the program for testing purposes. When they are no longer wanted only one hole need be punched to make their function 15 and they become ineffective.

The main functions of the translation routine have now been given, but it performs certain other operations as well.

It is possible to include ordinary DEUCE instructions among the one-address codes. They are punched in binary in the usual way with the P31 added and the routine stores then unmodified. This facility is used chiefly for inserting magnetic transfers.

Functions 16 to 31 can be used for subroutines. The user picks subroutines whose data and results are to be in the TS's (in particular the links must be placed in 13) and numbers them 16, 17, ... In the translation routine he inserts one parameter for each subroutine. Then each time a subroutine is needed he punches the corresponding function (without any address) and leaves the next word blank. The translation routine stores the instructions: the first plants the link and enters the subroutine the second is the link - a dummy instruction which leads back into the program. In this way multiplication, division, square root and other functions can be built in.

In the top row of each triad of cards the user punches the track on which the triad is stored. On the second row he punches the number of the delay line in which it will be obeyed. The third row is punched if every instruction is a DEUCE instruction: this is used for reading subroutines.

If the fourth row is punched the translation routine will not read any more program cards: instead it will bring track 1/0 into DL 2 and enter it at m.c. 0. The translated DEUCE instructions are now obeyed. Among them there will have to be magnetic transfers to bring the rest of the program down from the drum into the right delay lines.

The advantages of this translation routine are that short programs are easier to code and less likely to be in error when coded. The obvious disadvantage is that it wastes time in obeying the instructions because it ignores the possibility of optimum coding. This is not such a serious disadvantage, however, in programs where the master routine strings together various subroutines and does little else, and in particular in programs where most of the time is spent in input and output.

IV.6 The N.P.L. General Interpretive Program (G.I.P.)

There are three conventions about matrix algebra on DEUCE, all originating at the National Physical Laboratory and known as Schemes A, B and Z. The one of greatest interest and most widely used is Scheme B. This will now be described only briefly, as adequate accounts already exist elsewhere (Ref.12).

Matrices are all stored on the drum, the elements of each row being stored consecutively. The size of the matrix is stored with the matrix itself, as is the number of binary places and the grand sum of all the elements and these parameters. This sum is used as a check against machine failure.

The core of Scheme B is the General Interpretive Program (G.I.P.). To date (September 1957) four versions of the G.I.P. have been published, each one with rather more facilities than its predecessor. The latest of these G.I.P.'s is G.I.P.5*. The use of these programs is very similar and only G.I.P.5 will be described.

The program stores a series of sub-programs, called 'Bricks', selected by the programmer for his particular job. The bricks are obeyed under the control of a sequence of codewords. Writing these codewords is the only programming required if the necessary bricks are available. The codewords inform the G.I.P. about what is to be done next: this may be the execution of a brick or some administration, such as counting, of the code words themselves.

A brick must satisfy certain simple conditions. These are:

- (i) it uses one of two special initial cards;
- (ii) it leaves 12_0 and 12_{16-31} untouched;
- (iii) it leaves TS 13 clear at the end;
- (iv) it starts with either 9 - 24 or 12 - 24;
- (v) it has no punching in Hollerith column 1;

* G.I.P.4 was an experimental program, suppressed by its author.

- (vi) it ends with $12 - 1$ (32 m.c.), leading to 1_{30} ;
- (vii) it expects to find any parameters, a,b,c, it needs in $18_{0,1,2}$ respectively.

Most of these conditions are dictated by the G.I.P.

The bricks required to perform a complete program are first assembled and numbered sequentially, starting at 1. Some bricks consist of more than one section - such bricks count as if they were several consecutively numbered bricks. The codewords will refer to the bricks by the numbers they have been given.

The assembled bricks are now ready to be fed into the DEUCE. The complete pack is made up from

G.I.P.

Parameter Card, with N P17 in binary on the Y row where N bricks are to follow.

Bricks 1, ..., N (in sequence).

Codes.

Data (as required).

It is recommended that the bricks fill the DL's in the same order as ordinary programs, i.e. highest numbered DL first, but this is not essential.

The codewords are all of the form

a, b, c, r.

These quantities all have eight binary digits and are stored x P1, x P9, x P17 and x P25 respectively.

If $r \leq 31$ the numbers a, b, c are the parameters required by the brick whose number is r. This implies that not more than 31 programs (bricks or sections of bricks) may be used. There is a method, given below, for increasing this number. Before a brick is entered the G.I.P. will place a, b, c in $18_{0,1,2}$, all stored as multiples of P5.

Values of $r > 32$ refer to operations on the codewords. In this case the numbers a, b, c refer to the numbers of the codewords themselves. 96 codewords are allowed, and these are placed in 'codestores' 0 - 95. There are some extra codestores, 96 - 127, used by the G.I.P. Some of them contain useful constants. We denote the contents of codestores a, b, c by A, B, C. The full list of functions available is given in Table 1, on Page 140.

The codewords are normally 'obeyed' sequentially but the functions $r = 41, \dots, 45$ perform operations with the codewords that are similar to the use of destination 0. Generally B will be a count and A a basic codeword. We can then have a loop repeated with slightly varying parameters, e.g. the matrix power series in example 2.

TABLE 1

Functions of G.I.P.5

A, B, C denote the contents of codestores Nos. a, b, c, where these are the parameters specified in a codeword.

Meaning of r for values ≥ 32

r = 32	If $b = 0$ replace section a onwards, reading in c sections altogether (with the provision that at least $(a - 1)$ sections must be in store already). If $b \neq 0$ pass one brick through the reader and obey it, using the given values of a, b and c. If $b = c = 0$ read data card bearing N . P ₁₇ on Y row and N sections of program (numbered 1, ..., N).
r = 33	Jump to code No. c.
r = 34	Jump to code No. c if A is negative otherwise jump to b.
r = 35	Jump to code No. c if A is non-zero otherwise jump to b.
r = 36	Jump to code No. c if $B > A$ otherwise continue normally.
r = 37	Jump to code No. c if $B \neq A$ otherwise continue normally.
r = 38	Replace C by $A + B$ and proceed to next code.
r = 39	Replace C by $A - B$ and proceed to next code.
r = 40	Replace C by A and proceed to next code.
r = 41	Obey $A + B$, add P ₁ to B and jump to c.
r = 42	Obey $A + B$, add P ₉ to B and jump to c.
r = 43	Obey $A + B$, add P ₁₇ to B and jump to c.
r = 44	Obey $A + B$, add C to B and continue normally.
r = 45	Obey $A + B$ and jump to c.
r = 46	Jump to a, proceed until $(b + 1)$ is about to be obeyed then jump to c ($b \neq 0$).
r = 47	Read a set of 32 codes into DL $9 + c$ m.c.0 onwards and jump to b ($c = 0, 1$ or 2).
r = 48	Put a, b, c in QS 17 for next program.

The following special constants are available in codestores 120 - 127.

Codestore	
120	Zero
121	P ₁
122	P ₉
123	P ₁ + P ₉
124	P ₁₇
125	P ₁ + P ₁₇
126	P ₉ + P ₁₇
127	P ₁ + P ₉ + P ₁₇

Example 1 To form XY^n from given matrices X, Y.

We shall require bricks to read matrices, to multiply them and to punch out the result. The matrix multiplication, however, multiplies XY' (matrices are stored by rows and this arrangement is more convenient). The bricks, then, are

<u>Name</u>	<u>Sections</u>	<u>Numbered</u>
Read Matrix	1	1
Transpose Matrix	1	2
Multiply Matrices	3	3
Punch out Matrix	1	6

The necessary codewords are

<u>No.</u>					<u>Remarks</u>
0	0	0	60	1	Read Y, store on track 60 onwards.
1	60	0	90	2	Transpose Y; store on track 90 onwards.
2	0	0	60	1	Read X, store on track 60 onwards.
3	9	0	10	40	Set parameter (count)
4	60	90	60	3	Replace XY^{r-1} by XY^r .
5	10	121	10	39	Reduce parameter by 1.
6	10	7	4	35	Test count: if zero go to codeword 7, otherwise codeword 4.
7	60	0	0	6	Punch out XY^n .
8	0	0	0	33	Return to codeword 0, for new matrices.
9	n				
10	(n-r)				Count.

In this example we have made the limitation that the matrices X and Y do not take more than 30 tracks of the drum.

The G.I.P. starts by obeying codeword 0. This says that brick No.1 is to be obeyed and that it is to have the parameters

Zero in 18_0

Zero in 18_1

60.P5 in 18_2

Brick No.1 is 'Read Matrix' and a matrix will therefore be read and stored on the drum, starting at track 60. When the whole matrix is in the store control goes back to the G.I.P. which then selects and obeys codeword 1, etc.

Codeword 3 has $r = 40$ and so performs an operation on other codewords. In fact, the contents of codestore 10 are replaced by the contents of codestore 9. When this has been done we proceed with codeword 4, which

multiplies the two matrices, XY^{T-1} and Y , and stores the product XY^T in place of XY^{T-1} . Codeword 5 reduces the count by 1 - codestore 121 contains a P1. The next codeword examines the count: when this has reached zero we wish to punch out, i.e. obey codeword 7, otherwise we return to codeword 4.

Example 2 To form the matrix power series $A_0 + A_1 X + \dots + A_n X^n$.

We shall assume that the matrices are sufficiently small to take up only one track each. The bricks required are

	<u>Sections</u>	<u>Numbered</u>
Read Matrix	1	1
Transpose Matrix	1	2
Add Matrix	1	3
Multiply Matrix	3	4
Punch Matrix	1	7

Codewords

No.

0	0	0	100	1	Read X, store on track 100.
1	100	0	110	2	Store X' on track 110
2	3	17	4	43	Read $A_0, A_1, \dots, A_n$ to tracks 10, 11, ... (n + 10)
3	0	0	10	1	
4	17	15	2	37	
5	16	0	17	40	New count.
6	7	17	9	45	$A_n X$; stored at 120. $S_r \cdot X$.
7	10	110	120	4	
8	120	110	120	4	
9	17	121	17	39	Reduce count by 1.
10	11	17	12	45	$S_r = S_{r+1} \cdot X + A_r$. Jump to 8 if (17) $\neq$ 0.
11	10	120	120	3	
12	17	13	8	35	
13	120	0	0	7	Punch out $S_0 = \text{sum}$.
14	0	0	0	33	Return to codeword 0.
15	0	0	n+1	0	
16	n	0	0	0	
17	0	0	0	0	Used as working space for counts.

NOTE 1 The counter in codestore 17 is never cleared.

NOTE 2 The sequence of A's does not start at track 0, as this is used by the Matrix Multiplication.

It is possible to insert actual DEUCE instructions into the sequence of codewords. Any codeword with a P31 is regarded as a DEUCE instruction and is obeyed Q30.

G.I.P.5 allows more than 31 bricks. Those whose number is greater or equal to 32 include a P32 in the codeword and the six lowest digits of r are the number of the brick.

The digits of r are therefore used as follows:

P₃₁ not punched P₃₂ punched, P₂₅ $\overline{14}$ P₃₀ = Number of bricks.
P₃₁, P₃₂ " " P₃₀ " , P₂₅ - 30 = Number of G.I.P. function.
P_{30,31,32} " " , P₂₅ - 29 = Number of bricks.

Although programming a series of codewords is relatively simple, there must be some method of 'program testing'. G.I.P.5 includes facilities for stopping on a particular codeword and inserting new codewords from the I.D. switches. These are described in Ref.12, which also gives a list of Scheme B bricks available (in Summer 1957).

The N.P.L. interpretive program provides a useful method for making up a working program in a short time. However, if the program is to be used frequently it may be preferable to write a special master routine and use the Program Changing Routine (R.A.E.180). This is generally rather quicker in use, although it requires more programming. Bricks that satisfy the standard conditions may be used with R.A.E.180, which was, in fact, designed with this object in view. The G.I.P. is used mainly for matrix calculations but there is no such limitation implied in its construction: it is truly a general program and may be used as such, provided that the necessary bricks are available.

Cards are available, issued in the same way as subroutine cards, which give details of the latest 'bricks' produced. These should always be consulted before making any programs.

IV.7 Matrix Schemes A, B, Z

Standard Bricks are (at present) in 3 categories:-

- Scheme B Standard Bricks which require the specification of any or all of parameters a, b and c in $18_{0,1,2}$
- Scheme A Standard Bricks which do not belong to Scheme B and which, if they are linear algebra programmes, use the standard stores (when required) for a matrix, a diagonal and a scalar.
- Scheme Z Standard linear algebra bricks which use a non-standard matrix store.

Scheme B bricks may use up to 6 parameters. The 3 extra parameters are put into $17_{0,1,2}$ from the previous codeword by "Auxiliary Codes" (ZCO 3B).

It is thus apparent that if Scheme B bricks are to be obeyed in sequence from the reader (which is not usual) they must somehow be provided with the parameters a, b and c in $18_{0,1,2}$. This may be done by preceding each Scheme B brick with a special Scheme B brick:- "Read a, b, c, r" (ZCO 4B) which will read the parameters (from a card) into 18.

Scheme A storage

In Scheme A the drum is used to store a single matrix (or vector) starting at track 0 (0/0), element (1,1) being stored in m.c.0 and the elements following continuously in successive m.c.'s, row by row, without row sums, ($m \times n \leq 8192$).

D.L's 4-10 are used to store a diagonal matrix as a vector of order N. ($N \leq 224$). DL 11 can also be used if the drum is not in use ($N \leq 256$). The elements are stored in reverse order from 40.

$12_{9,10,11}$ and $12_{13,14,15}$ are used to store two scalars. Note that the order of the three quantities in each case is different from that on a scalar card.

The parameters for the matrix and the diagonal, and the scalars, are stored in DL 12 in the following manner:-

<u>DL 12 m.c. 0</u>	<u>Empty</u>	
1	m. P17	Referring to matrix.
2	n. P17	
3	k. P17	
4	P_A P17 (or D.P1)	Referring to matrix.
5	$\Sigma \Sigma_A$ (grand sum)	
6	P_d P17	Referring to diagonal.
7	Σ_d (sum)	
8	N. P17	
9	R	1st scalar.
10	P_R P17	
11	(R + P_R P17)	
12		2nd scalar.
13	S	
14	P_S P17	
15	(S + P_S)	

Any matrix or diagonal which is punched out has its row sums (or sum) shifted down k places, where k is the number stored in 12_3 at the time. The value of k read in with a matrix is not here.

To obtain a value of $k \neq 0$, a brick (LR05A) is used to read in k. P17 from the Y row of a parameter card.

To avoid any errors due to binary place values being lost from the store, the sum checks for matrix, diagonal and scalars all have the number of binary places added in. Binary place values are always stored and

punched in the P17 field and decimal place values always in the P1 field. The sum for the diagonal has the number of elements N. added in as well as the number of binary places.

The scalars R and S are not preserved longer than necessary, and programmes which wipe out any of DL's 4-10 are also allowed to use the diagonal parameter positions. 12₀ to 12₅, however, are always preserved except when reading data or a "k" card.

Matrix mult. and diagonal pre-and post-mult. programs examine the m's, n's, etc., and give a failure indication if these are not compatible.

The largest element of a matrix, vector or diagonal, and a scalar should "reach" P31, i.e. P31 and P32 should be different.

Scheme B storage

In Scheme B several matrices or vectors can be stored on the drum at the same time, and diagonal matrices are also stored on the drum (as vectors, usually row-vectors). Each matrix, etc., starts on a fresh track, but the elements are stored consecutively, row by row, without row sums, element (1,1) being stored in m.c.4 of the first track used, and being preceded by:-

m.c. 0 EE (the grand sum of the matrix and its parameters).

m.c. 1 m. P17

m.c. 2 n. P17

m.c. 3 p. P17

Note that EE is a single length, unshifted sum, truncated at the most significant end.

This is referred to as the "block-floating" form of a matrix and is the usual form. Each element has the same number of binary places, p. Matrices can also be stored in the "fully-floating" form in which each element is in standard floating binary, stored as a word-pair with the exponent preceding the mantissa, these counting in the parameter n as separate quantities, so that the order of the floating matrix is m by $\frac{1}{2}n$. In this form, the binary-place value is never referred to, and may be put equal to zero on the parameter card.

The Scheme A scalar stores are also used in Scheme B. It should be noted that the DL 12 referred to in Scheme B is not the same as that associated with G.I.P. One is on the drum when the other is in the high-speed store.

The parameters a, b and c in G.I.P. codewords: when used with Scheme B bricks and referring to matrices, vectors or diagonals are the numbers of the first tracks occupied by the respective matrices etc. (tracks being numbered consecutively from 0 onwards). When a matrix etc. is to be read from the drum the parameter a is used; b is used for a second matrix etc. to be read, as in Matrix Mult., Addition, etc., and c is used to specify the position for writing a resultant matrix, etc., on the drum.

For reference purposes, the elements of a matrix are considered to be numbered consecutively on the drum, starting at 0. An element of a fully-floating matrix consists of two consecutive words.

It is important to keep the unused part of the last track of a matrix cleared, since most Scheme B programs add a whole track at a time into the sum.

Scheme B matrix programs normally produce matrices in which the largest element does not quite "reach" P31, i.e. has P31 and P32 digits the same and differing from the P30 digit. When a matrix has been produced by extracting elements from another matrix, this does not apply, (unless one of the elements reaching P30 happened to be selected) and for the maximum accuracy in subsequent use such an extracted matrix should first be normalised to the above form by multiplying it by the scalar + 1 (or adding it to a zero matrix). In general it is unlikely that capacity failures would occur through presenting numbers which reach P31, though this is possible, e.g. if two matrices in Scheme A form are read in and added together without being normalised. This case will pass the checks and give wrong results. A special dec.-bin. conversion program (LK08/1) has been produced to avoid this.

Scheme Z

Standard linear algebra bricks which do not use the Scheme A or Scheme B methods of storage have been put into this category. They include bricks for matrix inversion, for solving equations and for finding latent roots. For matrix inversion and the solution of equations, the matrix is stored starting in track 0, with each row on a fresh track. There are bricks for reading in a matrix for inversion or equations for solution and storing them in this form. There are also bricks for taking a matrix from the Scheme A or a Scheme B store and putting it in the correct form for inversion.

Latent root bricks use the Scheme A store for the matrix but not the Scheme A diagonal store for the vector; they are therefore put in Category Z.

IV.8 Simplified programming, the "Alphacode"

The reader has no doubt noticed by now the vast amount of detailed clerical work that goes into writing programs. Not only is it necessary to plan the computation but many extra instructions have to be included, to take care of transfers of data and orders, to count, to modify instructions, and so on. In fact, all of the administration that is needed for use of the computer has to be supplied by the programmer. The previous sections have described some of the means whereby simplifications can be achieved: all of these methods, however, have required something in the way of programming knowledge, in the sense that problems cannot be presented to the computer in anything like their natural formulation. The Alphacode, described briefly below, is an attempt to overcome this objection. A full account is given in Ref.4.

All that is needed for the use of Alphacode is the ability to state a problem in algebraic terms and to give a sequence of steps by which the answer may be calculated. The result of each step is to be calculated from two numbers available previously, either as the result of calculation or as initial data. Many of the commonly occurring mathematical functions, such as square root, logarithm and exponential, trigonometric functions, are included in the available Alphacode instructions. In addition, there are more sophisticated ones like 'Solve Differential Equations' and 'Sum Series'.

The store of the computer may be thought of as containing a large number (about 2500) of variables, called X1, X2, X3, ... Arithmetical

'instructions' will refer to three (sometimes only two) of these variables, in forms like

X1 = X2 PLUS X3

or

X4 = EXP X5

or

X6 = X7 MOVED

In each case, the result of the operation demanded on the right-hand side of the equality sign replaces the number previously held in the position indicated on the left. The third instruction does a simple replacement. It is written like this because it is necessary to put something in the 'Function' column (see below).

When writing a program in Alphacode we set out the complete list of instructions in a series of columns, called

R	A	B	FUNCTION	C	D
---	---	---	----------	---	---

The R and D columns have special significance and are not used for every instruction. The equality sign is taken to be between columns A and B, it is not actually written in. What is written in these columns is then punched onto cards, where particular columns have particular significance.

The R and D columns are connected with discrimination instructions*. Normally, instructions are obeyed in strict sequence. When a discrimination occurs the machine requires a choice of next instruction: one of these is the instruction following, the other is given by the D column. However, it would be wasteful to name every instruction and so only those to which reference will be made, in a discrimination, are given reference numbers in the R column. These 'labels' need not be in numerical order. It is possible to give 99 different labels.

As we have seen earlier, much of the power of the modern computer derives from the ability to modify instructions and to repeat loops of instructions. These facilities are available in Alphacode. In addition to the variables there are 63 counters, N1, ..., N63. These are whole numbers and may be used for simple counting and modification. This is done by the instruction 'MODIFY', as below

N1	N2	MODIFY	N3
X4	X5	DIVIDED	X6 .

The effect of MODIFY is that the instruction following it has the numbers standing in N1, N2, N3 added to the addresses of the variables in the A, B and C columns. In the example above, if all the counters are zero the instruction will be the same as

X4	X5	DIVIDED	X6 .
----	----	---------	------

If N1 = 10, N2 = 20, N3 = 38 the effect would be that of the instruction

X14	X25	DIVIDED	X44 .
-----	-----	---------	-------

Not all variables occurring need be modified, and the modifiers may all be the same or different (as here).

* The D column also determines the type of punching on output.

There is a special counting instruction, UP TO, which increases a counter by unity, tests it for equality with the upper limit, and returns to the instruction named in the D column as long as the condition is not satisfied.

It is possible to include in a program subroutines of Alphacode instructions. They are numbered S1, S2, ..., up to S27 for any particular program, and are entered by the instruction

JUMP S1

and left by

END OF S1.

No link is necessary as the Alphacode interpreter program remembers from which instruction the subroutine was entered and returns to the next one.

With all these facilities writing a program in Alphacode is a very simple matter. A program that would take some weeks to write and test in machine instructions may be written in a matter of hours. The price paid, however, is heavy in machine time. The Alphacode works with floating point arithmetic, which slows down all calculation by a large factor, and then all instructions have to be interpreted every time they are obeyed, again a slow process. If much of the time is spent on input and output the loss of time may not be great - the reader and punch work at the same speed for orthodox programs and the Alphacode. The main virtue of the Alphacode (as indeed of any interpretive program) is that it allows the rapid writing of programs for a 'one-off' job. The range of numbers is so vast that there are no scaling difficulties, it is possible to concentrate on the calculation and there is little need to worry about the red-tape instructions.

One other use of the Alphacode is that it allows the testing of the logic of an otherwise long and difficult program. It is not difficult to obtain some test results that may be used to check a program written in the normal instruction code. Finally, the floating point arithmetic can give an indication of the scaling required to write a faster program used fixed point arithmetic.

IV.9 An introduction to logical design

This section is intended as a brief introduction to the subject known as 'logical design'. This is the study of how to make a computer perform its functions, as distinct from programming. Circuit design is not included.

Many of the ideas of logical design are quite general and are applicable to all machines. Those examples that are given in this section are all concerned with DEUCE.

We must first learn some notation and terminology. The symbols used are shown in Fig.4.

- | | | |
|-------|-----|--|
| Delay | ... | This is a device for delaying pulses by a stated amount. The effect is that a given pulse appears at a time different from its normal time. In a serial machine the effect is to multiply by a power of 2. |
| Gate | ... | This has many inputs and one output. It has threshold n if it gives an output pulse when at least n of the inputs are active. (The threshold is normally written inside the circle representing the gate. A 1-gate normally does |

not have a '1' written in). The output lasts as long as the input.

- Trigger ... A trigger will store one binary digit. It can be set or cleared. When a trigger has been set it gives a continuous stream of pulses until it is cleared. Like a gate it has a threshold.
- Inhibitor ... This overrides all the inputs to a gate or trigger. It is normally written with a small circle on the gate or trigger. This is the way a trigger is cleared.
- Inverter ... This gives the ones-complement of a word, i.e. all 0's become 1's and vice versa.
- Clock pulses ... Abbreviated to C.P. These are standard pulses available inside the machine at all the digit times. They are generated by an accurately controlled oscillator.

By convention pulses go from left to right (across the page). The least significant digits always lead. (Note that this is the opposite way to normal DEUCE practice.)

All the functions available on a machine are built up of units like these. The most important single unit is probably the 2-gate, as this allows a direct comparison of digits. Fig.5a shows the simplest function a 2-gate can perform, giving the logical '&', i.e. Source 25. There is an output only if both inputs are active.

A very important part is played in logical design by the function $\frac{1}{2}$. This can be made up from two 2-gates and inverters, as shown in Fig.5b. The outputs from the 2-gates are joined. The mode of operation is as follows. If $a = b$ each gate has only one active input, both outputs are 0 and the result is 0. If $a \neq b$ one gate has two inputs and output 1 and the other has no input at all, so that the result is 1.

Fig.5c illustrates the principle of a delay line. The delay line is N digits long, i.e. pulses take N digit times to travel down the line. On emerging from the delay line they are 'clocked' with the always available clock pulses. This reshapes the pulses as they become distorted in travelling through the delay line. Thereafter the output is taken to two places, going to the 'source gate' and also being led back to the entry to the delay line. The pulses can go through the source gate only if this source has been selected. In this case there is a stream of pulses on the line marked S and the contents of the delay line can flow out to the 'highway' and their required destination. The line S will be active only for as long as this particular source is called. At the other end of the delay line is the destination gate. This is slightly more complicated than the source gate. Normally, for ordinary recirculation, there is no signal on the line D. Pulses therefore go back into the delay line and nothing can go through the 2-gate. If we nominate this delay line as destination, the line D becomes active. Recirculation is therefore inhibited and whatever is on the line x flows into the delay line. As soon as transfer ends ordinary recirculation will recommence.

Of course, this is an idealised picture of a delay line. There is great attenuation of pulses as they travel down the line and therefore it is necessary to amplify the pulses. Such considerations, however, do not belong to logical design but are matters for the circuit engineer.

In DEUCE the next instruction gate comes just after the destination gate, before the pulses enter the delay line. This allows us to send an instruction to a particular destination and simultaneously call for this as the next instruction.

Fig.6 shows two possible adders. The first, Fig.6a can be used only on a serial machine. Its action is fairly obvious. The carry on the input is the carry from the last addition of a pair of digits. The 'new carry' must be delayed to act as input carry for the next pair of digits.

The second system uses what are known as 'half-adders'. Instead of adding the two digits and the carry digit at the same time, the half-adder only has two inputs. One half-adder is therefore used to add a pair of digits and the second half-adder adds the carry to the result. Binary addition can be represented as below:

In	a	0	0	0	0	1	1	1	1
	b	0	0	1	1	0	0	1	1
	Carry	0	1	0	1	0	1	0	1
Out	Digit	0	1	1	0	1	0	0	1
	Carry	0	0	0	1	0	1	1	1

N.B. The columns in this example are all independent.

The half-adder does this in two stages. The first, (left hand part of Fig.6b) corresponds to

In	{	a	0	0	1	1	
		b	0	1	0	1	
Out	{	Partial Digit	0	1	1	0	= a $\oplus$ b = d
		Partial Carry	0	0	0	1	= a & b

If the partial carry is a 1 this will definitely be the carry for the next pair of digits. This carry may therefore be taken directly to the output of the second half-adder, as 'new carry'.

The second stage consists of adding the carry from the previous digit pair to the partial digit. In symbols this is

In	d	0	1	1	0	
	Carry In	1	0	1	0	
Out	Digit	1	1	0	0	(d $\oplus$ carry)
	Carry Out	0	0	1	0	(d & Carry)

The system of using half-adders has the advantage that it is easily adapted for use on a parallel machine.

Fig.7 shows the arrangement required with TS 14 to give the two functional sources, source 23 and source 24, connected to TS 14. Shifting a number up entails very little difficulty, as the number has only to be delayed. Shifting down, however, means that the P2 has to be available at the time of a P1, etc. The pulses must be early. The following arrangement is therefore adopted. The delay line itself is only 31 digits long instead of 32 as would be expected. The extra digit time of delay is inserted outside, between the points marked A and B. This allows us to use pulses that have been delayed by the full 32 digit times and pulses that have been delayed by 31 digits only, and are therefore a digit early.

Source 14 presents no difficulty at all; this is exactly like the delay line of Fig.5c. To obtain source 24, which shifts up, we must take the output at B and delay it by a further digit. This multiplies by 2 but gives us a possibly incorrect digit as P1. The effect of the extra digit delay is that the P32 digit is not available until P1 time. We, however, wish to force the P1 digit to be zero and therefore inhibit the source 24 gate by a single clock pulse at P1 time.

Source 23, which shifts down, takes the output of the delay line at the point A. This gives the correct digits until the P32 digit of source 23. Here we must copy the P32 digit of TS 14. We therefore take this digit from B, allowing it through a 2-gate which is open only at P32 time and at the same time inhibit the train of pulses from A.

Fig.8 shows how the output staticizer works. This must convert a train of pulses into a parallel representation. Here we have 32 triggers, each of which will store one digit. The number that is sent to destination 29 flows along the line marked 'source'. Its digits go through the respective 2-gates, which are opened at P1 time, P2 time, etc. The triggers are therefore set up in turn. Each trigger is connected to a lamp on the console. There is also a connection to the punch knives. As soon as a row has been punched the triggers must be cleared so as to be ready for the next row.

A similar arrangement is used on the input dynamiciser, which converts the parallel input into a sequence of pulses.

The last illustration of some of the ideas mentioned is the working of TS Count. Obviously it is impossible here to give a full account of how the most important part of DEUCE works, but the mechanism whereby transfer starts at the right time will be explained.

It takes one minor cycle for an instruction to flow into TS Count. This is the selection minor cycle. Thereafter the instruction circulates in TS Count, but 1 is subtracted from the wait and timing numbers during every minor cycle. The subtraction is not a true one, however, in that only the five digits of the wait and timing numbers are concerned, i.e. there is no carry into the P22 or P31 digits. The subtraction is done by changing every 0 into a 1 until a 1 is reached. This digit is changed into a 0 and the other digits are left unchanged. As an example, subtraction of 1 from 00011 (24) gives 11101 (23). Just before the P17 digit emerges from TS Count a trigger ('Carry') is put on. Carry is put off again by the first non-zero digit of the wait number. The output of Carry then goes into a 2-gate with a pulse at P22 time. The output of this gate puts on another trigger (trigger R) which starts the transfer timing (TT) signal at the beginning of the next minor cycle. Trigger R can be put on only if 'Carry' is still on at P22 time, which can happen only if the wait number was zero.

Selection of the next instruction is done by a similar counting down procedure on the timing number. However, an instruction must be obeyed before its successor can be allowed into TS Count, and if the timing number

is smaller than the wait number it would appear that the next instruction is presented to TS Count too early. This is the reason for trigger R. The Transfer Control Instruction (TCI) signal which is normally controlled by the timing number can be put on only if trigger R is also on. Therefore, if $W > T$ the machine will ignore the first time that the timing number is counted down to zero.

The reason for the minor cycle gap between instruction is now clear. During the last minor cycle of transfer the next instruction flows into TS Count. During the next minor cycle the counting down takes place for the first time. Even if the wait number is zero the machine does not realise this until P22 time of this minor cycle, which is called the set-up minor cycle.

Here it is impossible to do more than mention some of the ideas that go into the design of computing machines. The interested reader is referred to the books by R.K. Richards (Ref.6) and by M.V. Wilkes (Ref.10) for a full discussion of the principles involved. Ref.2 is concerned solely with DEUCE.

PART V EXERCISES AND SOLUTIONS

V.1 Exercises

The figures in brackets are the number of instructions in the solutions given here.

Exercise A

At the beginning of each example TS 13, 14, 15, 16 contain w, x, y, z respectively. To multiply fractions use the method explained in I.4 before Example 11.

1. Put $w+z$ in 13 [1]
2. Put $x-y$ in 13 [2]
3. Put $2w+x$ in 13 [2]
4. Put $w+2x$ in 14 [1]
5. Put $2y-z$ in 13 [3]
6. Put $4x$ in 14 [1]
7. Put $16w$ in 13 [1]
8. Put $1/8x$ in 14 [1]
9. Put $1/2w$ in 13 [2]
10. Put $10x$ in 13 [3]
11. Put $2p-5q$ in 20_2 where $p = C(19_2), q = C(19_3)$ [5]
12. Put xy in 15 [2+1 multiplications]
13. Put $xy-wz$ in 13 [8+2 multiplications]
14. Put p^2-q^2 in $20_2, 2pq$ in 20_3 , where $p = C(19_2), q = C(19_3)$ [11+2 multiplications]

15. Put az^2+bz+c in 14 where a,b,c are the contents of $17_0, 1, 2$ respectively [7+2 multiplications]
16. $C(15) = 2r-1$, $C(16) = r^2$, where r is an integer. Replace them by $2(r+1)-1$ and $(r+1)^2$ respectively. [5]

Exercises B

At the beginning of each example TS 13, 14, 15, 16 contain w, x, y, z respectively, and $19_2, 19_3$ contain p, q .

1. Put $-|x|$ in 14. [5]
2. Put $|w-x|$ in 14. [5]
3. Given $x \geq 0$, put the larger of x and $|z|$ in 13. [8]
4. If $z < 0$ put p in 21_3 , but if $z \geq 0$ put q in 21_3 . In either case put $-z$ in 16. [7]
5. What happens in question 3 if $z = 0$? Modify your program so that $\frac{1}{2}(p+q)$ is placed in 21_3 in this case. [12 in all]
6. 18_0 contains either $3/8$ or $5/8$. Whichever it is, replace it by the other. You may assume that any constants you need are in DS 20. [3]
7. a,b,c are the contents of $17_0, 1, 2$ respectively. Put the largest in 14 and test whether the three numbers can be the lengths of the sides of a triangle. [The sum of any two sides of a triangle must be greater than the third]. [15]

Exercise C

$x = C(16)$, $a = C(19_3)$. Evaluate $ax^n + ax^{n-1} + \dots + a$ and place it in 14.

- Hints: (i) $ax^n + ax^{n-1} + \dots + a = (\dots(((a.x+a).x+a).x+a)\dots)x+a$.
If $S_{n-r} = ax^r + ax^{r-1} + \dots + a$, $S_{n-r-1} = x S_{n-r} + a$, and S_0 is the result required.
- (ii) Program $S_{n-r-1} = x S_{n-r} + a$.
- (iii) Keep a counter $(n-r)$. Program reducing it by 1 and testing for 0.
- (iv) Set the initial conditions. n will have to be kept in the delay line with the program.

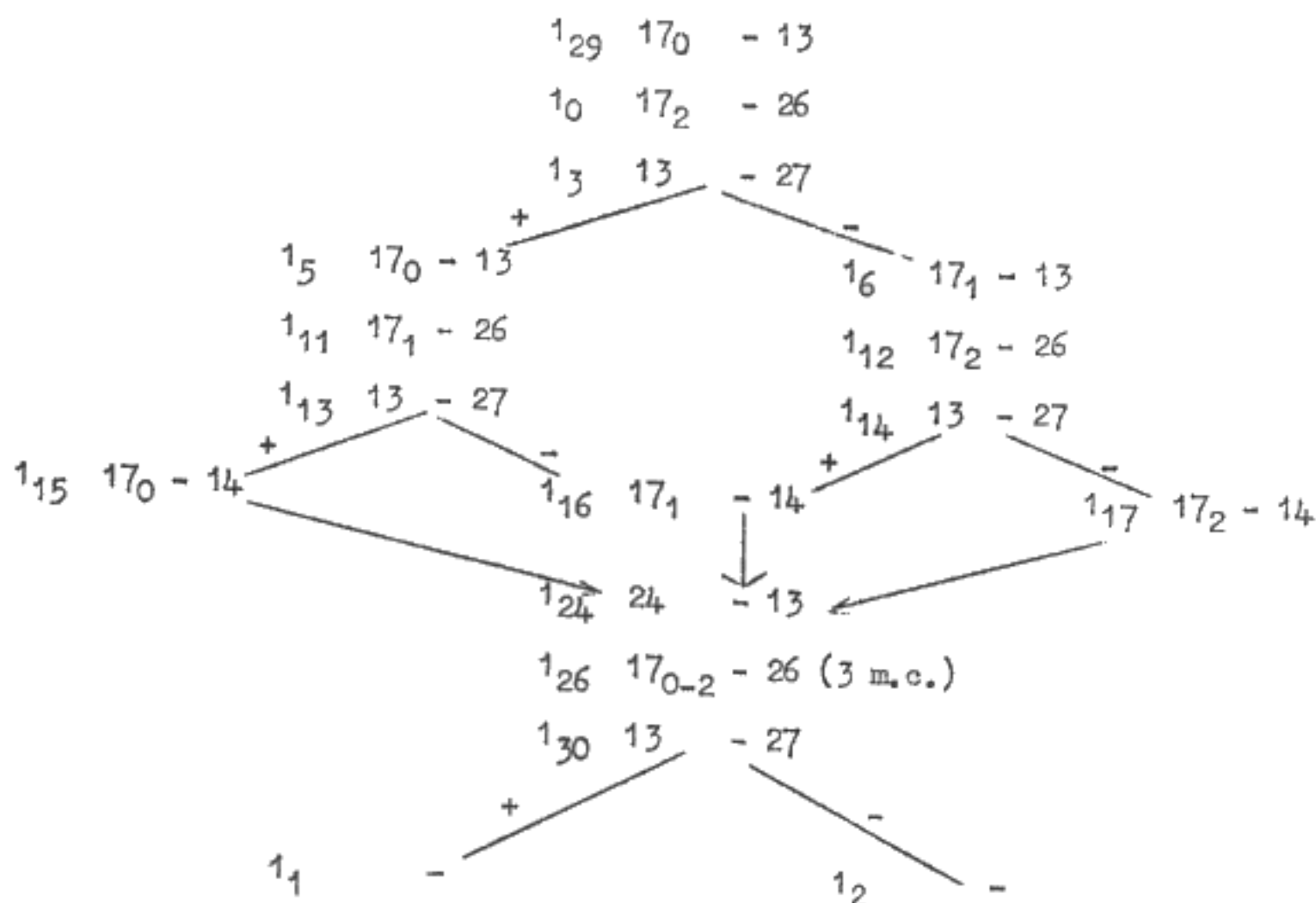
Exercises D

1. Code the following program. It has been constructed solely to test the ability to code instructions: it is meaningless and it is not a good example of optimum coding.

1_{17}	2_{19}	-	1_3	
1_{19}	4_{21}	-	2_5	
1_{23}	12_{26}	-	2_6	←
1_{28}	1_3	-	5_{31}	
1_7	12_{26}	-	1_4	
1_{11}	11_0	-	1_5	
→ 1_{21}	1_4	-	12_{15}	
1_{16}	1_5	-	11_{30}	
1_{30}	1_8	-	5_8	
1_{24}	3_{29}	-	17_1	
1_3	18_0	-	6_8	
1_9	7_{15}	-	21_3	
1_{20}	19_2	-	5_0	
1_5	20_3	-	1_3	
1_{13}	1_3	-	21_2	
1_{18}	17_3	-	1_4	
1_{25}	1_4	-	17_0	
1_{29}	19_3	-	17_1	
1_6	17_2	-	20_2	
1_{14}	1_3	-	8_{16}	
1_{15}	18_2	-	2_7	
z			nz	
1_{22}	1_4	-	12_{16}	
1_0	$12_{4,5}$	-	$21_{2,3}$	(2 m.c.)
1_8	$17_{2,3}$	-	$11_{14,15}$	(2 m.c.)
1_{26}	$19_{2,3}$	-	$18_{2,3}$	(2 m.c.)
1_1	2_0	-	2_2	(2 m.c., e.o.)
1_{12}	$5_{31,0}$	-	$7_{31,0}$	(2 m.c.)
1_{31}	6_{25-27}	-	17_{1-3}	(3 m.c.)
1_{27}	12_{1-4}	-	18_{1-0}	(4 m.c.)
1_4	1_8	-	1_7	(4 m.c.)
1_{10}	2_3	-	1_4	(21 m.c.)
1_2	2_4	-	1_4	(5 m.c.)

How long does the program take to go once round the inner loop, and how long round the outer loop if the inner loop is obeyed N times? [Hint: Put a small asterisk between two instructions if minor cycle 0 is passed while the first is being obeyed; e.g. the first two come after the instructions in 128 and 17.]

2. Code the following program - exercise B.7 with the locations of the instructions allotted.



How long does this program take if, using the notation of exercise B.7, the three numbers a , b , c can be the sides of a triangle and

- (i) a is the greatest.
- (ii) b is the greatest.
- (iii) c is the greatest.

Exercise E

In these and the following exercises you can assume that the following subroutines are already in the store:

- A 12/1 (No.222) in DL 2; multiplication, division, summation of series.
R 01/3 (No.228) in DL 3; read one decimal fraction.
P 01 (No. 12) in DL 4; punch one decimal fraction.
F 04/1 (No.136) in DL 5; square root.
T 01/2 (No.195) in DL 6; sine and cosine.

A 12/1. Multiplication. Enter in 2_8 .

$a \times b = c$, rounded. All numbers to 30 b.p.

Short stores used	13	14	15	16	21_2	21_3
Contents at entry	Link	a	-	b	-	-
Contents at exit	Link	a	-	b	-	c

A 12/1. Division. Enter in 2_7 .

$a \div b = c$, rounded. All numbers to 30 b.p. Buzzes and fails unless $-1 < c < 1$.

Short stores used	13	14	15	16	21_2	21_3
Contents at entry	Link	a	-	b	-	-
Contents at exit	Link	2c	-	b	c	-

A 12/1. Summation of series. Enter in 2_6 .

$c = a_0 + a_1 x + \dots + a_n x^n$. The $(n + 1)$ coefficients are stored in A_m, A_{m+1} up to A_{m+n} — a_0 is in A_m and all the coefficients are in the same delay line ($n \leq 31$). All numbers to 30 b.p.

Short stores used	13	14	15	16	21_2	21_3
Contents at entry	Parameter	Link	-	x	-	-
Contents at exit	-	-	-	x	-	c (30b.p.)

The "Parameter" planted in TS 13 is $AP_5 + (m + n)P_{17} + (n - 1)P_{22}$; if $m + n \geq 32$ it is $AP_5 + (m + n - 32)P_{17} + nP_{22}$.

N.B. n is the order of the polynomial, not the number of terms.

R 01/3. Read one decimal fraction. Enter in 3_{28} .

The fraction, c , is punched on a card with sign in DEUCE column 1 (Y row for +, X row for -) and ten decimal digits in columns 2 to 11. The decimal point is between columns 2 and 3. $-2 < c < 2$. Result given to 30 b.p.

Short stores used	13	14	15	16	21_2	21_3
Contents at entry	Link	-	-	-	-	-
Contents at exit	-	-	-	-	c	-

N.B. The subroutine fails if the card is punched wrongly.

P 01. Punch one decimal fraction. Enter in 4_{28} .

The fraction, a, given to 30 b.p. is punched in decimal in the same way as the cards R 01/3 reads.

Short stores used	13	14	15	16	20_2	20_3	21_2	21_3
Contents at entry	Link	a	-	-	-	-	-	-
Contents at exit	-	-	-	-	-	-	-	-

N.B. The published description of this subroutine failed to mention that DS 20 is used.

F 04/1. Square root. Enter in 5_{28} .

$c = \sqrt{a}$, rounded. Both numbers to 30 b.p.

Short stores used	13	14	15
Contents at entry	a	Link	-
Contents at exit	-	c	-

T 01/2. Sine and cosine. Enter in 6_{29} for $\sin x$, in 6_0 for $\cos x$.

$c = \sin x^\circ$ or $\cos x^\circ$, given $x/90$. Numbers to 30 b.p.

Short stores used	13	14	15	16	21_2	21_3
Contents at entry	$x/90$	-	Link	-	-	-
Contents at exit	c	-	-	-	0	-

(See III.3, Trigonometrical functions, for the reason for giving $x/90$)

Using the subroutines, make up master routines for some of the following exercises. (The solutions given in V.2 have had storage locations allotted as these exercises are used again in exercises G.)

1. Read 2 numbers x and y on successive cards. Calculate xy and punch it. [9]
2. Read a sequence of 10 numbers. Calculate the sum of their squares and punch it. [The constant 10 will be stored with the master routine.] [17]
3. Read a sequence of 10 pairs of numbers (x,y). Calculate xy for each pair, form Exy and punch it when 10 pairs have been read. [The constant 10 will be stored with the master routine.] [19]
4. Read a number x. Calculate $x^{3/2}$ and $x^{5/4}$ and punch them on 2 cards. [15]
5. Read a number x. Calculate $\sqrt{x/x+1}$ and punch it. [x will be a fraction with 30 b.p. so the fraction 1.0 with 30 b.p. must be added to it to form the denominator of the fraction.] [11]
6. Read a number ϵ . Find the smallest integer n for which $\sqrt{n/n+1} - \sqrt{n-1/n} < \epsilon$. [In calculating $\sqrt{n/n+1}$, n must be an integer so source 27 can be used for the 1 in the denominator. When n has been found, multiply it by $2^{30} \cdot 10^{-9}$ and add P₁ to the result, then punch this fraction.] [27]

7. Read a sequence of numbers. Accumulate in 19_2 the sum of the positive numbers: accumulate in 19_3 the sum of the negative numbers. When a card punched with the number 0 is read, stop reading and punch these totals. [14]

8. Read the numbers x and y and place them in 19_2 and 19_3 respectively. Replace them by $\frac{1}{2}(x + y)$ and $\sqrt{xy}$. Punch these two numbers. [20]

9. Read a constant ϵ . Then read two numbers x and y into 19_2 and 19_3 respectively. Replace them by $\frac{1}{2}(x + y)$ and $\sqrt{xy}$. Continue this replacement of the numbers in 19_2 and 19_3 until

$$C(19_2) - C(19_3) \leq \epsilon.$$

Then punch both numbers and go back to the stage where another pair of numbers (x, y) is read. [Assume that ϵ, x, y all lie between 0 and 1.] [26]

10. Read a pair of numbers (r, θ) , calculate and punch $\cos 100\theta^\circ$ and $r \sin 100\theta^\circ$. [23]

11. Patients arrive at a doctor's surgery at 5 minute intervals. The doctor attends to the first immediately. The length of time he takes over each patient is punched in minutes divided by 100 on a succession of cards and the pack of cards ends with a negative number. Compute the time for which each patient must wait and punch the greatest of these times. [17]

[Warning: What happens if the doctor has to wait for a patient?]

Exercises F

Prepare programs for the following examples and allot the storage locations. It does not matter if the program is rather slow. Do not use spill-over techniques for counting.

1. Given a set of numbers in $10_0, 10_1, 10_2 \dots$ and given rP_{17} in 19_2 ($r < 31$) place the content of 10_r in 16 and add its modulus to the number in 19_3 . [9]

2. Using the previous exercise, find the sum of the moduli of the numbers in $10_0, 10_1, \dots, 10_{19}$. [16]

3. Find the sum of the squares of the 16 numbers in $9_{16}, 9_{17}, \dots, 9_{31}$. [15]

4. What would happen in your program for exercise 3 if instead of 16 numbers there were (i) 32 numbers?
(ii) 33 numbers?

5. Replace the 16 numbers in $9_{16}, 9_{17}, \dots, 9_{31}$ by their squares. [15]

6. Test the positive numbers in $11_0, 11_1, \dots$ until a number x satisfying $x > 1$ is found. Place x in 13. [Assume that one of the numbers in DL 11 satisfies the condition.] [11]

7. The numbers in $11_0, 11_1, \dots, 11_{31}$ are positive. If any of them is greater than or equal to 1, halve all the numbers. [23]

8. $y = a_0 + a_1 x + a_2 x^2 + \dots + a_n x^n$, where $n \leq 31$. Given $a_0, a_1, \dots, a_n$ in $A_p, A_{p+1}, \dots, A_{p+n}$ and x in 16, nP_{17} in 20_2 and $(AP_5 + pP_{17})$ in 20_3 , make a subroutine to place y in 14 without using the sum series subroutine in A 12/1. What is the order of the subroutine? [12]

9. $x_0, x_1, \dots, x_{n-1}$ are stored in $A_p, A_{p+1}, \dots, A_{p+n-1}$ and $y_0, y_1, \dots, y_{n-1}$ are stored in $B_q, B_{q+1}, \dots, B_{q+n-1}$. Given $AP_5 + pP_{17}$ in 19_2 , $BP_5 + qP_{17}$ in 19_3 and nP_{17} in 20_2 ($n \leq 32$), place $x_0 y_0 + x_1 y_1 + \dots + x_{n-1} y_{n-1}$ in 20_3 . [This is the scalar product of the n -length vectors $\underline{x}, \underline{y}$.]

Exercises G

Allot storage to the examples of Exercises E in such a way that the programs are optimum coded.

V.2 Solutions to the previous exercises

Exercises A

1.	16 - 25	$C'(13) = w + z$
2.	14 - 13	$C'(13) = x$
	15 - 26	$C'(13) = x - y$
3.	13 - 25	$C'(13) = 2w$
	14 - 25	$C'(13) = 2w + x$
4.	24 - 25	$C'(13) = w + 2x$
5.	15 - 13	$C'(13) = y$
	13 - 25	$C'(13) = 2y$
	16 - 26	$C'(13) = 2y - z$

It is better not to use TS 14 unless this will save instructions.

6.	24 - 14 (2 m.c.)	$C'(14) = 2^2 x$
7.	13 - 25 (4 m.c.)	$C'(13) = 2^4 w$
8.	23 - 14 (3 m.c.)	$C'(14) = 2^{-3} x$
9.	13 - 14	$C'(14) = w$
	23 - 13	$C'(13) = 2^{-1} w$

In this case the use of TS 14 is unavoidable.

10.	$24 - 13$	$C'(13) = 2x$
	$24 - 14$ (2 m.c.)	$C'(14) = 4x$
	$24 - 25$	$C'(13) = 2x + 8x = 10x$

The alternative of

$24 - 13$
$24 - 14$ (3 m.c.)
$14 - 25$

is just one minor cycle slower.

11.	$19_2 - 13$	$C'(13) = p$
	$13 - 25$	$C'(13) = 2p$
	$19_3 - 14$	$C'(14) = q$
	$14 - 26$	$C'(13) = 2p - q$
	$24 - 26$ (2 m.c.)	$C'(13) = (2p - q) - 2 \cdot (2q) = 2p - 5q$

The second instruction is slightly better than $19_2 - 25$ (they have the same numerical effect) because it can be obeyed in an even or odd minor cycle. Similarly the fourth instruction is preferred to $19_3 - 26$.

12.	$15 - 16$	$C'(16) = y, \quad C(14) = x$
-----	-----------	-------------------------------

MULT

$21_3 - 15$	$C'(15) = xy$
-------------	---------------

13.	$13 - 19_2$	$C'(19_2) = w$
	$16 - 19_3$	$C'(19_3) = z$
	$15 - 16$	$C'(16) = y, \quad C(14) = x$
	MULT	$C'(21_3) = xy$
	$19_2 - 14$	$C'(14) = w$
	$19_3 - 16$	$C'(16) = z$
	$21_3 - 19_3$	$C'(19_3) = xy$
	MULT	$C'(21_3) = wz$
	$19_3 - 13$	$C'(13) = xy$
	$21_3 - 26$	$C'(13) = xy - wz$

14.	$19_2 - 16$	$C'(16) = p$
	$19_3 - 14$	$C'(14) = q$
	MULT	$C'(21_3) = pq$
(a)	$16 - 13$	$C'(13) = p$
(a)	$14 - 26$	$C'(13) = p - q$
	$13 - 16$	$C'(16) = p - q$
(b)	$24 - 25$	$C'(13) = p + q$
(b)	$21_3 - 14$	$C'(14) = pq$
(b)	$24 - 20_3$	$C'(20_3) = 2pq$
(b)	$13 - 14$	$C'(14) = p + q$
	MULT	$C'(21_3) = (p - q)(p + q)$
(c)	$21_3 - 13$	$C'(13) = p^2 - q^2$
(c)	$13 - 20_2$	$C'(20_2) = p^2 - q^2$

- NOTES:
- (a) Where a number is in 2 stores of different lengths (e.g. p in TS 16 and DS 19, even) we again use the shorter.
 - (b) If any of these four instructions (which all use TS 14) got out of sequence the result would be wrong. The obvious sequence here is

$24 - 25$	$C'(13) = p + q$
$13 - 14$	$C'(14) = p + q$
$21_3 - 13$	$C'(13) = pq$
$13 - 25$	$C'(13) = 2pq$
$13 - 20_3$	$C'(20_3) = 2pq$

but this needs one more instruction.

- (c) The single instruction $21_3 - 20_2$ is impossible: see I.4, example 9.
- (d) $(p + qi)^2 = (p^2 - q^2) + 2pq i$.

Inside DEUCE a complex number like $(p + qi)$ will be stored as the pair of numbers (p, q) . This program would be used to square the complex number in DS 19.

$$15. \quad az^2 + bz + c = (a.z + b).z + c$$

$$17_0 - 14$$

$$C'(14) = a$$

MULT

$$21_3 - 13$$

$$C'(13) = az$$

$$17_1 - 25$$

$$13 - 14$$

$$C'(14) = az + b$$

MULT

$$21_3 - 13$$

$$C'(13) = az^2 + bz$$

$$17_2 - 25$$

$$13 - 14$$

$$C'(14) = az^2 + bz + c$$

This method is always used for evaluating polynomials.

$$16. \quad 15 - 13$$

$$C'(13) = 2r - 1$$

$$27 - 25 \quad (2 \text{ m.o.})$$

$$C'(13) = (2r - 1) + 2$$

$$13 - 15$$

$$C'(15) = 2r + 1$$

$$16 - 25$$

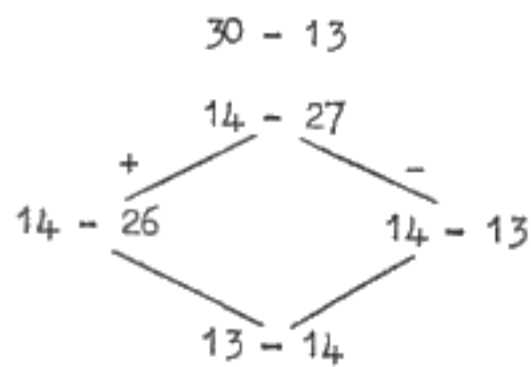
$$C'(13) = (2r + 1) + r^2$$

$$13 - 16$$

$$C'(16) = (r + 1)^2$$

Exercises B

1.

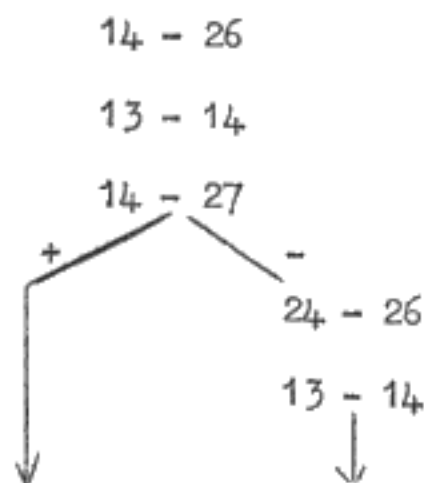


Test x

$$\begin{cases} \text{Put } -x = -|x| \text{ in 13 if } x \geq 0 \\ \text{Put } +x = -|x| \text{ in 13 if } x < 0 \end{cases}$$

$$-|x|$$

2.



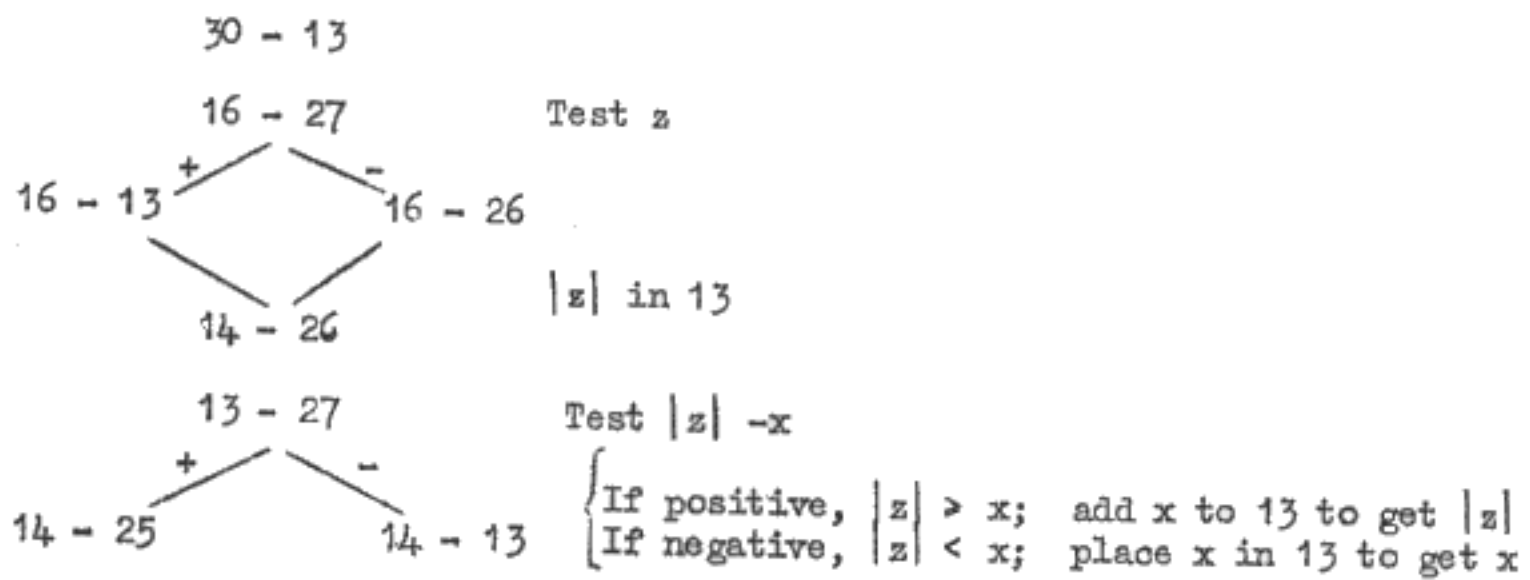
$w - x$

Test $w - x$

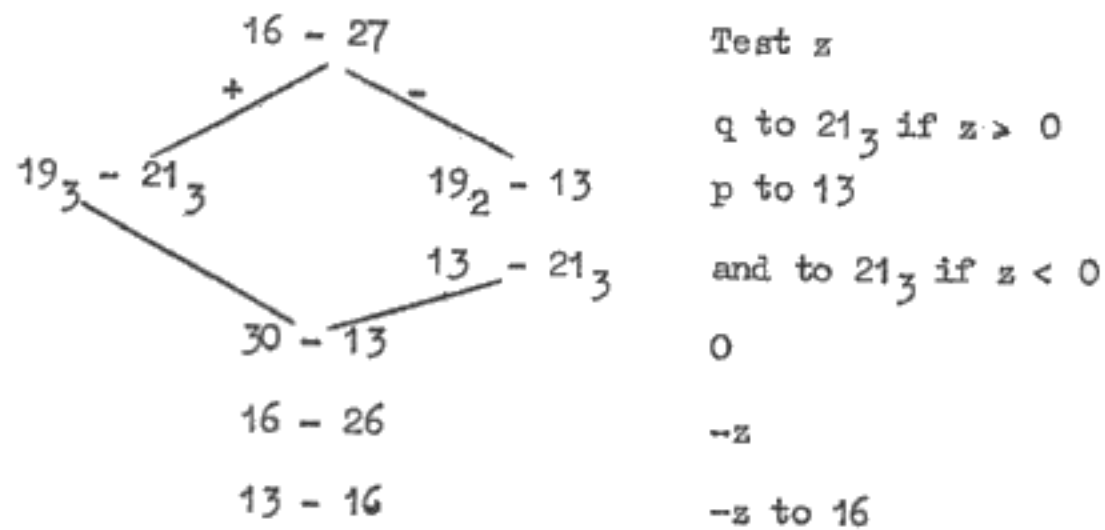
$$\begin{cases} \text{If positive, } C'(14) = |w-x| \\ \text{If negative, } C'(13) = (w-x) - 2(w-x) \end{cases}$$

$$C'(14) = -(w-x) = |w-x|$$

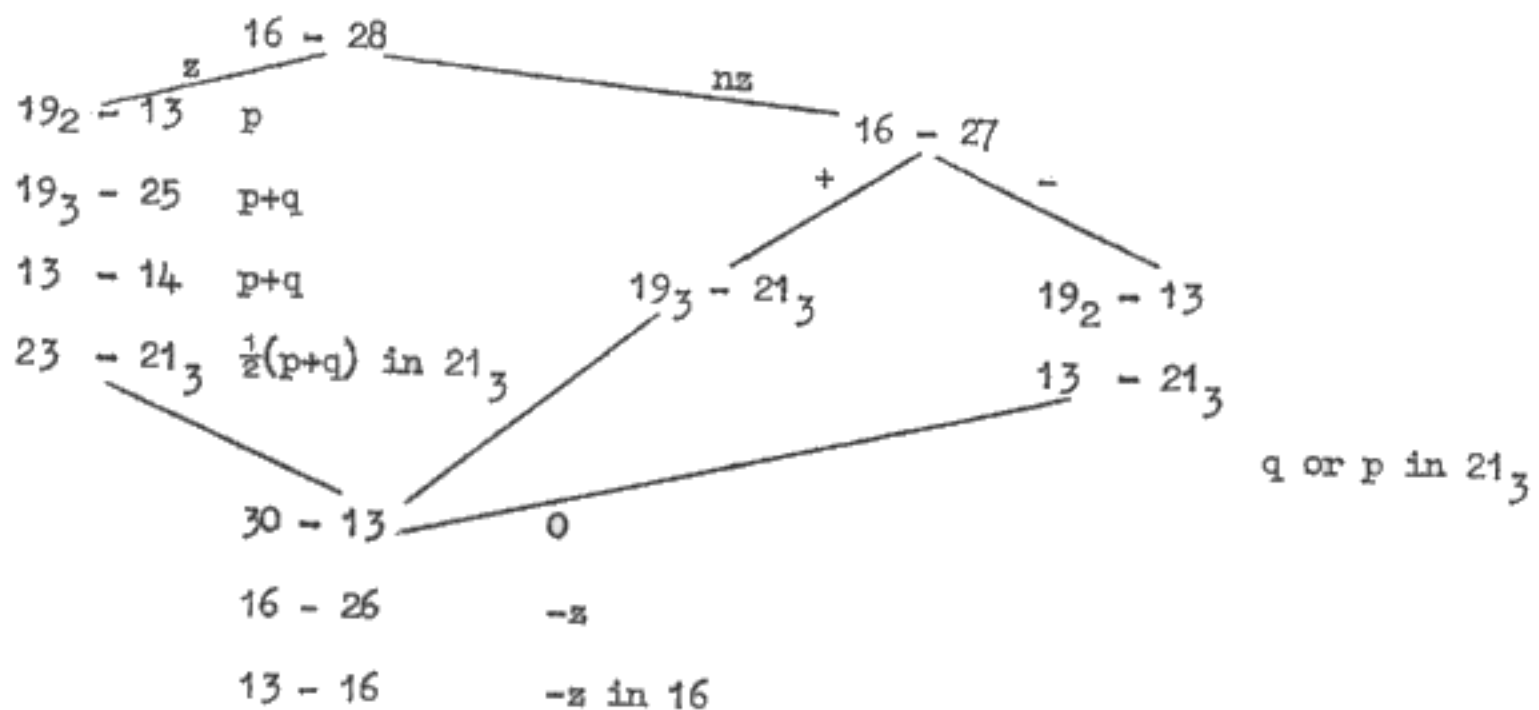
3.



4.



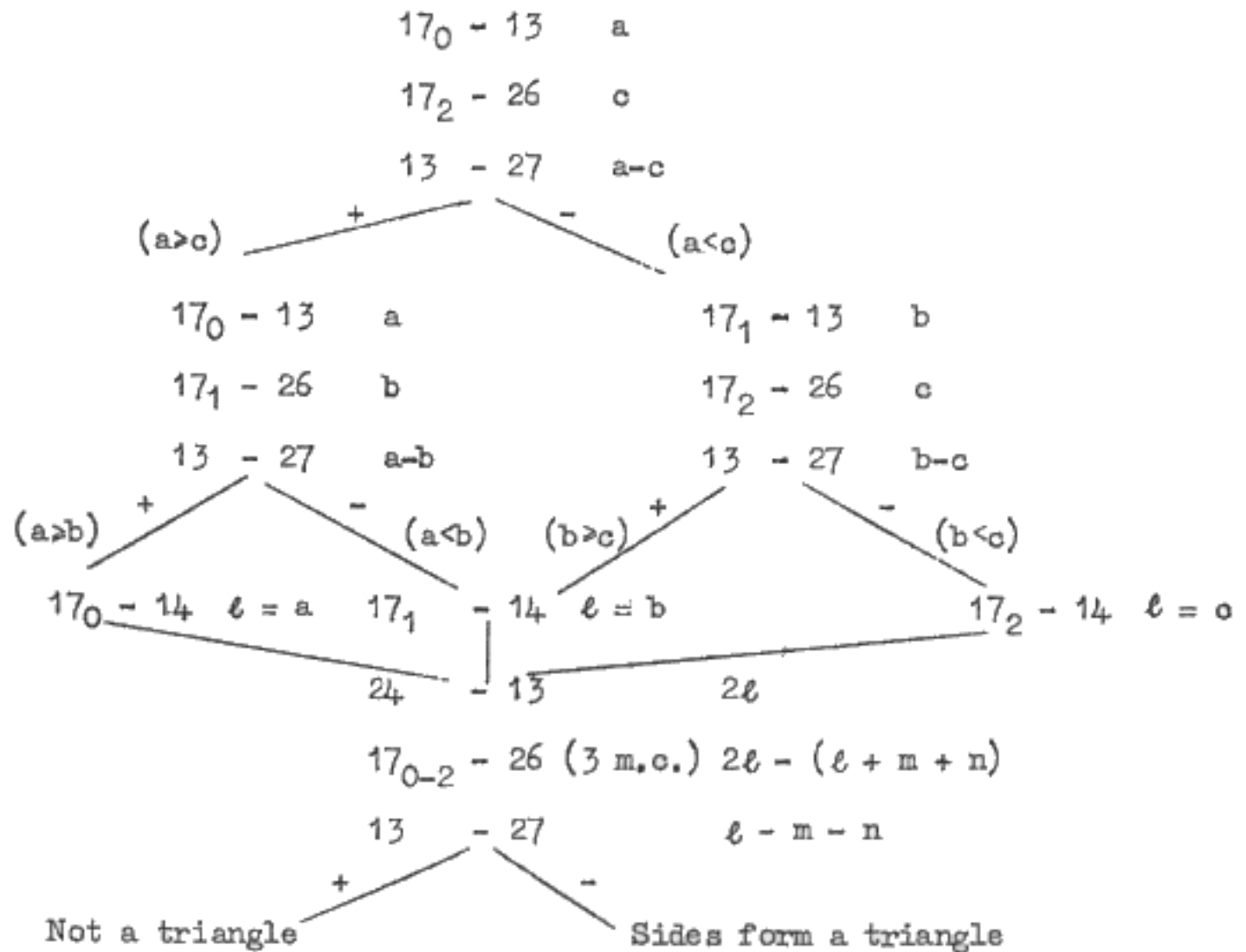
5.



6. Let 20₂ contain the sum of 3/8 and 5/8, i.e. 1. (This is the fraction 1.0 with 30 b.p., not the integer 1 given by source 27.)

$$\begin{array}{ll}
 19_2 - 13 & 1 \\
 18_0 - 26 & 1 - 3/8 = 5/8 \text{ or } 1 - 5/8 = 3/8 \\
 13 - 18_0 & 5/8 \text{ or } 3/8.
 \end{array}$$

7. If the three sides of the triangle are ℓ , m , n , the program must test $\ell - m - n$, $m - n - \ell$, $n - \ell - m$ to see that all three are negative. If we know that ℓ is the longest, then $\ell - m - n < 0$ is a sufficient criterion: the other two expressions are less than $\ell - m - n$. It is more convenient to compute $2\ell - (\ell + m + n)$.



NOTES:

1. The instruction $17_1 - 14$ is on the on side of one discrimination and on the off side of another. This is permissible: the instructions $17_0 - 14$, $17_1 - 14$, $17_2 - 14$ are in three consecutive minor cycles of the same delay line. It would not be permissible to have the $17_1 - 14$ on the same side of the discriminations, e.g. the on side if $c - b$ had been tested instead of $b - c$, for in this case the $17_0 - 14$ and $17_2 - 14$ instructions would both have to be in the preceding minor cycle.

2. By testing $(\ell - m - n)$ we say that $\ell = m + n$ is not a triangle. (This is the case of a triangle with area 0.) If we tested $(m + n - \ell)$ the case $\ell = m + n$ would count as a triangle.

Exercise C

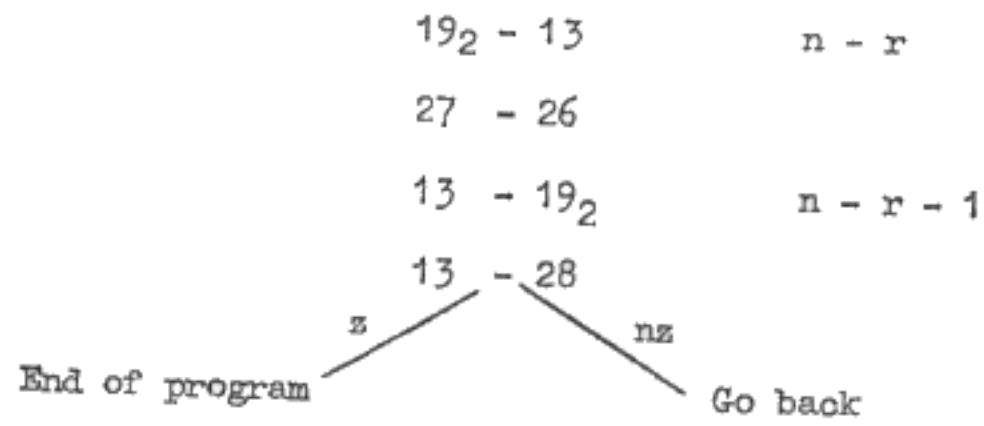
Hint (ii). To evaluate $S_{n-r-1} = x S_{n-r} + a$.

Keep S_{n-r} in 14 and leave x , a permanently in 16, 19_3 .

```

    MULT      x x Sn-r
    213 - 13  x Sn-r
    193 - 25  x Sn-r + a
    13 - 14   Sn-r-1
    
```

Hint (iii). Keep $(n-r)$ in 19_2 .

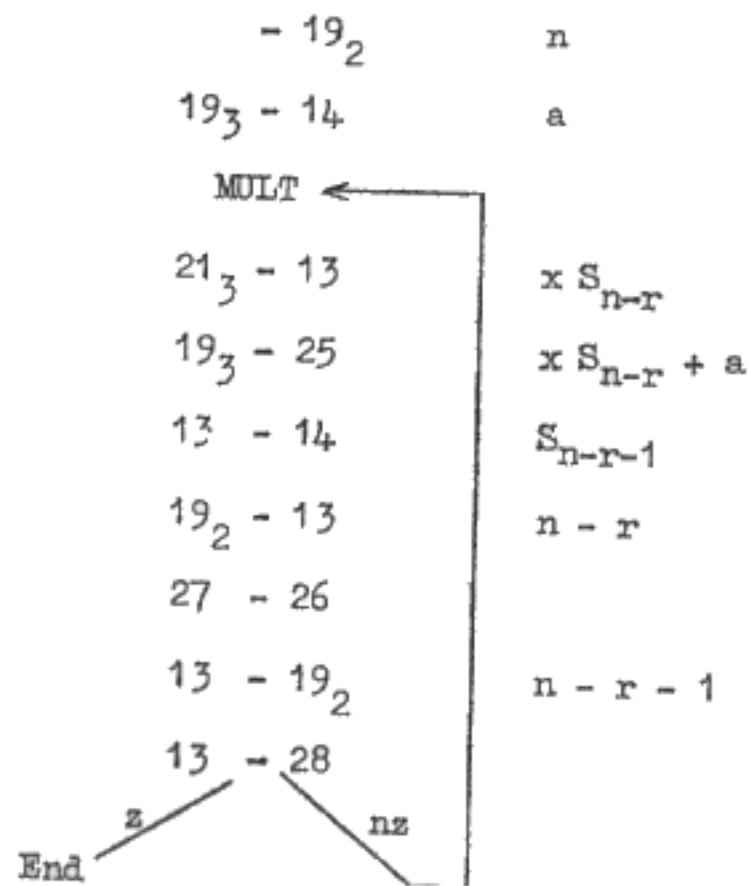


Hint (iv). The initial conditions are given by $r = 0$.

Then $n - r = n$, $S_n = a$.

$$\begin{array}{ll} - 19_2 & n \\ 19_3 - 14 & a \end{array}$$

So the complete program is



Exercises D

1.	<u>m.c.</u>	<u>N</u>	<u>S - D</u>	<u>Ch</u>	<u>W</u>	<u>T</u>
	0	1 ₈ ,	12 - 21,	2,	2,	6
	1	1 ₁₂ ,	20 - 22,	2,	1,	9
	2	1 ₂₃ ,	24 - 14,	1,	15,	19
	3	1 ₉ ,	18 - 6,	0,	3,	4
	4	1 ₁₀ ,	18 - 17,	1,	1,	4
	5	1 ₁₃ ,	20 - 13,	0,	0,	6
	6	1 ₁₄ ,	17 - 20,	0,	2,	6
	7	1 ₁₁ ,	12 - 14,	0,	17,	2
	8	1 ₂₆ ,	17 - 11,	2,	4,	6
	9	1 ₂₀ ,	7 - 21,	0,	4,	9
	10	1 ₂ ,	23 - 14,	1,	2,	22
	11	1 ₂₁ ,	11 - 15,	0,	19,	8
	12	1 ₃₁ ,	5 - 7,	2,	17,	17
	13	1 ₁₈ ,	13 - 21,	0,	1,	3
	14	1 ₁₅ ,	13 - 8,	0,	0,	31
	15	1 ₂₁ ,	18 - 27,	0,	1,	4
	16	1 ₃₀ ,	15 - 11,	0,	12,	12
	17	1 ₁₉ ,	2 - 13,	0,	0,	0
	18	1 ₂₅ ,	17 - 14,	0,	3,	5
	19	1 ₂₃ ,	4 - 25,	0,	0,	2
	20	1 ₅ ,	19 - 5,	0,	10,	15
	21	1 ₁₆ ,	14 - 12,	0,	24,	25
	22	1 ₀ ,	14 - 12,	0,	24,	8
	23	1 ₂₈ ,	12 - 26,	0,	1,	3
	24	1 ₃ ,	3 - 17,	0,	3,	9
	25	1 ₂₉ ,	14 - 17,	0,	1,	2
	26	1 ₁ ,	19 - 18,	2,	2,	5
	27	1 ₄ ,	12 - 18,	1,	4,	7
	28	1 ₇ ,	13 - 5,	0,	1,	9
	29	1 ₆ ,	19 - 17,	0,	2,	7
	30	1 ₂₄ ,	1 - 5,	0,	8,	24
	31	1 ₂₇ ,	6 - 17,	1,	24,	26

Asterisks should come after the instructions in the following minor cycles: 28, 7 (because $W > T$), 11 ($W > T$), 21, 30, 24, 20, 29, 14, two after 22 (one because $W > T$) 26, 12 ($W = T$ with characteristic 2), 31, 27 and 10.

The inner loop from 1₂₁ to 1₁₅ and back to 1₂₁ therefore takes 6 major cycles. The outer loop has 10 further asterisks (the coding is far from the optimum), so the outer loop takes $(6N + 10)$ major cycles.

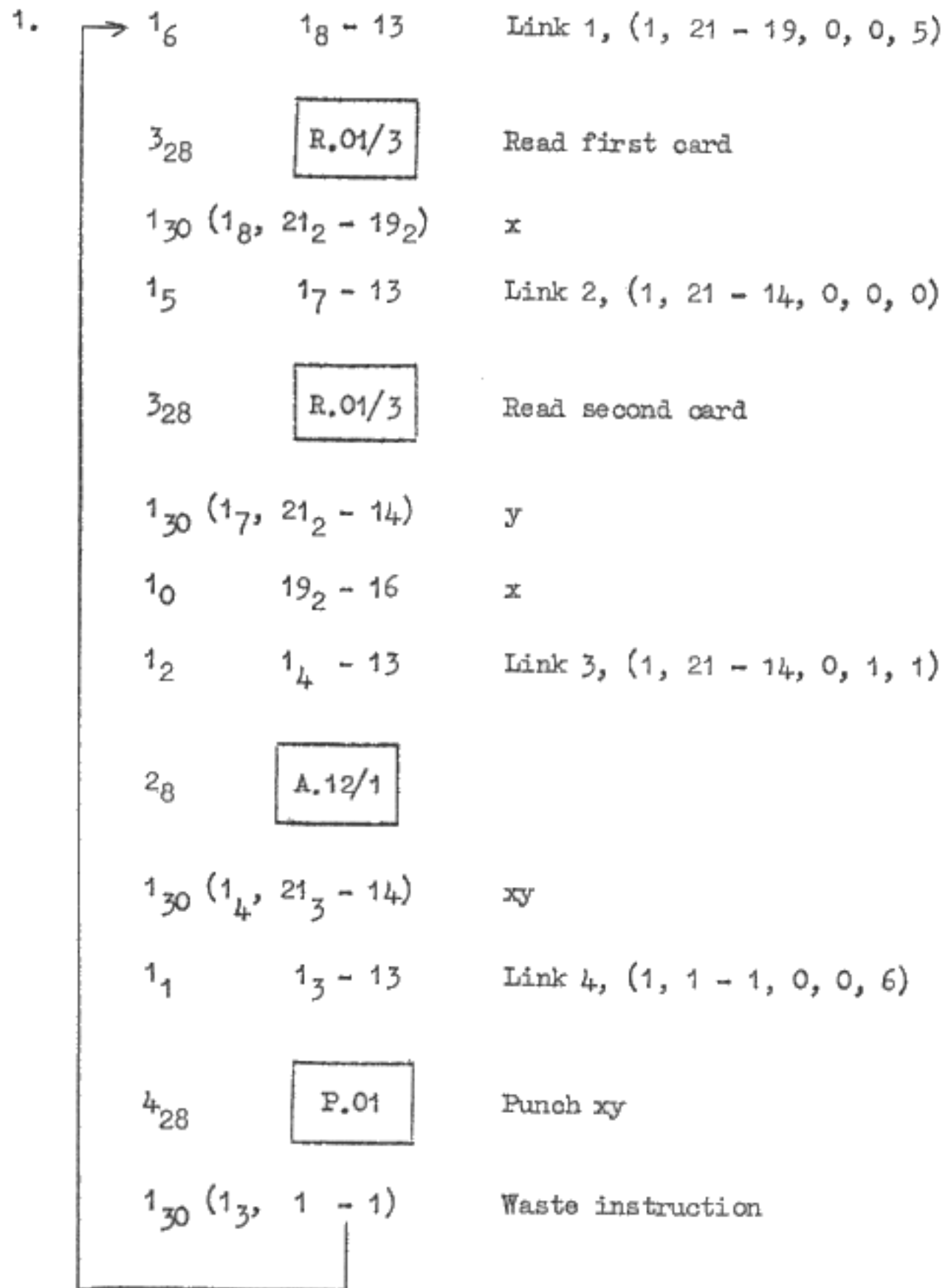
The obvious way to calculate the time round, say, the inner loop is to put an asterisk each time m.c.21 is passed. The reader should do this and verify that the same result is obtained. The two reasons for choosing m.c.0 rather than the first m.c. of the loop are:

- (i) m.c.0 is easily recognised;
- (ii) when the asterisk mark m.c.21, the sixth and last asterisk comes on the return from 1₁₅ to 1₂₁. This is easily forgotten.

2.	<u>m.c.</u>	<u>N</u>	<u>S - D</u>	<u>Ch</u>	<u>W</u>	<u>T</u>
	0	1 ₃ ,	17 - 26,	0,	0,	1
	1		-			
	2		-			
	3	1 ₅ ,	13 - 27,	0,	0,	0
	5	1 ₁₁ ,	17 - 13,	0,	1,	4
	6	1 ₁₂ ,	17 - 13,	0,	1,	4
	11	1 ₁₃ ,	17 - 26,	0,	0,	0
	12	1 ₁₄ ,	17 - 26,	0,	0,	0
	13	1 ₁₅ ,	13 - 27,	0,	0,	0
	14	1 ₁₆ ,	13 - 27,	0,	0,	0
	15	1 ₂₄ ,	17 - 14,	0,	3,	7
	16	1 ₂₄ ,	17 - 14,	0,	3,	6
	17	1 ₂₄ ,	17 - 14,	0,	3,	5
	24	1 ₂₆ ,	24 - 13,	0,	0,	0
	26	1 ₃₀ ,	17 - 26,	1,	0,	2
	29	1 ₀ ,	17 - 13,	0,	1,	1
	30	1 ₁ ,	13 - 27,	0,	0,	1

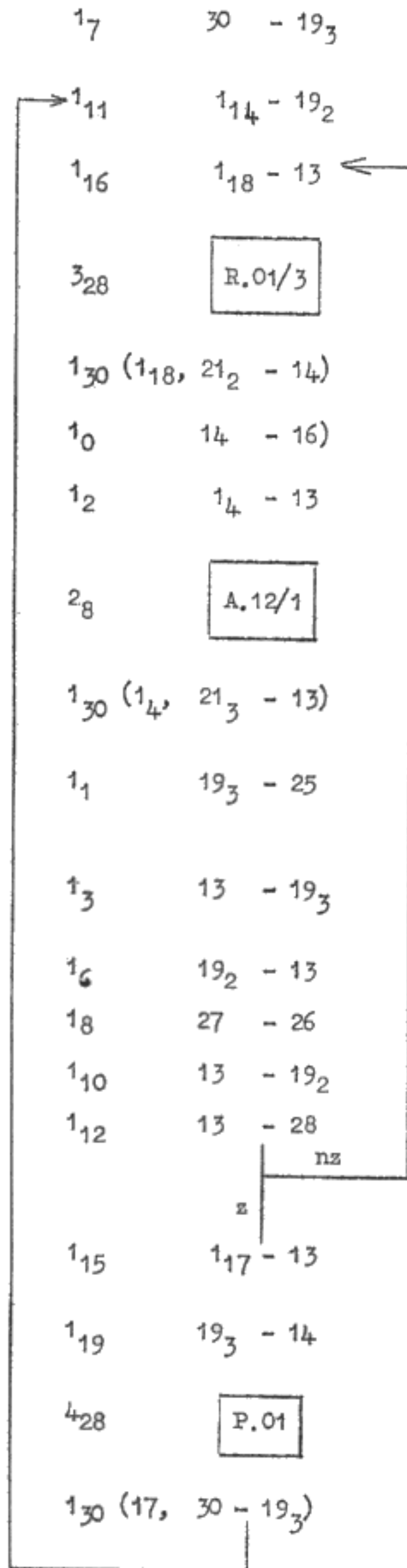
Time: 1 major cycle, 5 minor cycles in each case.

Exercises E



Many of these simple examples end with a link which does nothing except lead back to the beginning. In this one the final link can either be a waste instruction or it can do the work of the first instruction and lead to the second: this latter method is used in the exercise following.

2.



Set $\sum_n^{10} x_r^2 = 0$ } Set n = 10
 10P₁

Link 1, (1, 21 - 14, 0, 0, 0)

Reqd x_n

x_n

Link 2, (1, 21 - 13, 0, 1, 1)

x_n^2

Add $\sum_{n+1}^{10} x_r^2$

$\sum_n^{10} x_r^2$

nP₁

(n - 1)P₁

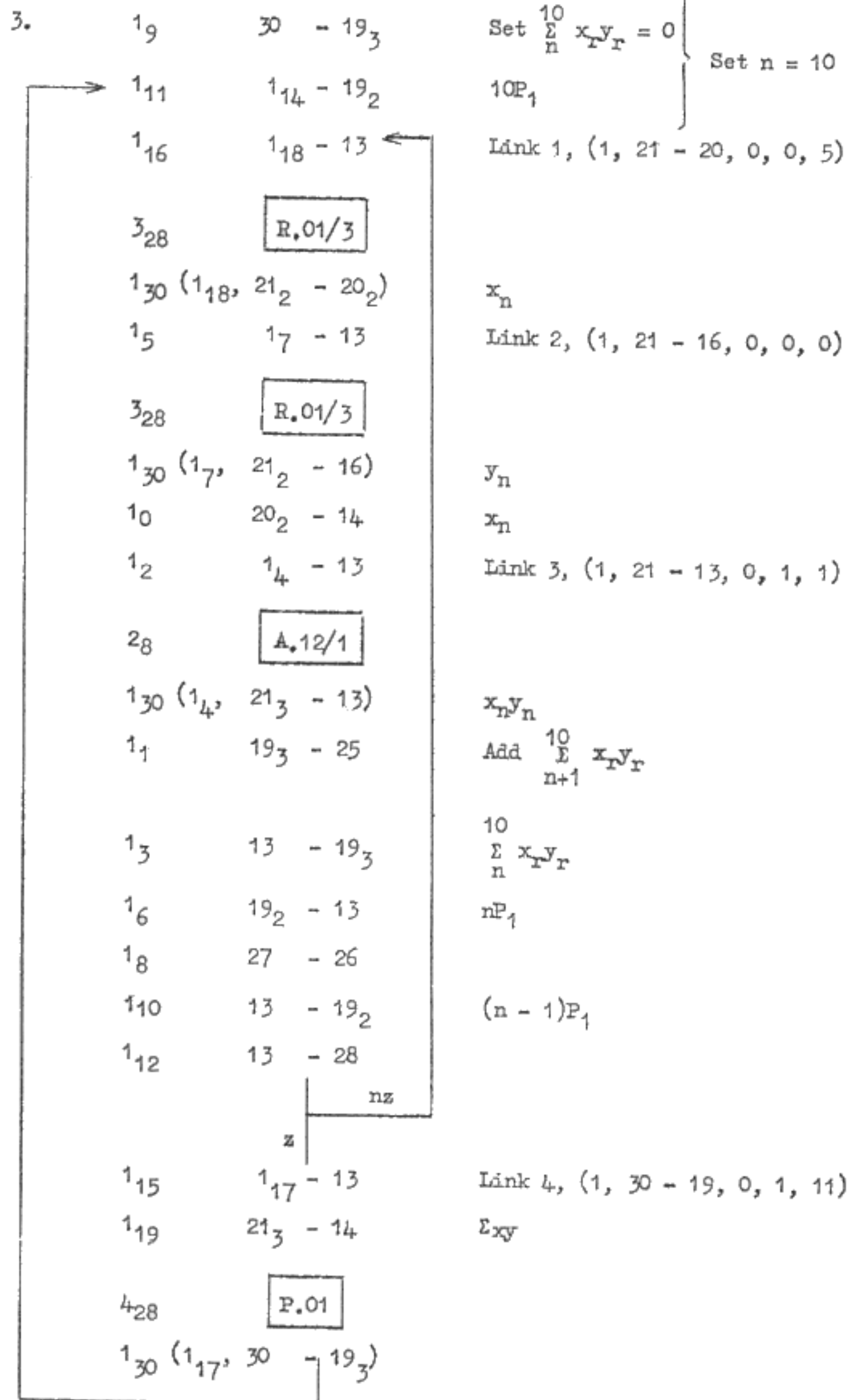
On if n = 1

Link 3, (1, 30 - 19, 0, 1, 11)

$\sum_1^{10} x_r^2$

Punch $\sum x^2$

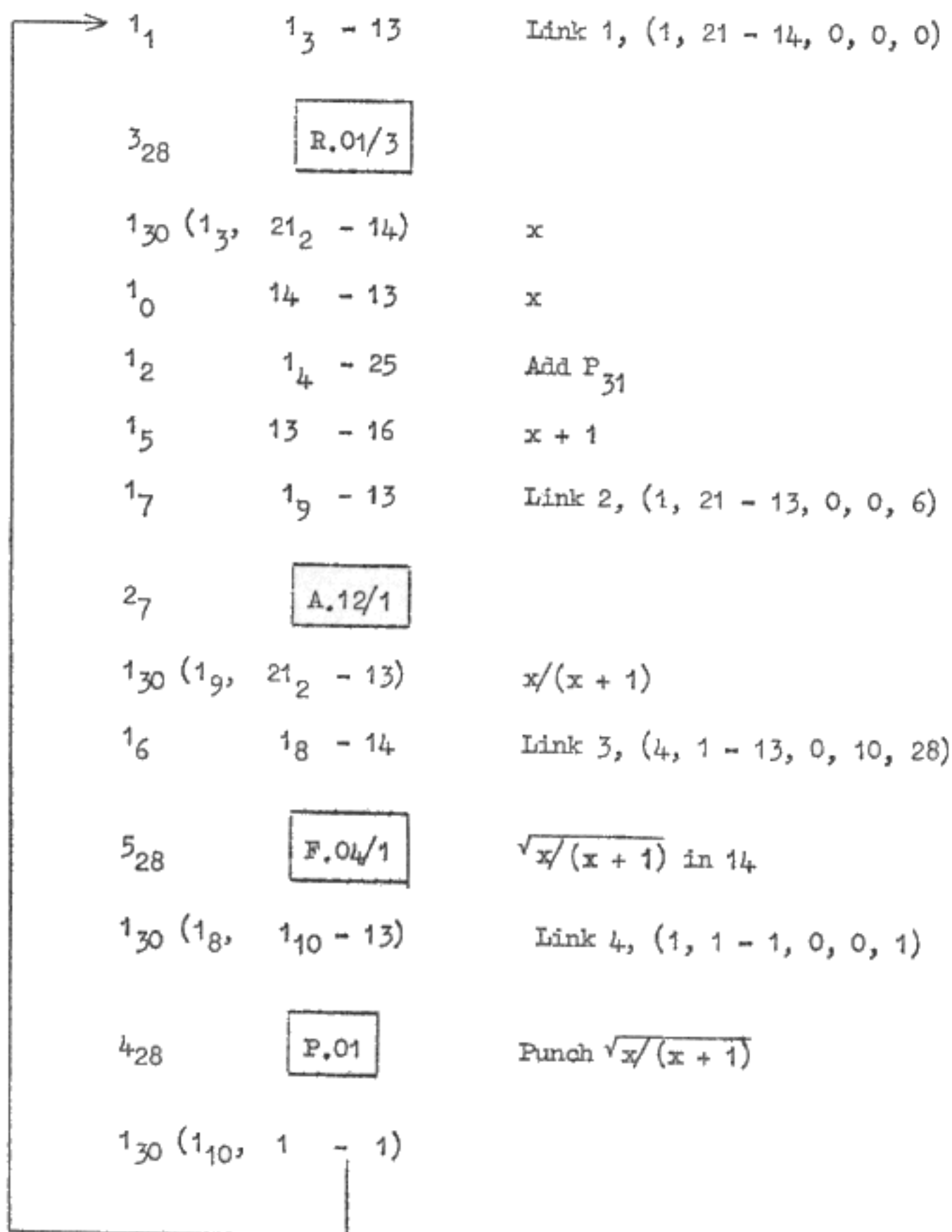
Because the multiplication subroutine in A.12/1 leaves in 21_2 the less significant half of the product ab with P_{32} added, it increases the accuracy of the calculation if Σx^2 is summed double-length and rounded off to single length only when it is to be punched. This will be done when the double-length accumulator has been explained in I.10.



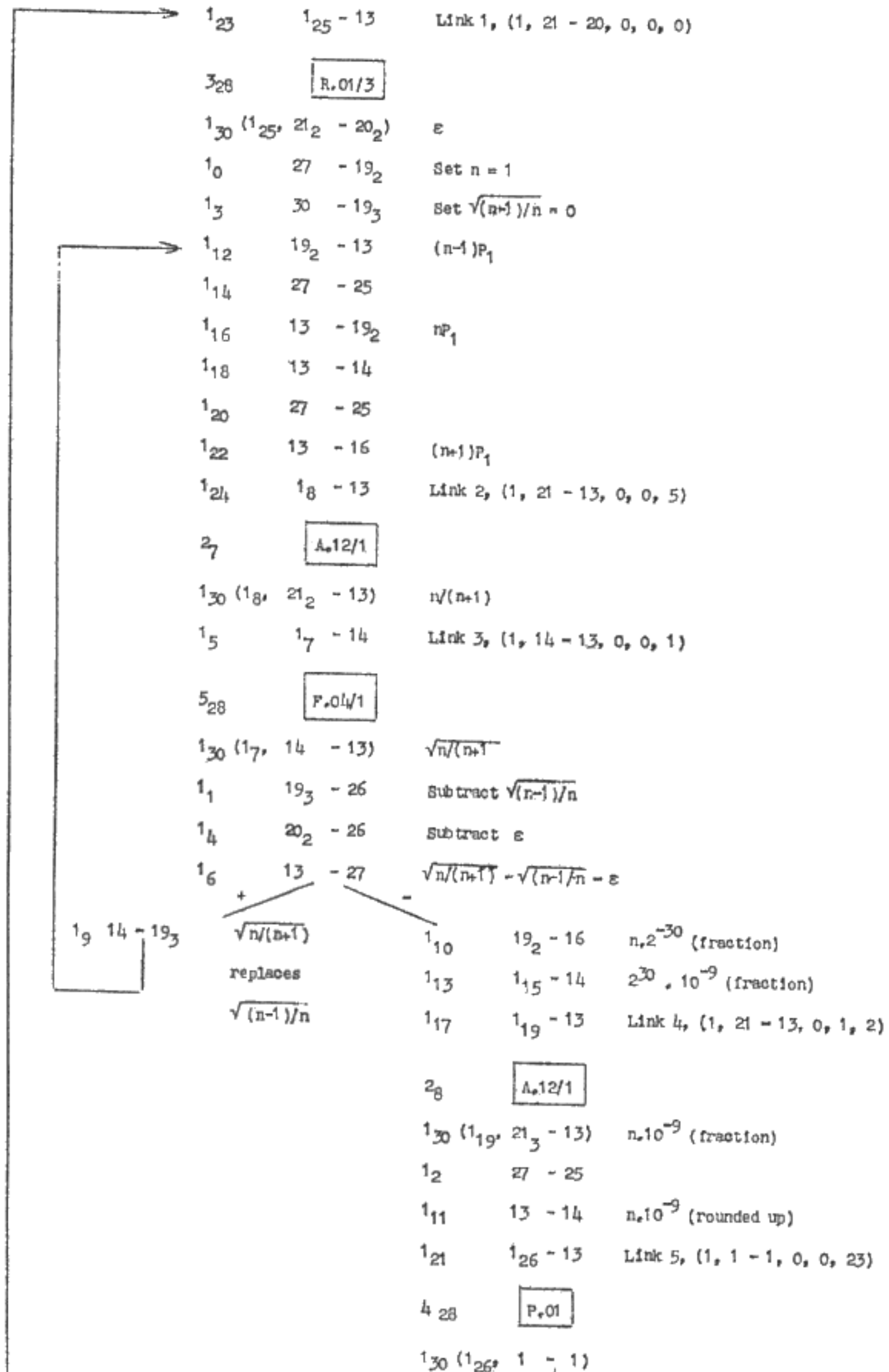
4.

1_1	$1_3 - 13$	Link 1, (1, 21 - 16, 0, 0, 0)
3_{28}	R.01/3	
$1_{30} (1_3, 21_2 - 16)$		x
1_0	$16 - 13$	x
1_2	$1_4 - 14$	Link 2, (2, 1 - 13, 0, 5, 8)
5_{28}	F.04/1	$x^{1/2}$ in 14, x in 16
$1_{30} (1_4, 1_5 - 13)$		Link 3, (1, 21 - 19, 0, 1, 8)
2_8	A.12/1	
$1_{30} (1_5, 21_3 - 19_3)$		$x^{3/2}$
1_8	$1_4 - 13$	$x^{1/2}$
1_{10}	$1_{13} - 14$	Link 4, (2, 1 - 13, 0, 6, 8)
5_{28}	F.04/1	$x^{1/4}$ in 14, x in 16
$1_{30} (1_{13}, 1_6 - 13)$		Link 5, (1, 19 - 14, 0, 1, 7)
2_8	A.12/1	
$1_{30} (1_6, 19_3 - 14)$		$x^{3/2}$
1_7	$21_3 - 19_3$	$x^{5/4}$
1_9	$1_{11} - 13$	Link 6, (1, 19 - 14, 0, 1, 12)
4_{28}	P.01	Punch $x^{3/2}$
$1_{30} (1_{11}, 19_3 - 14)$		$x^{5/4}$
1_{12}	$1_{14} - 13$	Link 7, (1, 1 - 1, 0, 0, 1)
4_{28}	P.01	Punch $x^{5/4}$
$1_{30} (1_{14}, 1 - 1)$		

5.

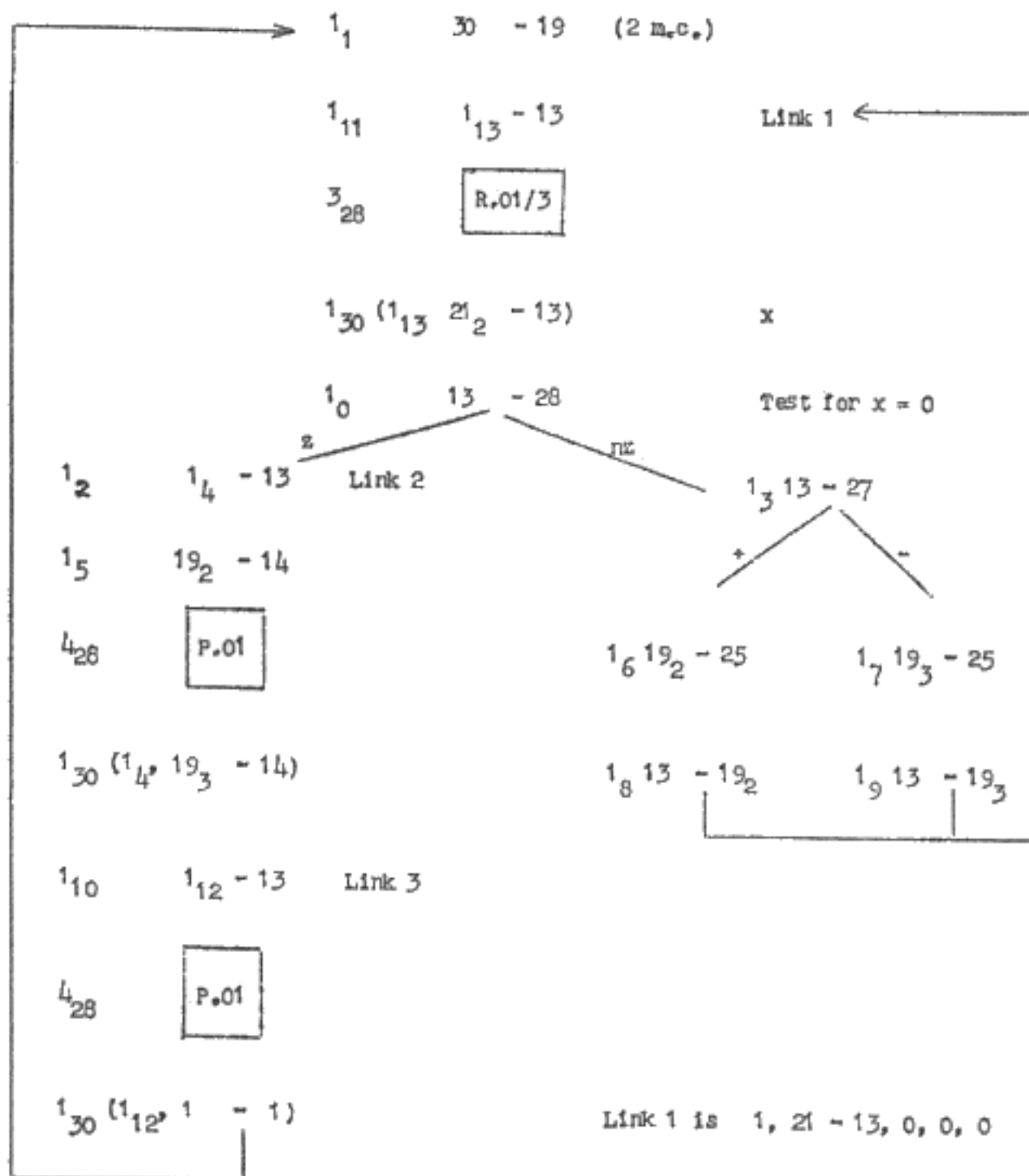


6.



This is a clumsy method of punching an integer. Usually the special subroutine for punching integers recommended in III.3 would be incorporated in the program pack and used.

7.



Link 1 is 1, 21 - 13, 0, 0, 0

Link 2 is 1, 19 - 14, 0, 1, 10

Link 3 is 1, 1 - 1, 0, 0, 1

8.

1_1	$1_3 - 13$	Link 1, (1, 21 - 19, 0, 0, 2
3_{28}	R.01/3	
$1_{30} (1_3, 21_2 - 19_2)$		x
1_2	$1_5 - 13$	Link 2, (1, 21 - 15, 0, 0, 0)
3_{28}	R.01/3	
$1_{30} (1_5, 21_2 - 15)$		y
1_0	$15 - 19_3$	
1_4	$19_2 - 13$	x
1_7	$19_3 - 25$	
1_9	$13 - 14$	x + y
1_{12}	$19_2 - 16$	x
1_{14}	$23 - 19_2$	$\frac{1}{2}(x + y)$
1_{17}	$19_3 - 14$	y
1_{19}	$1_6 - 13$	Link 3, (1, 21 - 13, 0, 1, 8)
2_8	A.12/1	
$1_{30} (1_6, 21_3 - 13)$		xy
1_8	$1_{10} - 14$	Link 4, (1, 14 - 19, 0, 1, 13)
5_{28}	F.04/1	
$1_{30} (1_{10}, 14 - 19_3)$		$\sqrt{xy}$
1_{13}	$19_2 - 14$	$\frac{1}{2}(x + y)$
1_{16}	$1_{18} - 13$	Link 5, (1, 19 - 14, 0, 1, 11)
4_{28}	P.01	
$1_{30} (1_{18}, 19_3 - 14)$		$\sqrt{xy}$
1_{11}	$1_{15} - 13$	Link 6, (3, 1 - 13, 0, 3, 28)
4_{28}	P.01	
$1_{30} (1_{15}, 1_3 - 13)$		Punch $\sqrt{xy}$
		Link 1

9.	1_1	$1_3 - 13$	Link 1, (1, 21 - 20, 0, 0, 5)
	3_{28}	R.01/3	
	$1_{30} (1_3, 21_2 - 20_2)$	ϵ	
	1_5	$1_9 - 13$	Link 2, (1, 21 - 19, 0, 0, 13)
	3_{28}	R.01/3	
	$1_{30} (1_9, 21_2 - 19_2)$	x	
	1_{13}	$1_{17} - 13$	Link 3, (1, 21 - 15, 0, 0, 7)
	3_{28}	R.01/3	
	$1_{30} (1_{17}, 21_2 - 15)$	y	
	1_7	$15 - 19_3$	
	1_{10}	$19_2 - 14$	x_r
	1_{12}	$19_3 - 25$	Add y_r
	1_{15}	$13 - 14$	$x_r + y_r$
	1_{18}	$19_2 - 16$	x_r
	1_{20}	$23 - 19_2$	$\frac{1}{2}(x_r + y_r) = x_{r+1}$
	1_{22}	$19_3 - 14$	y_r
	1_{25}	$1_0 - 13$	Link 4, (1, 21 - 13, 0, 1, 19)
	2_8	A.12/1	
	$1_{30} (1_0, 21_3 - 13)$	$x_r y_r$	
	1_{19}	$1_{23} - 14$	Link 5, (1, 14 - 19, 0, 1, 2)
	5_{28}	F.04/1	
	$1_{30} (1_{23}, 14 - 19_3)$	$\sqrt{x_r y_r} = y_{r+1}$	
	1_2	$19_2 - 13$	x_{r+1}
	1_4	$14 - 26$	Subtract y_{r+1}
	1_6	$20_2 - 26$	Subtract ϵ
	1_8	$13 - 27$	$x_{r+1} - y_{r+1} - \epsilon$
		$+$	
	1_{11}	$19_2 - 14$	x_n
	1_{14}	$1_{16} - 13$	Link 6, (1, 19 - 14, 0, 1, 21)
	4_{28}	P.01	Punch x_n
	$1_{30} (1_{16}, 19_3 - 14)$	y_n	
	1_{21}	$1_{24} - 13$	Link 7, (3, 1 - 13, 0, 9, 28)
	4_{28}	P.01	Punch y_n
	$1_{30} (1_{24}, 1_9 - 13)$	Link 2	

$$x_{r+1} = \frac{1}{2}(x_r + y_r) = \text{arithmetic mean of } x_r \text{ and } y_r.$$

$$y_{r+1} = \sqrt{x_r y_r} = \text{geometric mean of } x_r \text{ and } y_r.$$

It is well known that the arithmetic mean is greater than the geometric mean.

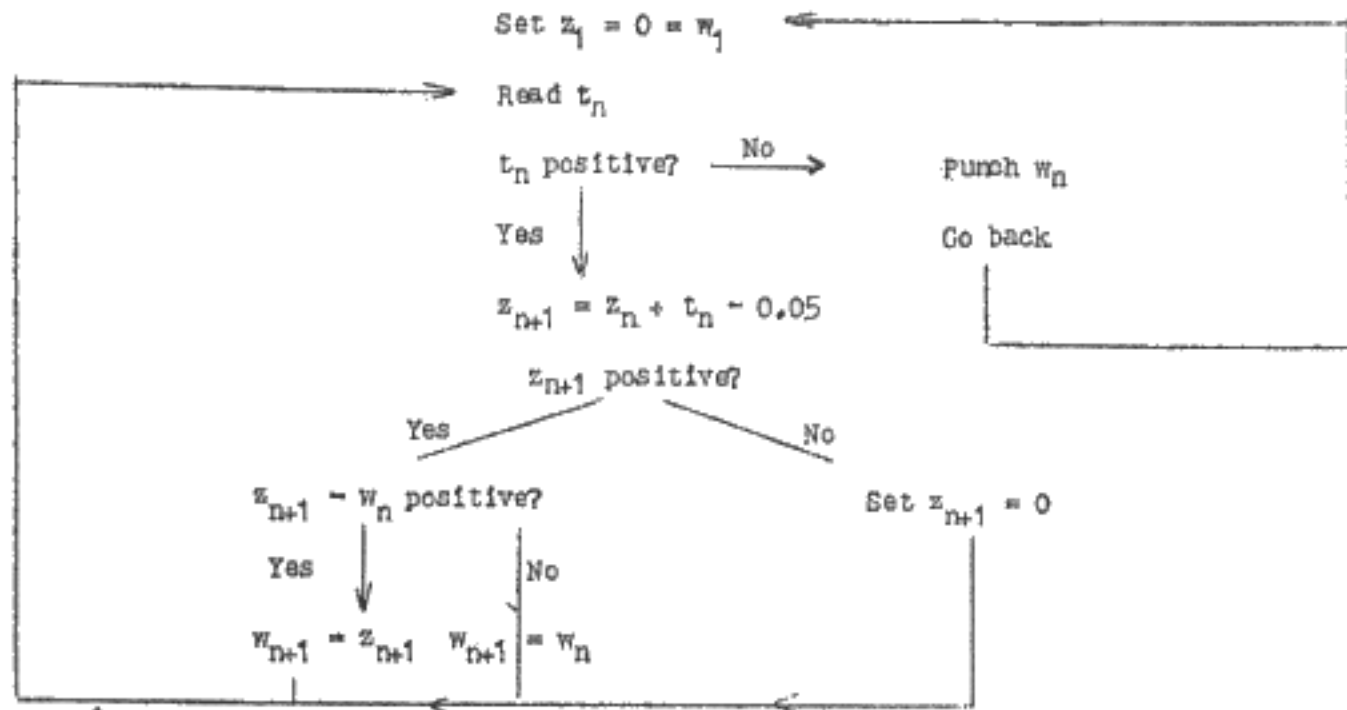
10. If T.01/2 is given y in 13 it calculates $\sin 90y^\circ$ or $\cos 90y^\circ$. We must therefore put 100/9 in 13 to get $\sin 100\theta^\circ$ and $\cos 100\theta^\circ$.

1 ₁	1 ₉ - 13	Link 1, (1, 21 - 20, 0, 0, 10)
3 ₂₈	R.01/3	
1 ₃₀ (1 ₉ , 21 ₂ - 20 ₂)		r
1 ₁₀	1 ₁₂ - 13	Link 2, (1, 21 - 16, 0, 0, 0)
3 ₂₈	R.01/3	
1 ₃₀ (1 ₁₂ , 21 ₂ - 16)		0
1 ₀	1 ₂ - 14	10/9
1 ₃	1 ₅ - 13	Link 3, (1, 21 - 20, 0, 1, 11)
2 ₈	A.12/1	
1 ₃₀ (1 ₅ , 21 ₃ - 20 ₃)		100/9
1 ₁₁	21 ₃ - 13	
1 ₁₃	1 ₁₅ - 15	Link 4, (1, 13 - 16, 0, 0, 4)
6 ₀	T.01/2	Entry for cosine
1 ₃₀ (1 ₁₅ , 13 - 16)		cos 1000
1 ₄	20 ₂ - 14	r
1 ₆	1 ₈ - 13	Link 5, (1, 21 - 14, 0, 1, 14)
2 ₈	A.12/1	
1 ₃₀ (1 ₈ , 21 ₃ - 14)		r cos 1000
1 ₁₄	1 ₁₆ - 13	Link 6, (1, 20 - 13, 0, 1, 17)
4 ₂₈	P.01	Punch r cos 1000
1 ₃₀ (1 ₁₆ , 20 ₃ - 13)		100/9
1 ₁₇	1 ₁₉ - 15	Link 7, (1, 13 - 16, 0, 0, 18)
6 ₂₉	T.01/2	Entry for sine
1 ₃₀ (1 ₁₉ , 13 - 16)		sin 1000
1 ₁₈	20 ₂ - 14	r
1 ₂₁	1 ₇ - 13	Link 8, (1, 21 - 14, 0, 1, 20)
2 ₈	A.12/1	
1 ₃₀ (1 ₇ , 21 ₃ - 14)		r sin 1000
1 ₂₀	1 ₂₂ - 13	Link 9, (1, 1 - 1, 0, 0, 1)
4 ₂₈	P.01	punch r sin 1000
1 ₃₀ (1 ₂₂ , 1 - 1)		

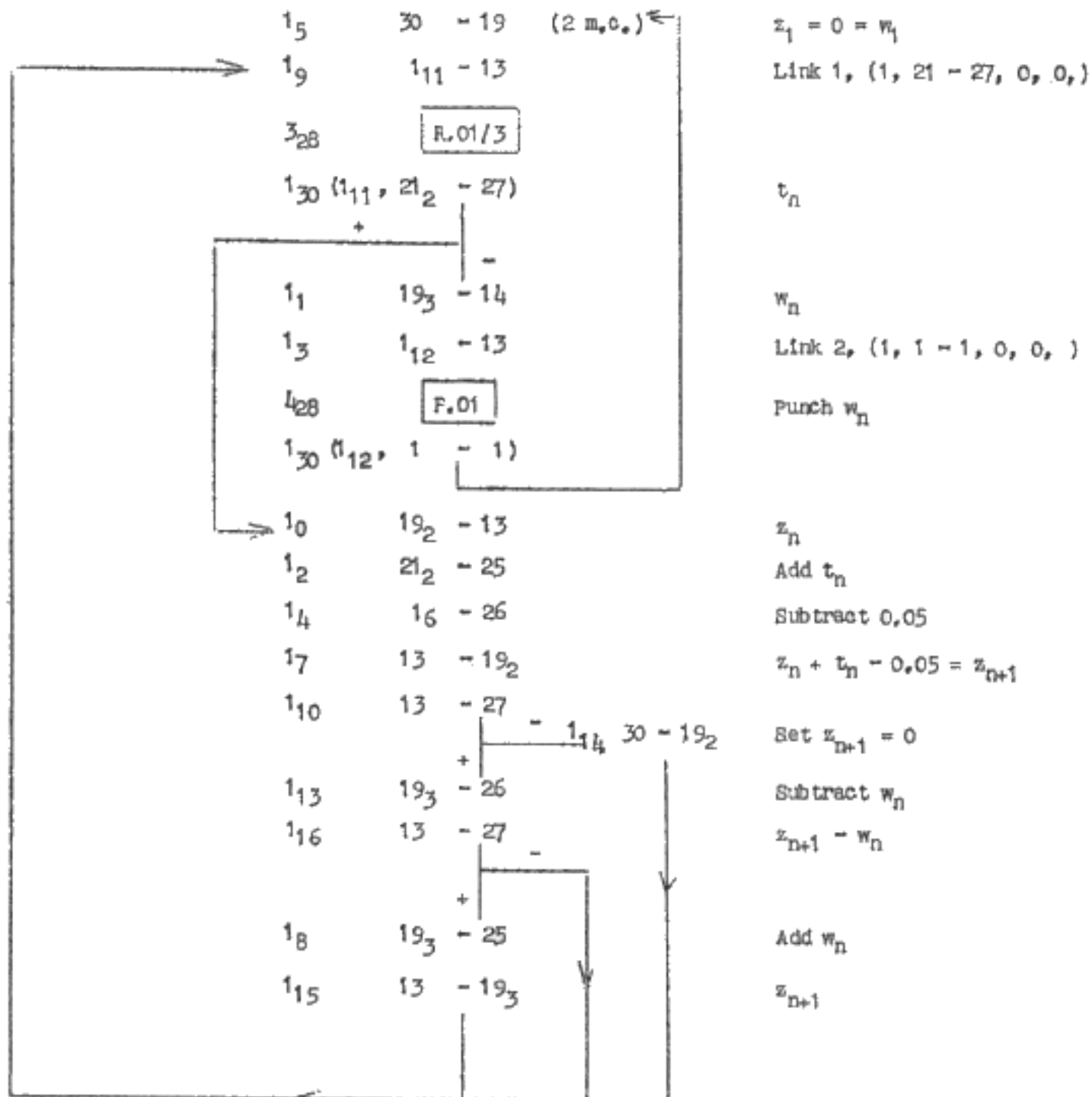
11. Let the number on the n'th card be t_n . Suppose the n'th patient arrives at time x_n and the doctor starts to deal with him at time y_n . (These times are measured in minutes divided by 100.) Then $x_n - x_{n-1} = 0.05$ and $y_n - y_{n-1} = t_{n-1}$, because the n'th patient goes in when the (n-1)'th comes out. The n'th patient therefore waits for a time $z_n = y_n - x_n$, and $z_n - z_{n-1} = t_{n-1} - 0.05$. We shall store z_n in 19₂ and $w_n = \max z_r$ for $r = 1, 2 \dots n$ in 19₃.

$z_n < 0$ means that the doctor is ready before the patient arrives. As the patient cannot wait for a negative time we shall set $z_n = 0$ in such a case and show the patient in as soon as he arrives.

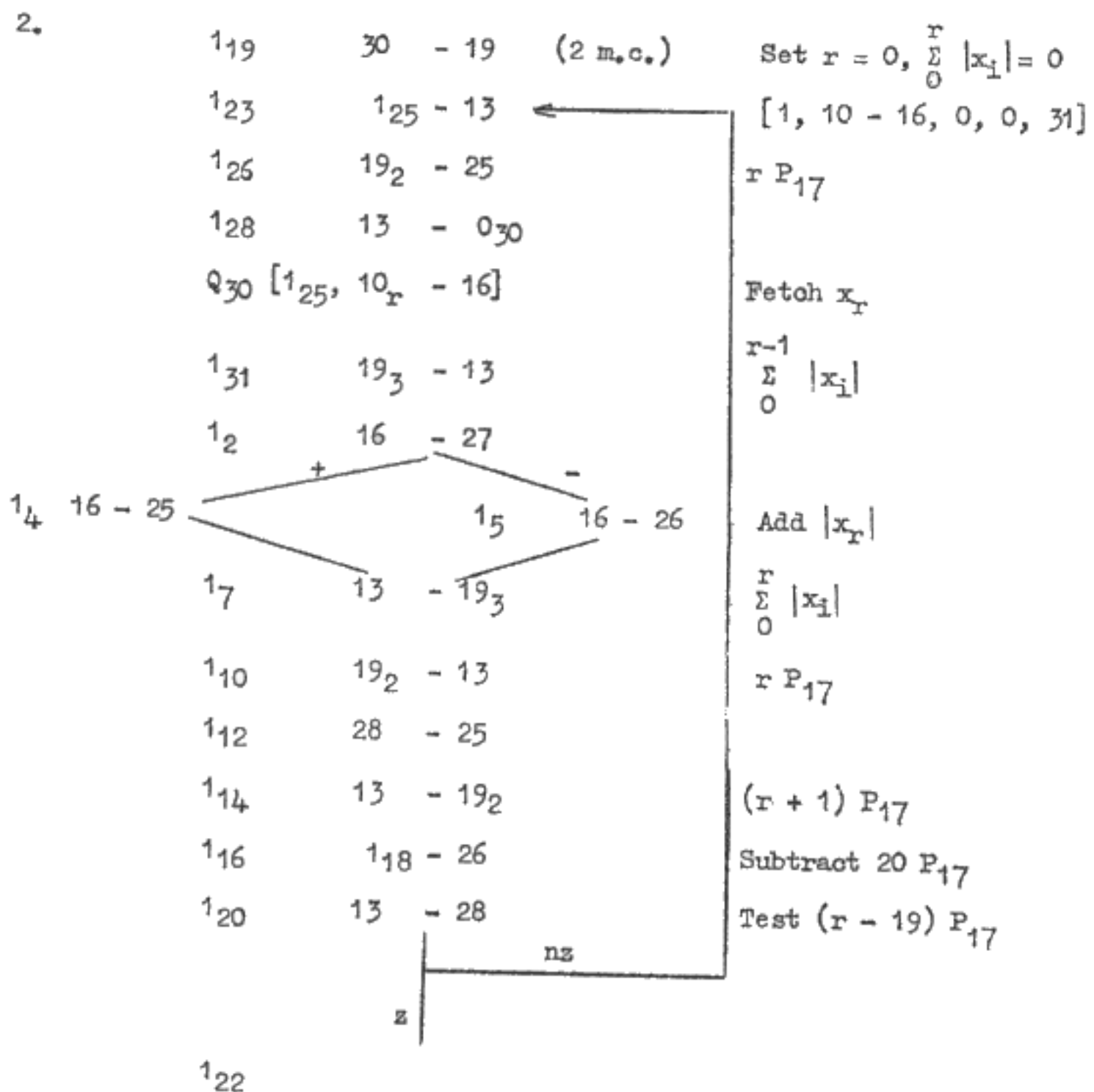
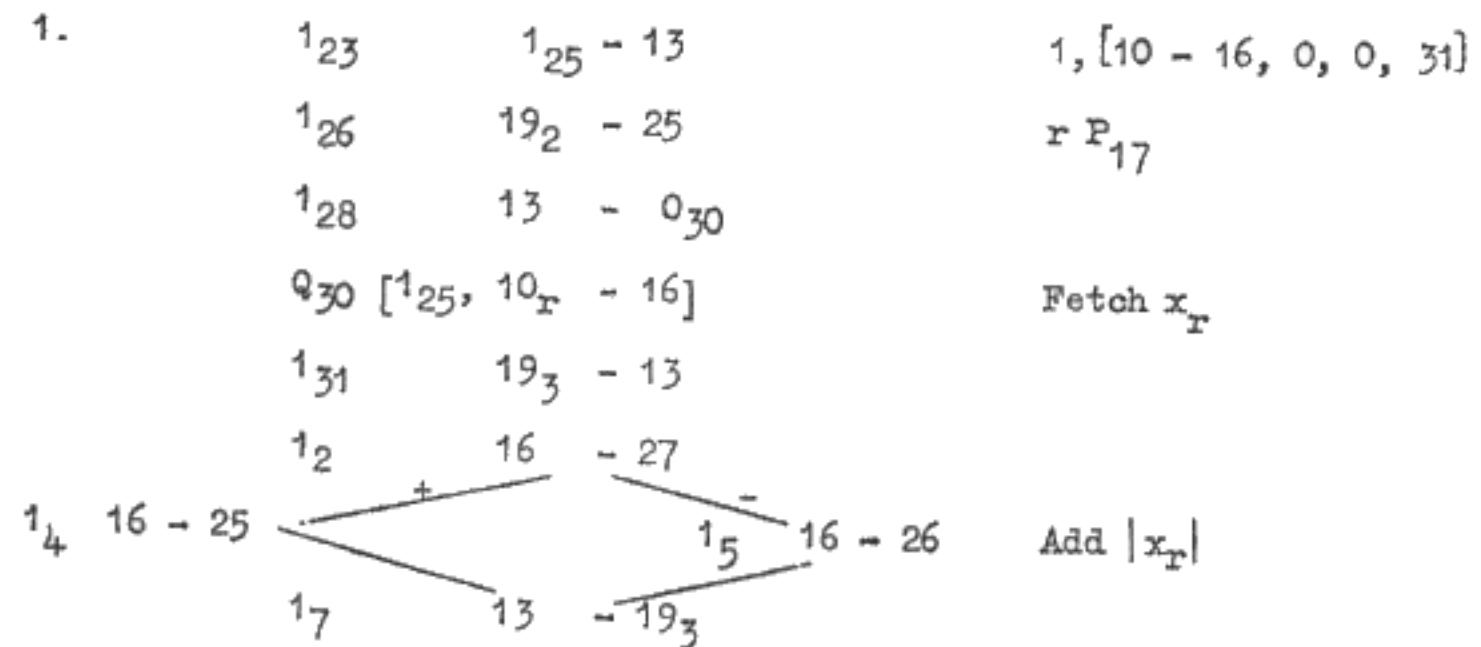
We know $z_1 = 0$, so the other initial condition is $w_1 = 0$. The block flow diagram is:



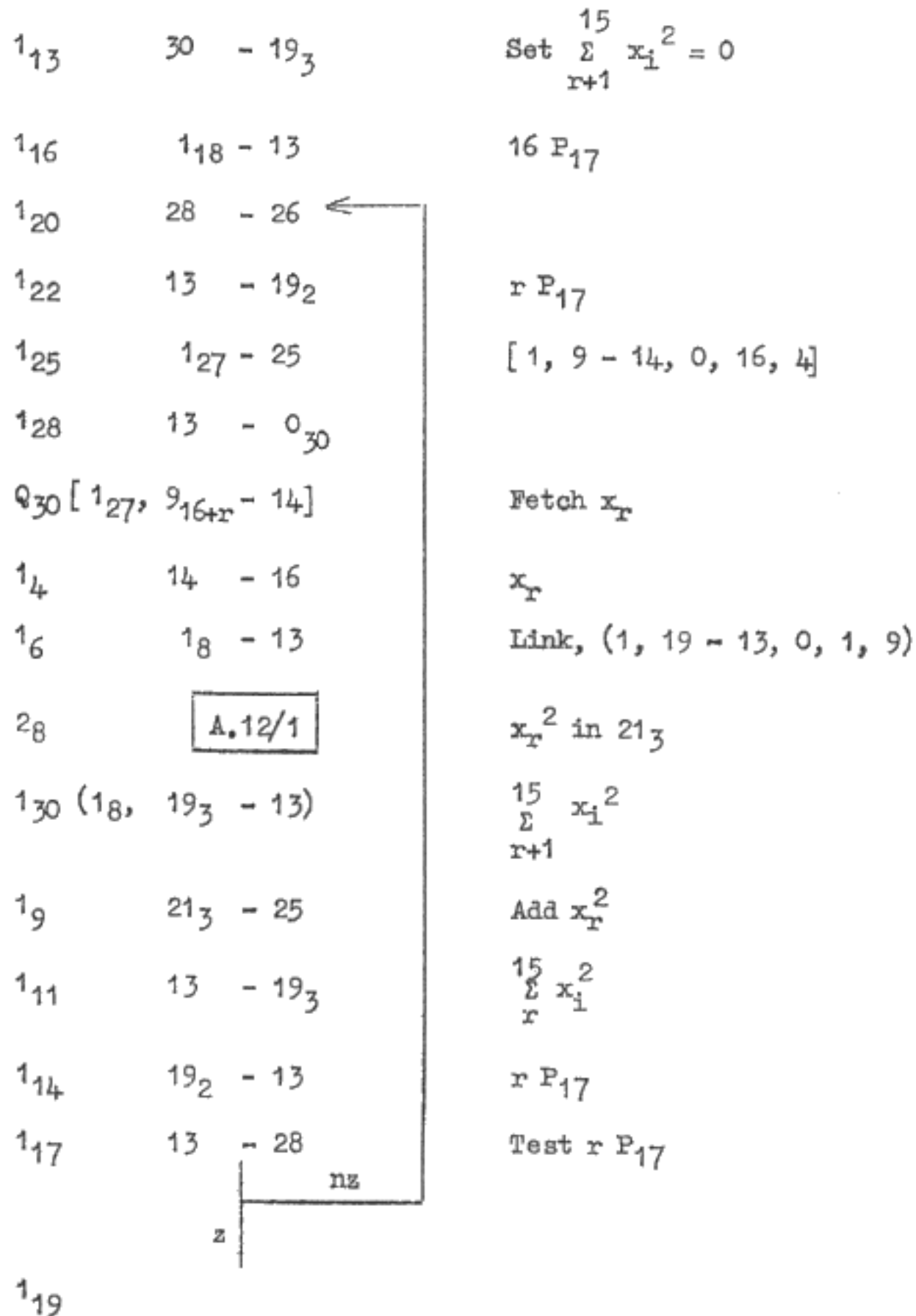
The program is:



Exercises F The solutions given here do not use A.I.M. In several cases this would result in a slightly shorter program.

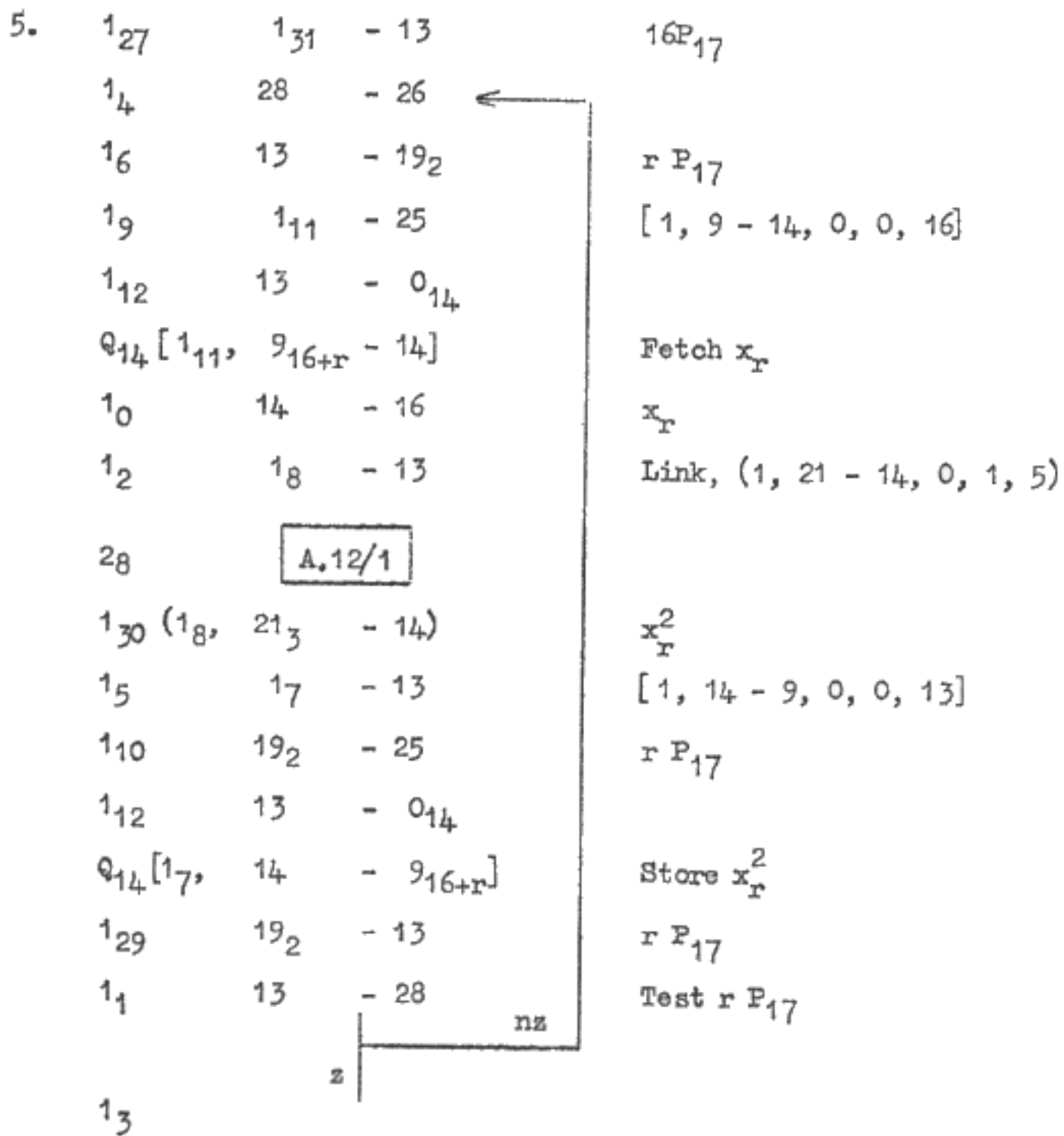


3. Because this exercise is similar to the previous one, the solution given uses a count going down from 16 to 0 so that the reader can see the relative merits of counting up and down when either can be used.



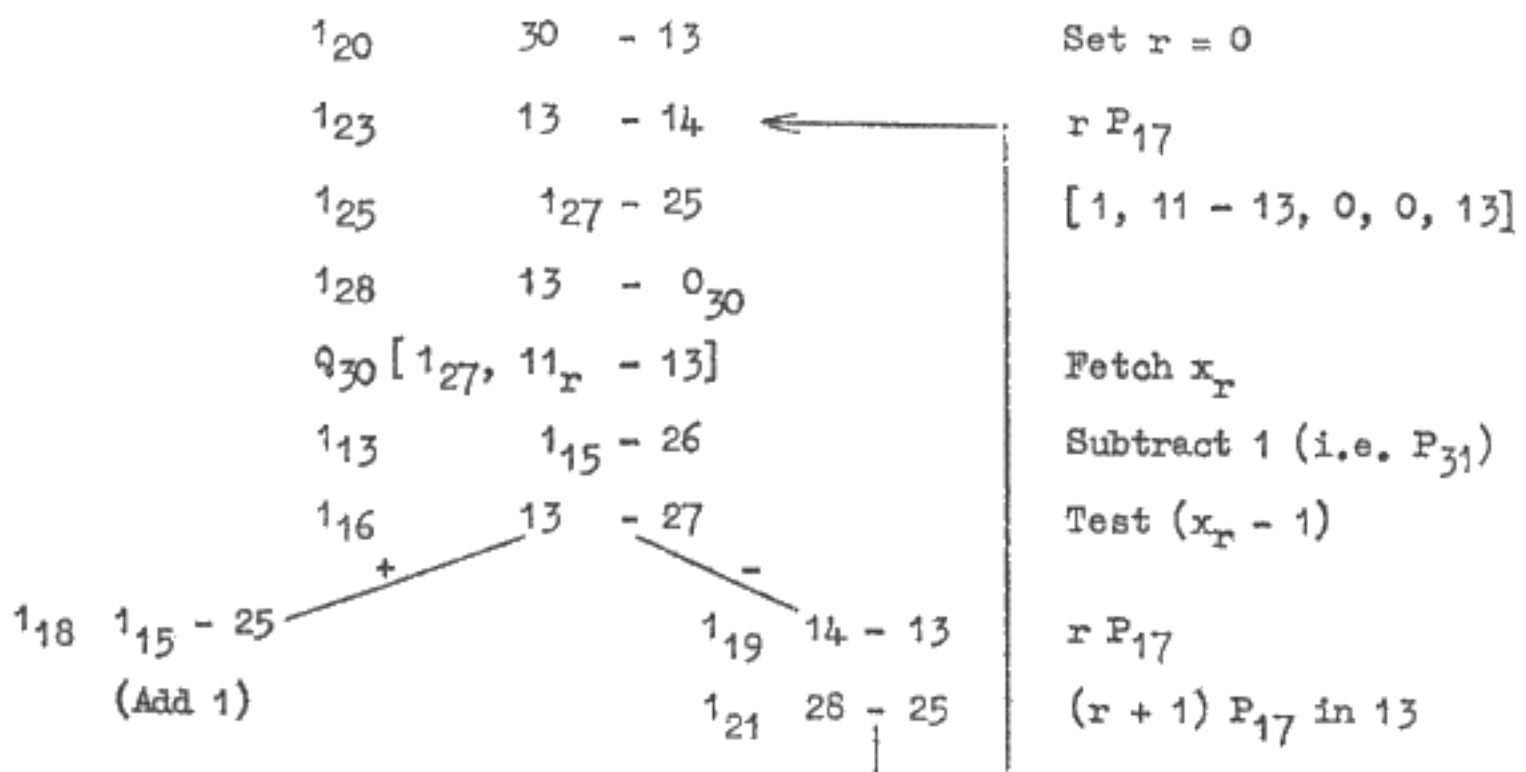
4.

- (i) If the number in 1_{18} is changed to $32P_{17}$, the program forms $C^2(9_{15}) + C^2(9_{14}) + \dots + C^2(9_0) + C^2(9_{31}) + \dots + C^2(9_{16})$.
- (ii) If the number in 1_{18} is changed to $33P_{17}$, the program forms $C^2(9_{16}) + C^2(9_{15}) + \dots + C^2(9_0) + C^2(9_{31}) + \dots + C^2(9_{16})$

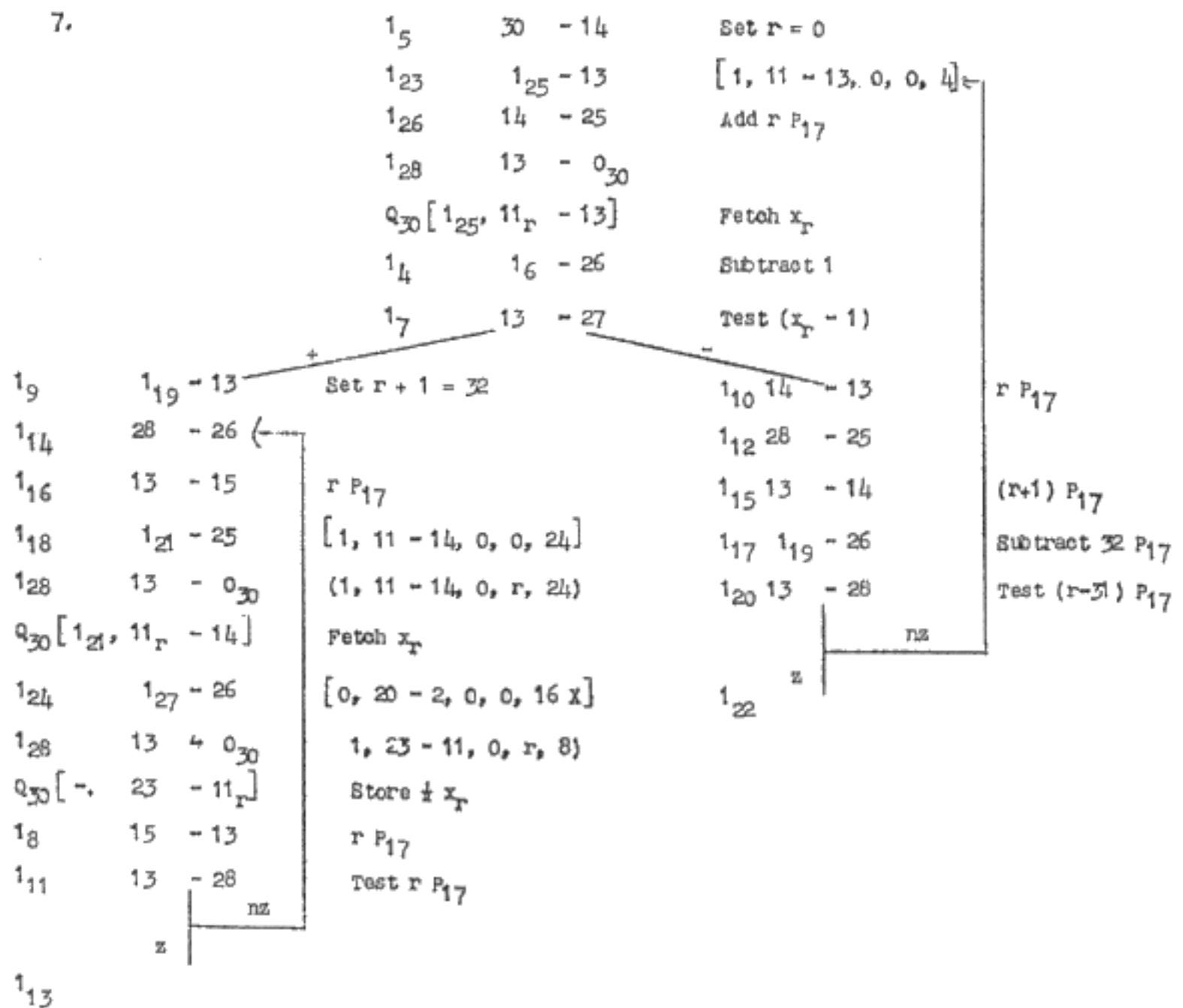


The first instruction cannot be placed in 1_{28} because A.12/1 needs the usual $13 - 0_{30}$ instruction there.

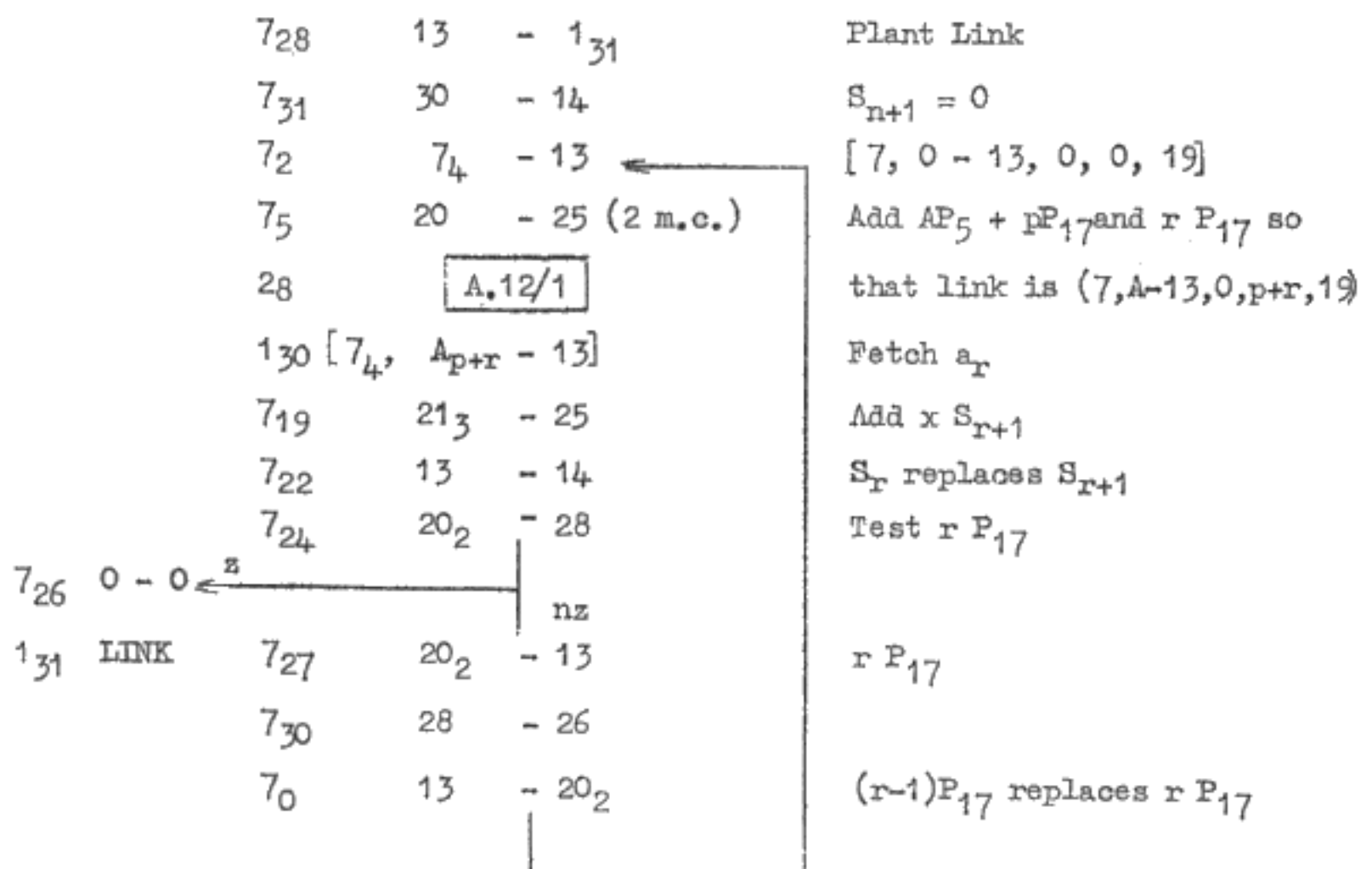
6.



In this example the count must go up from zero.



8. Let $s_r = a_r + x a_{r+1} + \dots + x^{n-r} a_n$. Then $s_r = a_r + x s_{r+1}$ and $y = s_0$



Second order subroutine.

In this example the counting has been done by the method not recommended in I.8, namely,

Test Count

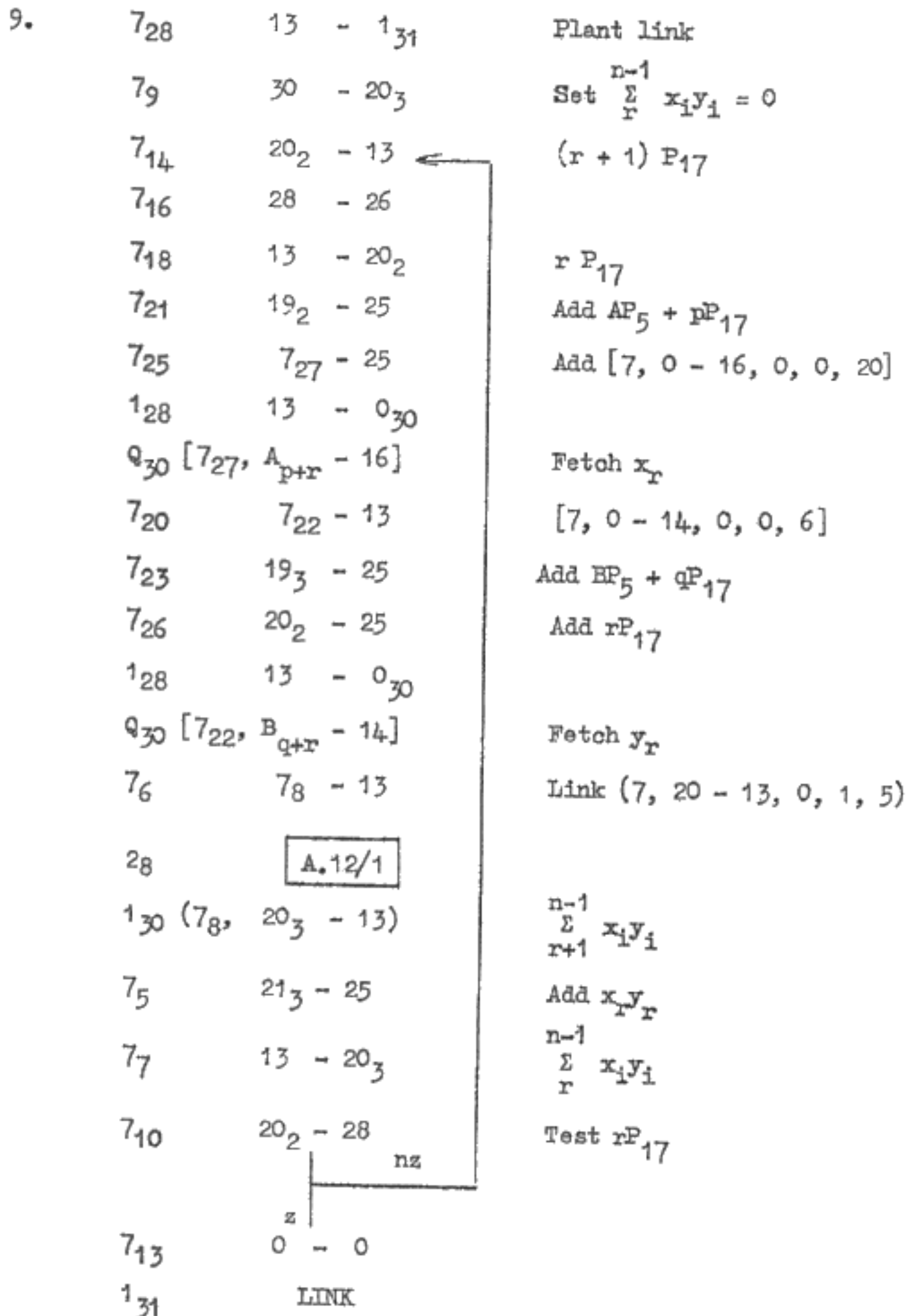
Reduce Count.

This was done because $(n + 1)$ numbers a_r have to be fetched - the working part of the loop must be traversed $(n + 1)$ times - but the count starts at n , not $(n + 1)$. The recommended method

Reduce Count,

Test Count,

could be used provided the count is tested with destination 27 and the loop left when it goes negative.



For the sake of completeness we give alternative counting, using A.I.M., to some of the examples above.

2a.

$${}^1_{20}: \quad {}^1_{22,23} = {}^{18}_{2,3}(2 \text{ m.c.}) \quad \begin{aligned} G(1_{22}) &= (1, 10-16, 0, 31, (0), 31), \\ G(1_{23}) &= 20 \text{ P}_{17} \end{aligned}$$
$$1_{24}: \quad 30 \quad -13 \quad \text{Set } \Sigma |x_i| = 0$$
$${}^1_{26}:(10 P1) \quad {}^{18}_2 \quad - \quad {}^0_{30}, 1$$
$$Q_{30}[(1_{22})]: 10_r - 16]$$
Fetch x_r
$$1_{31}; \quad + \quad \frac{16}{-27}$$
$$1_1: 16 - 25 \qquad 1_2: 16 - 26$$
Add $|x_T|$
$$^1_4(\mathbb{P}^1) \quad 18_3 \quad -0_7, 0$$

Reduce Count

$$^{18}_8\text{O} + ^{18}_8\text{O} \rightarrow ^{36}_{16}\text{S}$$

Test Count. (This instruction must not be in 17 - why?)

24

$$1_{25} \quad 13 \quad -19_3$$

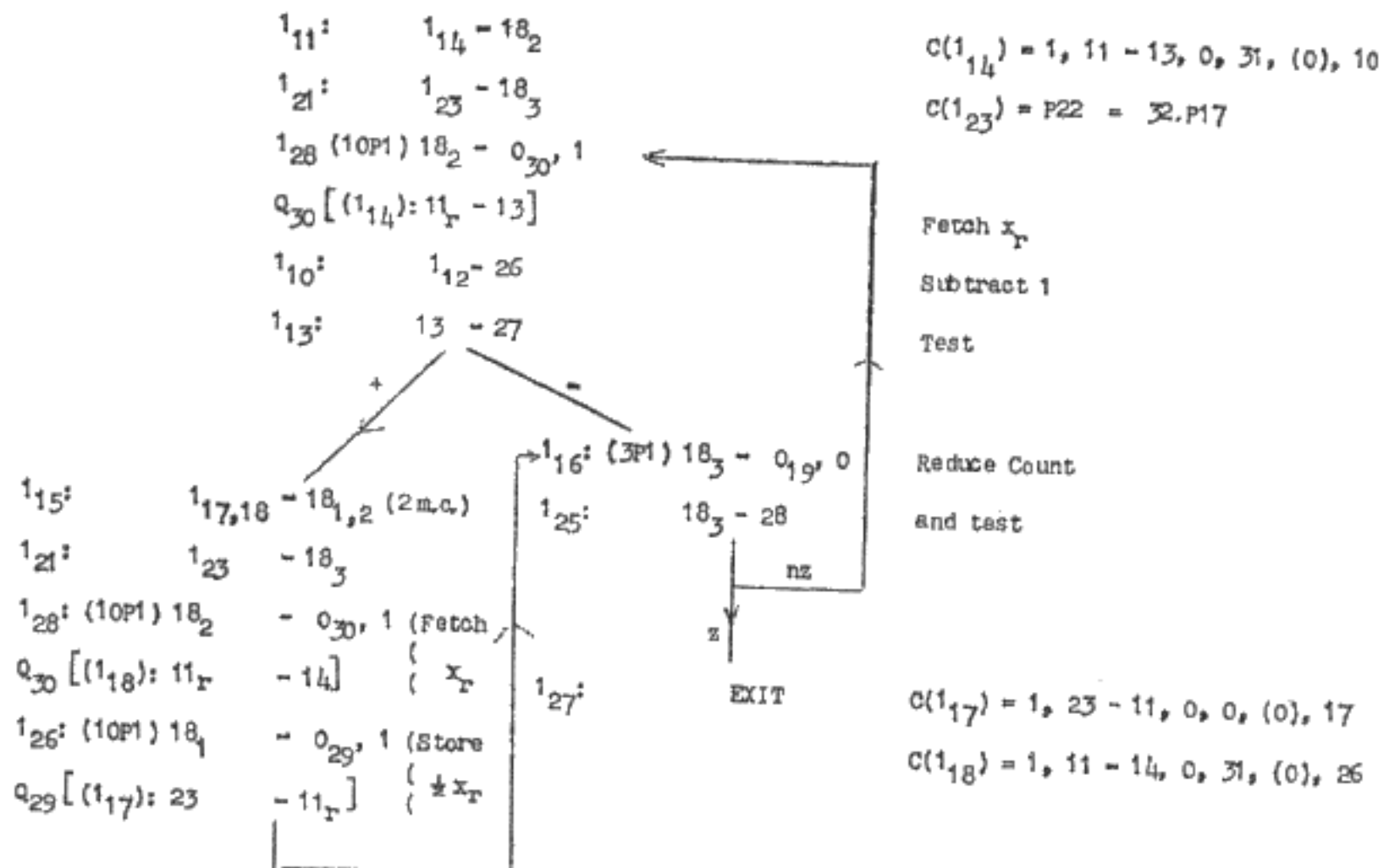
6a.

$${}^1 24_+ \quad {}^1 26 - 17_2 \quad O({}^1 26) = (1, 11 - 13, 0, 31, (0), 31)$$
$$1_{28}: (\mathbb{P}^1) \quad 17_2 \quad - \quad 0_{30}, 1 \leftarrow$$
$$Q_{30} [(1_{26}) : 11_r - 13]$$
Fetch x_r
$${}^1_{31}:\quad {}^1_1 = 26$$

Subtract P31 (= 1 to 30 b.p.)

$$1_2: \quad 13 \quad - \quad 27$$
$${}^1_{27}\text{Zn} \quad {}^1_1\text{H} = 25$$

7a.



The instruction in 121 is used to set the counter for both loops, and in each case $116, 125$ are the count and test instructions. If numbers have to be halved the test instruction leads to the 128 written on the left-hand side of the page. Spill-over could have been used and made a saving in the number of instructions and time taken. There would then have been two exits.

LIST OF REFERENCES

(Books are denoted by an asterisk)

<u>Ref. No.</u>	<u>Author</u>	<u>Title, etc.</u>
1	Barritt, M.M. Burnett-Hall, D.G.	Notes on Decimal-Binary and Binary-Decimal Conversion. M.S. Computing Note, (Series A) No.103. 1956.
2	Davis, G.M.	DEUCE Programming Manual. English Electric Co. Report. 1955.
3	Davis, G.M.	DEUCE Logical Design Manual. English Electric Co. Report NSy 37. 1956.
4	Denison, S.J.M. Hawkins, E.N. Robinson, C.	DEUCE Alphacode. English Electric Co. Report NSy 87. (This is DEUCE News No.20.) 1958.
	Hawkins, E.N.	Supplement to DEUCE News 20. English Electric Co. Report No. NSy 114. 1958.
5	Dodd, K.N.	A simplified coding system for DEUCE. R.A.E. Technical Note No. M.S.39. 1957.
6	*Richards, R.K.	Arithmetic Operations in Digital Computers. (New York). 1955.
7	Samet, P.A.	A detailed coding programme for DEUCE. R.A.E. Technical Note No. M.S.37. 1957.
8	Samet, P.A.	Operating instructions for the Detailed Coding Programme (R.A.E.415). M.S. Computing Note (Series B), No.217. 1957.
9	Watt, J.M.	The assembly of large programmes for the automatic computer DEUCE. R.A.E. Technical Note No. M.S.31. (1956).
10	*Wilkes, M.V.	Automatic Digital Computers (London). 1956.
11	Wilkinson, J.H.	An assessment of the system of optimum coding used on the pilot automatic computing engine at the National Physical Laboratory. Philosophical Transactions of the Royal Society, (Series A), Vol.248, p.253-281. 1955.
12	-	Interpretive and Brick schemes with special reference to matrix operations. DEUCE News 10. 1956.

ATTACHED: Appendices 1-4
 Drg. Nos. M.S. 3052, 3817 - 3822
 Detachable Abstract Cards

ADVANCE DISTRIBUTION LIST:

Headquarters

Chief Scientist
 DCA (RD)
 DGLRD
 DSR(L)
 DGW (Tech)
 AD/AR
 DGW(B)
 TIL 150
 Pats 1B

Establishments

D/NGTE
 D/RRE
 D/ARDE
 D/RPE
 CS/A & AEE
 D/SRDE

R.A.E.

Director
 DD(A)
 DD(E)
 Head of Aero Dept
 " GW "
 " Structures Dept
 " Armaments "
 " Radio "
 " IAP "
 " Electrical "
 " Instrumentation Dept
 " MS Dept 50
 CS/RAE Bedford
 Pats 1B(1)
 Secretary, Digital Computing Panel,
 Library 3

APPENDIX 1

FUNCTION CODE OF DEUCE

The Instruction Word

1	111	11111	11111	11	11111	1111	11111	1	1
A	NIS	S	D	Ch	W	(J)	T		X
1	2-4	5-9	10-14	15-16	17-21	22-25	26-30	31	32

(2)

Sources and Destinations

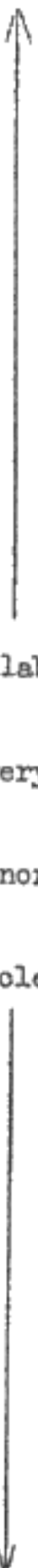
	<u>Source</u>	<u>Destination</u>	<u>N.I.S.</u>
0	Reader or I.D.	TS Count	DL 8
1	DL 1	DL 1	DL 1
2	DL 2	DL 2	DL 2
3	DL 3	DL 3	DL 3
4	DL 4	DL 4	DL 4
5	DL 5	DL 5	DL 5
6	DL 6	DL 6	DL 6
7	DL 7	DL 7	DL 7
8	DL 8	DL 8	-
9	DL 9	DL 9	
10	DL 10	DL 10	
11	DL 11	DL 11	
	(DL 11 associated with magnetic drum)		
12	DL 12	DL 12	
13	TS 13	TS 13	
14	TS 14	TS 14	
15	TS 15	TS 15	
16	(a) TS 16	(b) TS 16	
17	QS 17	QS 17	
18	QS 18	QS 18	
19	DS 19	DS 19	
20	DS 20	DS 20	
21	DS 21	DS 21	
22	(o) (DS 21) + 2	(c) Add to DS 21	
23	(TS 14) ÷ 2	(o) Subtract from DS 21	
24	(TS 14) × 2	(d) Triggers	
25	(TS 14) & (TS 15)	Add to TS 13	
26	(TS 14) ÷ (TS 15)	Subtract from TS 13	
27	P 1	(e) Discriminator, sign	
28	P 17	(e) Discriminator, zero	
29	P 32	Punch or O.S.	
30	0	(d) Magnetic read or write	{ Ch. even - read.
31	"Ones" = -P1	(d) Magnetic head shift	{ Ch. odd - write
(a)	DL 10 delayed by one m.c. if TCA on.	(c) Behaviour depends on state of TCB.	
(b)	Puts TCA off.	(d) Special source numbers.	

(e) Discriminations:-

A discrimination is coded to lead to the OFF side, which is the earlier of two consecutive instructions

	OFF (Normal)	ON (Abnormal)
D 27	+	-
D 28	z	nz
2-24	No TIL	TIL

Numbering of minor cycles

D.L. D.L.1-12 Also S.16 if TCA on and D.O.	Q.S. Q.S.17,18	D.S. D.S.19-21 Also S 22,D.22,23	T.S. T.S.13-16 Also S.0,23-31 and D.25-29
0	0	2	 Available every minor cycle
1	1	3	
2	2	2	
3	3	3	
4	0	2	
5	1	3	
6	2	2	
7	3	3	
8	0	2	
9	1	3	
10	2	2	
11	3	3	
12	0	2	
13	1	3	
14	2	2	
15	3	3	
16	0	2	
17	1	3	
18	2	2	
19	3	3	
20	0	2	
21	1	3	
22	2	2	
23	3	3	
24	0	2	
25	1	3	
26	2	2	
27	3	3	
28	0	2	
29	1	3	
30	2	2	
31	3	3	

Automatic instruction modification

y	17 - 0, 0	17 - 0, 1
0 - 3	$+P_x$	$+P_x$
4 - 7	$+P_x$	$+P_x + P_{22}$
8 - 11	$+P_x$	$-P_x$
12 - 15	$+P_x$	$-P_x - P_{22}$

y	18 - 0, 0	18 - 0, 1
0 - 3	$-P_x$	$-P_x$
4 - 7	$-P_x$	$-P_x - P_{22}$
8 - 11	$-P_x$	$+P_x$
12 - 15	$-P_x$	$+P_x + P_{22}$

The successive values of P_x in each group are $P_5, P_{10}, P_{17}, P_{18}$.
y represents the value of the four digits P_1, P_2, P_3, P_4 .

APPENDIX 2

BINARY ARITHMETIC

Scale 2

In the scale of 10 (the decimal scale) the number 142 means $1 \cdot 10^2 + 4 \cdot 10^1 + 2 \cdot 10^0$ and 0.625 means $0 \cdot 10^0 + 6 \cdot 10^{-1} + 2 \cdot 10^{-2} + 5 \cdot 10^{-3}$. There are ten possible values of each digit.

In the scale of 2 (the binary scale) there are only two possible values for each digit, 0 and 1. The number 10001110 means $1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$ - it is 142 in the decimal scale - and 0.101 means $0 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3}$, which is 0.625 in decimal.

When working on DEUCE it is customary to write the least significant binary digit on the left, reversing the usual procedure for decimal digits. Thus

$$142 \quad \equiv \quad 01110001$$

$$0.625 \quad \equiv \quad 101.0$$

$$142.625 \quad \equiv \quad 101.01110001$$

When numbers written with the least significant digit on the left (as is done throughout the remainder of this section) are added together, the least significant digits are still added first, so the result is written down from left to right. The least significant digits are added and the sum digit written down. There is also a carry digit which has been written down in examples 1 to 4 immediately below the sum digit and immediately transferred to the next column to be added with the next least significant digit. Normally the carry is added mentally without being written down. To clarify the process examples have been given in decimal as well as binary.

Example 1 Add 975 and 264 in decimal (Least significant digit on the right.)

975	a
264	b
<u>110-</u>	Carry from previous digit
<u>1239</u>	Sum
0110	Carry to next digit

Example 2 Add 0101101 and 0011011 in binary. (Least significant digit on the left.)

a	0101101
b	0011011
Carry from previous digit	<u>-0001111</u>
Sum	<u>01100011</u>
Carry to next digit	00011110

The numbers are 96 and 108 in decimal and the sum (correctly) is 198.

Example 3 Subtract 53 from 225 in decimal. (Least significant digit on the right.)

225	a
53	b
<u>010-</u>	Negative carry from previous digit
<u>172</u>	Difference
010	Negative carry to next digit.

Example 4 Subtract 101011 from 10000111 in binary. (Least significant digit on the left.)

a	10000111
b	10101100
Negative carry from previous digit	<u>-0011110</u>
Difference	<u>00110101</u>
Negative carry to next digit	00111100

The numbers are 53 and 225 in decimal, and the difference (correctly) is 172.

Example 5 Multiply 11001 by 11011 (in binary)

11001
011001
0000000
00011001
000011001
<u>1000000001</u>

(In decimal, $19 \times 27 = 513$.)

Example 6 Divide 1111110111 by 10101 (in binary).

10101
<u>1111110111</u>
10101
<u>111110001</u>
10101
<u>1110111</u>
10101
<u>110001</u>
10101
<u>0111</u>

Quotient 101101, remainder 0111.

(In decimal, dividing 959 by 21 gives quotient 45, remainder 14.)

Negative numbers

If 437 on the setting levers is subtracted from 0 in the product register of a desk machine, the result is 999563. The negative of any decimal can be obtained in this way, writing the nines-complement of every digit except the least significant non-zero digit, where the tens-complement is written. A negative number has a string of 9's at the most significant end, a positive number has a string of 0's.

This is the most convenient method of representing a negative number in binary inside a digital computer. The negative of 00101011011000.... is 00110100100111.... A negative number will have a 1 in the most significant digit and a positive number will have a 0.

Just as 999563 can be used satisfactorily on a desk machine instead of -437 for addition and subtraction but not for multiplication and division, so, on a digital computer special care has to be taken over negative numbers in multiplication and division, but not in addition and subtraction.

Adding a 0 to the least significant end of 000437 gives 004370 multiplying the number by 10: (a fixed word length of 6 decimal digits is being used because most digital computers have words of fixed length). The negatives of the two numbers are 999563 and 995630: so a negative number represented in the complement form can be multiplied by 10 by shifting it up one place.

Shifting 000440 down one place gives 000044, a tenth of the number. The complements are 999560 and 999956, so shifting a negative number in complement form down one place divides by 10 provided the new most significant digit is a copy of the old one.

Similar statements are true in binary. Shifting up 101011011000 and 110100100111 up one place gives 010101101100 and 011010010011 respectively, doubling the numbers. (A fixed word length of twelve binary digits is used in these examples.) Shifting down 000111011000 and 000100100111 by one place gives 001110110000 and 001001001111 respectively, halving the numbers, provided the most significant digit (the "sign digit") is a copy of the previous one.

Of course shifting down does not always give correctly rounded division by 10 or 2: shifting down 000437 gives 000043 whereas 000044 is the correctly rounded value, and the same applies in binary.

Scale of 32

In transcribing strings of 32 digits, all ones and noughts, there is a considerable possibility of error. The usual system on DEUCE is to divide the number by commas, starting from the binary point, and then treating each group of five as an integer, to write its decimal equivalent*. Thus

001100001011111000011010000111.10

becomes 00110,00010,11111,00001,10100,00111.10 and is written

12 , 8 , 31 , 16 , 5 , 28 . 1.

* A list of the binary equivalents of the integers from 0 to 31 is given on page 4 of Ref.1.

A more sensible system would have been to divide the binary digits into groups of three. There would then be no need to write commas between the resulting "decimal" digits and tables (Ref.1) exist for conversion between decimal and octal.

Probably the original choice was prompted by the fact that Hollerith cards have rules dividing the columns into fives. On a desk machine conversion to and from the scale of 32 is quicker than the scale of 8 because fewer operations are necessary.

It can be seen that this is equivalent to writing the number in the scale of 32. The numbers $12.32^{-6} + 8.32^{-5} + 31.32^{-4} + 16.32^{-3} + 5.32^{-2} + 28.32^{-1} + 1.32^0$ and $(0.2^{-30} + 0.2^{-29} + 1.2^{-28} + 1.2^{-27} + 1.2^{-26}) + \dots + (0.2^{-5} + 0.2^{-4} + 1.2^{-3} + 1.2^{-2} + 1.2^{-1}) + 1.2^0$ are identical.

It is sometimes necessary (particularly when testing programs) to convert numbers between the scales of 10 and 32. The methods for doing this on various makes of desk machine will be found in Ref. 1. The process should be avoided where possible. If a constant has to be put into a program in binary it is better that the Detailed Coding program (R.A.E.415, Refs.7 and 8) should be used.

Numbers "to p b.p."

On DEUCE numbers are often described as having p binary places. This means that the number has p binary digits below (i.e. less significant than) the binary point. Of course the position of the point does not matter when numbers are added or subtracted but it does affect multiplication and division. Subroutines for multiplying and dividing 30 binary place numbers are most commonly used. It is much better to describe all numbers to 30 b.p. e.g. $\pi/4$ and $2^{-4} \log_2 x$, rather than $\pi(28 \text{ b.p.})$ and $\log_2 x (26 \text{ b.p.})$, although the two descriptions are synonymous. It takes considerable thought to decide what is the product of $\pi(28 \text{ b.p.})$ and $\log_2 x (26 \text{ b.p.})$ when a 30 b.p. multiplication subroutine is used. It takes much less thought to find the product of $\pi/4$ and $2^{-4} \log_2 x$, so fewer mistakes will be made.

It is unfortunate that most library subroutines use the $\pi(28 \text{ b.p.})$ approach. A programmer using such subroutines should immediately translate all such references into numbers with the binary point in the position he is using.

APPENDIX 3

USEFUL CONSTANTS

All values expressed in the scale of 32,
with the most significant digit on the right.

10	.10									$\cdot 10^{-1}$	26,	12,	6,	19,	25,	12,	6,	3.
10^2	. 4,	3								$2^5 \cdot 10^{-2}$	24,	21,	7,	10,	24,	21,	7,	10.
10^3	. 8,	31								$2^5 \cdot 10^{-3}$	25,	11,	26,	13,	18,	24,	0,	1.
10^4	.16,	24,	9							$2^{10} \cdot 10^{-4}$	28,	24,	26,	5,	14,	27,	8,	3.
10^5	. 0,	21,	1,	3						$2^{15} \cdot 10^{-5}$	14,	28,	8,	12,	13,	17,	15,	10.
10^6	. 0,	18,	16,	30						$2^{15} \cdot 10^{-6}$	27,	2,	20,	23,	23,	17,	1,	1.
10^7	. 0,	20,	5,	17,	9					$2^{20} \cdot 10^{-7}$	29,	21,	6,	5,	31,	11,	11,	3.
10^8	. 0,	8,	24,	11,	31,	2				$2^{25} \cdot 10^{-8}$	2,	6,	2,	23,	3,	19,	23,	10.
10^9	. 0,	16,	18,	21,	25,	29				$2^{25} \cdot 10^{-9}$	19,	0,	29,	11,	16,	11,	2,	1.
10^{10}	. 0,	0,	25,	23,	0,	10,	9			$2^{30} \cdot 10^{-10}$	24,	14,	22,	31,	13,	30,	13,	3.
10^{11}	. 0,	0,	26,	13,	7,	4,	29,	2		$2^{35} \cdot 10^{-11}$	6,	15,	1,	31,	31,	26,	31,	10.
10^{12}	. 0,	0,	4,	10,	10,	10,	3,	29		$2^{35} \cdot 10^{-12}$	23,	4,	16,	25,	28,	5,	3,	1.
$1/3$	21,	10,	21,	10,	21,	10,	21,	10.		$1/7$	18,	4,	9,	18,	4,	9,	18,	4.
$1/4$	0,	0,	0,	0,	0,	0,	0,	8.		$1/8$	0,	0,	0,	0,	0,	0,	0,	4.
$1/5$	19,	25,	12,	6,	19,	25,	12,	6.		$1/9$	18,	3,	7,	14,	28,	24,	17,	3.
$1/6$	11,	21,	10,	21,	10,	21,	10,	5.		$1/10$	26,	12,	6,	19,	25,	12,	6,	3.
$1/21$	0,	0,	0,	0,	0,	0,	0,	16.		$1/91$	7,	30,	14,	28,	2,	0,	0,	0.
$1/31$	11,	21,	10,	21,	10,	21,	10,	5.		$1/101$	20,	28,	7,	9,	0,	0,	0,	0.
$1/41$	11,	21,	10,	21,	10,	21,	10,	1.		$1/111$	25,	28,	26,	0,	0,	0,	0,	0.
$1/51$	2,	17,	8,	4,	2,	17,	8,	0.		$1/121$	23,	7,	2,	0,	0,	0,	0,	0.
$1/61$	27,	2,	12,	11,	16,	13,	1,	0.		$1/131$	17,	5,	0,	0,	0,	0,	0,	0.
$1/71$	13,	0,	20,	1,	16,	6,	0,	0.		$1/141$	13,	0,	0,	0,	0,	0,	0,	0.
$1/81$	2,	16,	6,	0,	26,	0,	0,	0.		$1/151$	1,	0,	0,	0,	0,	0,	0,	0.
π	6,	4,	2,	21,	22,	31,	16,	4.3		$\log_{10} e$	28,	4,	21,	24,	30,	22,	28,	13.0
e	11,	20,	24,	2,	21,	16,	31,	22.2		$\log_e 10$	11,	21,	29,	14,	3,	27,	21,	9.2
$\sqrt{2}$	20,	31,	25,	12,	30,	4,	8,	13.1		$\log_2 e$	24,	21,	20,	12,	7,	10,	5,	14.1
γ	30,	27,	24,	15,	6,	2,	15,	18.0		$\log_e 2$	18,	30,	29,	15,	1,	25,	5,	22.0
$\sqrt{3}$	5,	12,	1,	29,	26,	19,	13,	23.1		$1/\pi$	7,	25,	13,	3,	12,	30,	5,	10.0
$1/\sqrt{3}$	2,	4,	11,	20,	19,	6,	15,	18.0		$1/e$	20,	5,	22,	17,	21,	22,	24,	11.0

Powers of 2

n	2^n	n	2^n
1	2	36	6
2	4	37	13
3	8	38	27
4	16	39	54
5	32	40	109
6	64	41	219
7	128	42	439
8	256	43	879
9	512	44	1759
10	1024	45	3518
11	2048	46	7036
12	4096	47	14073
13	8192	48	28147
14	16384	49	56294
15	32768	50	112589
16	65536	51	225179
17	131072	52	450359
18	262144	53	900719
19	524288	54	1801439
20	1048576	55	3602879
21	2097152	56	7205759
22	4194304	57	14411518
23	8388608	58	28823037
24	16777216	59	57646075
25	33554432	60	115292150
26	67108864	61	230584300
27	134217728	62	461168601
28	268435456	63	922337203
29	536870912	64	1844674407
30	1073741824		
31	2147483648		
32	4294967296		
33	8589934592		
34	17179869184		
35	34359738368		

Negative Powers

2^{-3}	0.125	10^5	2^{-8}	390.625
2^{-4}	0.0625	10^5	2^{-10}	97.65625
2^{-5}	0.03125	10^5	2^{-16}	1.52587
2^{-6}	0.015625	10^{10}	2^{-32}	2.32830
2^{-7}	0.0078125			64365
2^{-8}	0.00390625			38696
				28906
				25

APPENDIX 4

HOLLERITH TIMINGS

To the programmer the important times are the minima between events. He must treat them as his maxima. His program should then work on any DEUCE at any time.

The engineer must see that the measured time lies between the maximum and the minimum. If it does not, the machine is unserviceable.

All times are given in major cycles unless otherwise stated.

<u>Program number</u>	<u>Test</u>	<u>Specification</u>	
		<u>Reader</u>	<u>Punch</u>
<u>Time during which row is available</u>			
TH 01/1	Minimum time after the single-shot for which source 0 gives the word punched in any row.	2 min.	-
TH 02/1	Maximum time after the single-shot for which source 0 gives the word punched in any row.	6 max.	-
	Minimum time after the single-shot for which destination 29 is punched on row	-	5 min.
<u>Time between single-shots</u>			
TH 03	Minimum time between successive single-shots from same card.	15 min.	38 min.
TH 04	Maximum time between successive single-shots from same card.	21 max.	46 max.
TH 05	Minimum time between 9 row single-shot of one card and Y row single-shot of next.	80 min.	116 min.
TH 06	Maximum time between 9 row single-shot of one card and Y row single-shot of next.	100 max.	135 max.
<u>Length of TIL.</u>			
TH 07	Minimum time between 8 row single-shot and beginning of TIL.	11 min.	33 min.
TH 08	Minimum time between beginning of TIL and 9 row single-shot.	2 min.	2 min.
TH 09	Maximum time between beginning of TIL and 9 row single-shot.	7 max.	9 max.
TH 10	Time between 9 row single-shot and end of TIL when cards run continuously.	(18 min. 23 max.	35 min. 50 max.
TH 13	Time between end of TIL and Y row single-shot of next card.	(57 min. (	75 min. 90 max.
TH 16	Maximum time between stopping reader/punch by 9 - 24 X on 9 row and end of TIL.	23 max.	30 max.

<u>Program number</u>	<u>Test</u>	<u>Specification</u>	
		<u>Reader</u>	<u>Punch</u>
TH 11	Maximum time between stopping reader by hole in column 54 on 9 row and end of TIL,	23 max.	-
<u>Source 0 reverting to I.D.</u>			
TH 14	Maximum time between 9 - 24 X on 9 row and Source 0 giving I.D. keys.	13 max.	-
TH 12	Maximum time between hole in column 54 on 9 row and Source 0 giving I.D. keys.	13 max.	-
<u>Stopping at end of card - extra card</u>			
TH 20	Maximum time between 9 row single-shot and 9 - 24 on reader if card is not to be missed.	40 max.	-
TH 21	Maximum time between 9 row single-shot and 9 - 24 on reader if "Missed Card" warning is to be given.	70 max.	-
TH 22	Maximum time between 9 row single-shot and 9 - 24 on punch if no extra card is to be passed.		20 max.
<u>Stopping; restarting</u>			
TH 25	Maximum time between single-shot (not 9 row) and 9 - 24 to avoid further single-shots from same card.	13 max.	35 max.
TH 24	Minimum time between stopping and restarting in middle of card to avoid further single-shots from same card.	2 min.	2 min.
TH 15/1	Minimum time between single-shot where there is hole in column 54 and recalling reader.	0 min.	-
<u>Speed of operation</u>			
TH 18	Reader speed } Punch speed } in cards per minute.	190 min.	95 min.
TH 19		210 max.	105 max.

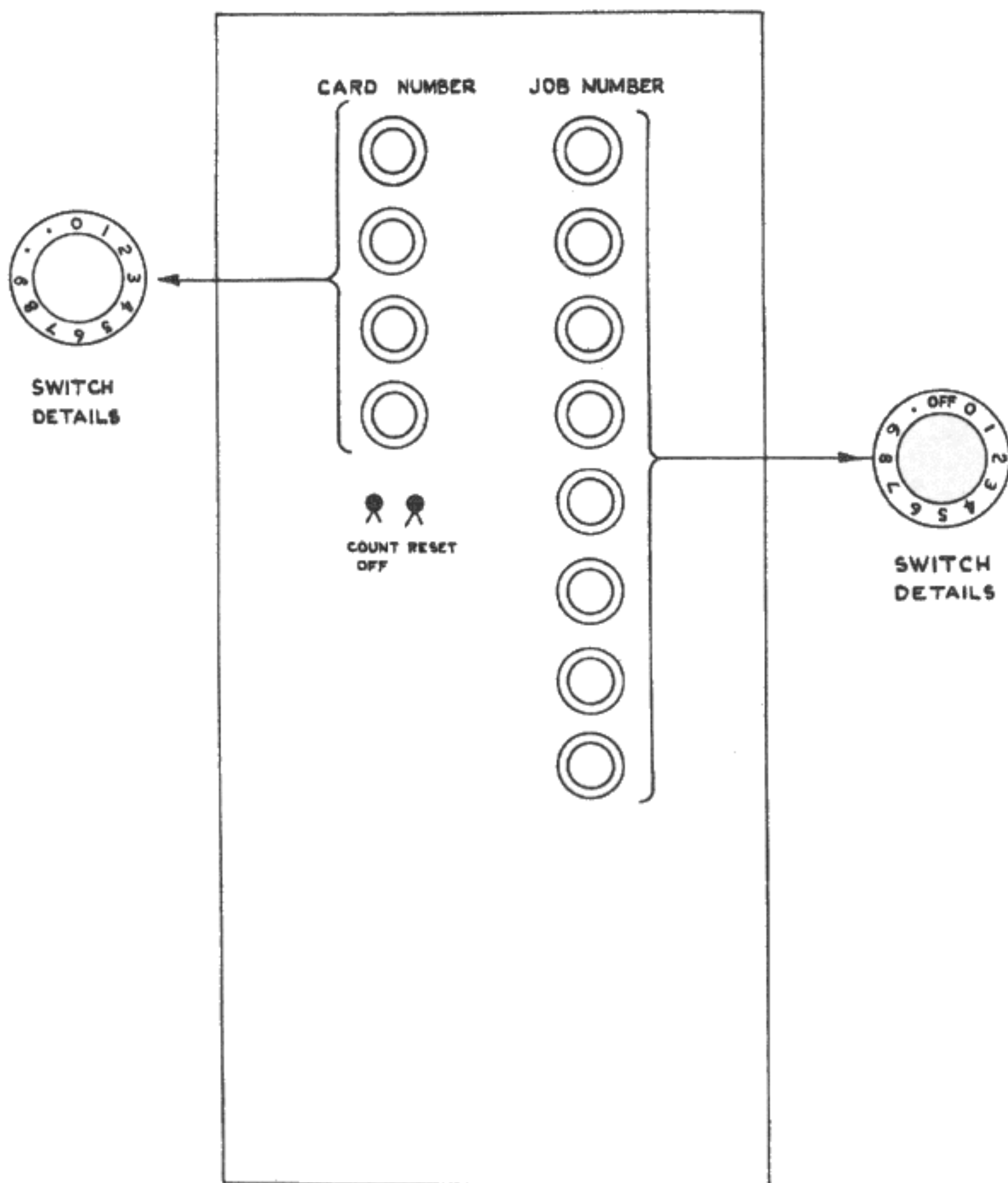
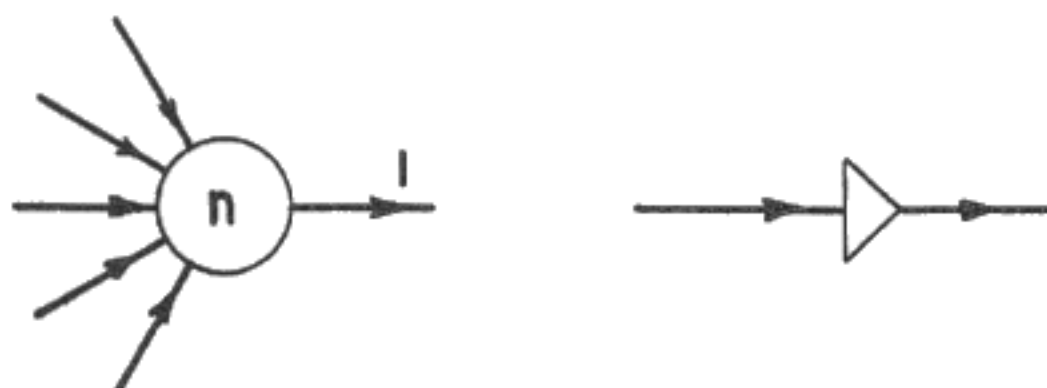


FIG. 3. NUMBERING SWITCHES ON THE PUNCH.



GATE WITH THRESHOLD n . DELAY OF 1 UNIT

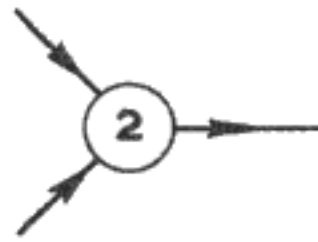


TRIGGER, (WITH 2 INPUTS) DELAY OF n UNITS

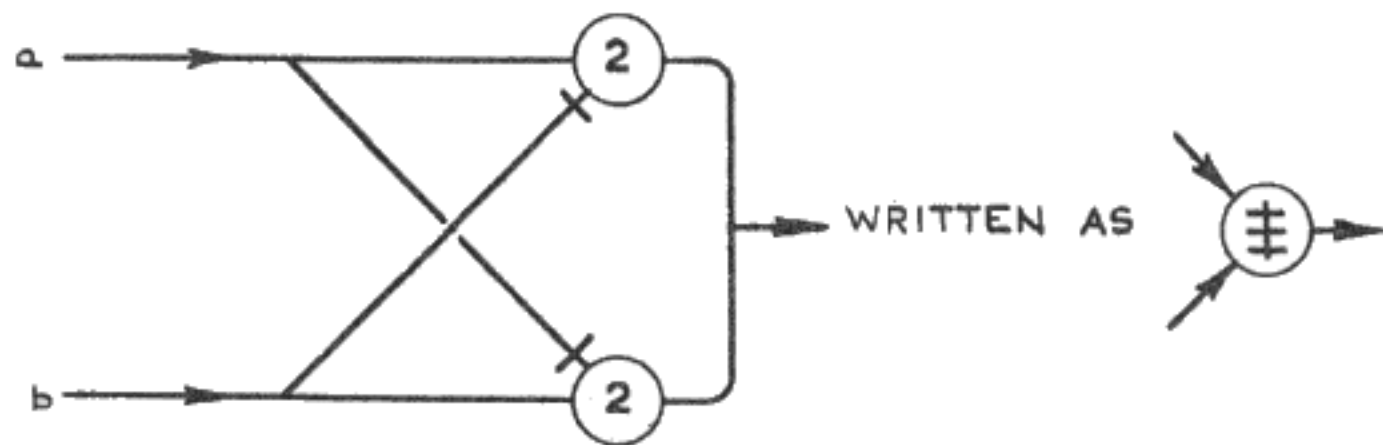


GATE, WITH INHIBITOR INVERTER (OR 'NEGATE')

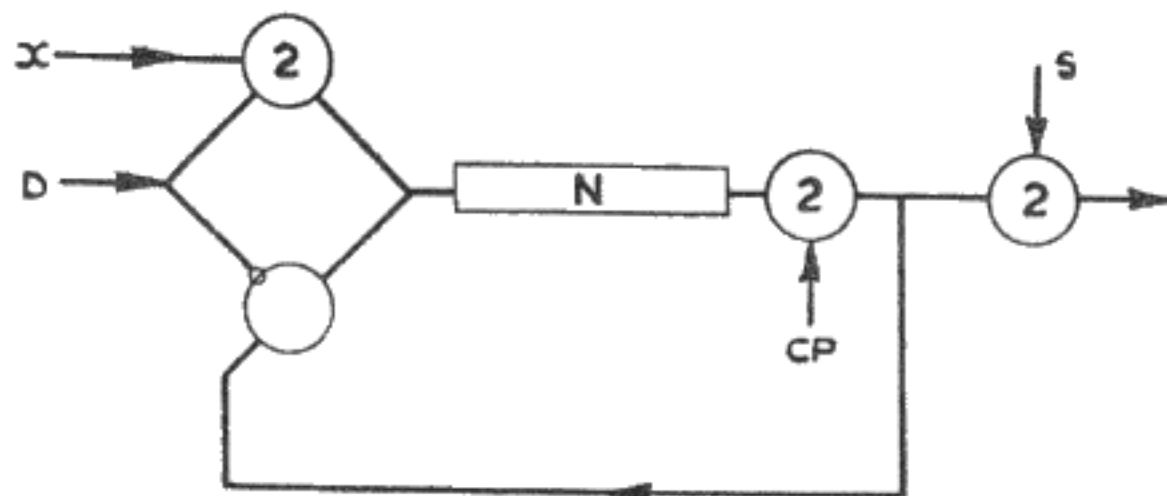
FIG. 4.



(a) THE LOGICAL FUNCTION '2'

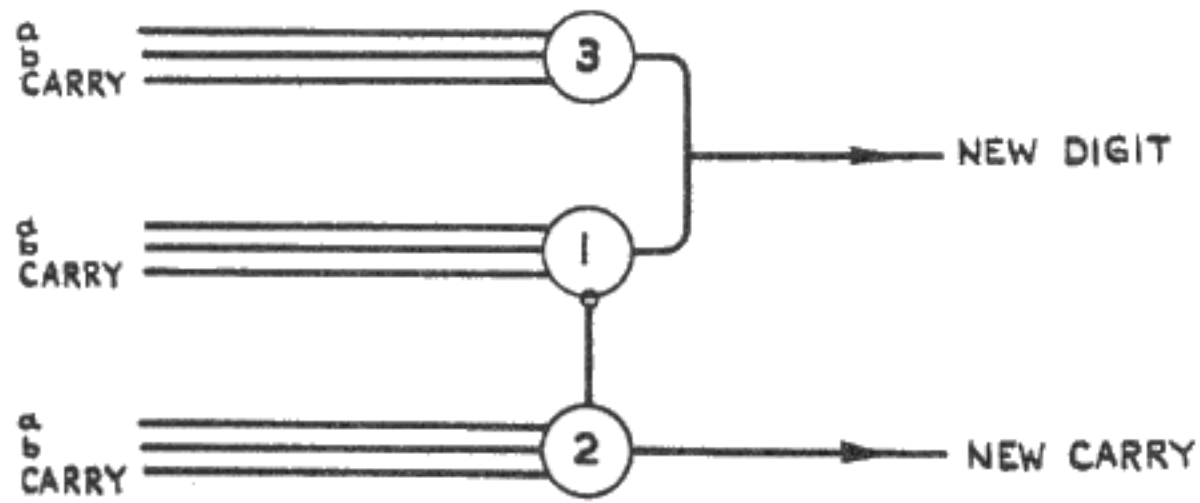


(b) THE LOGICAL FUNCTION $\equiv$

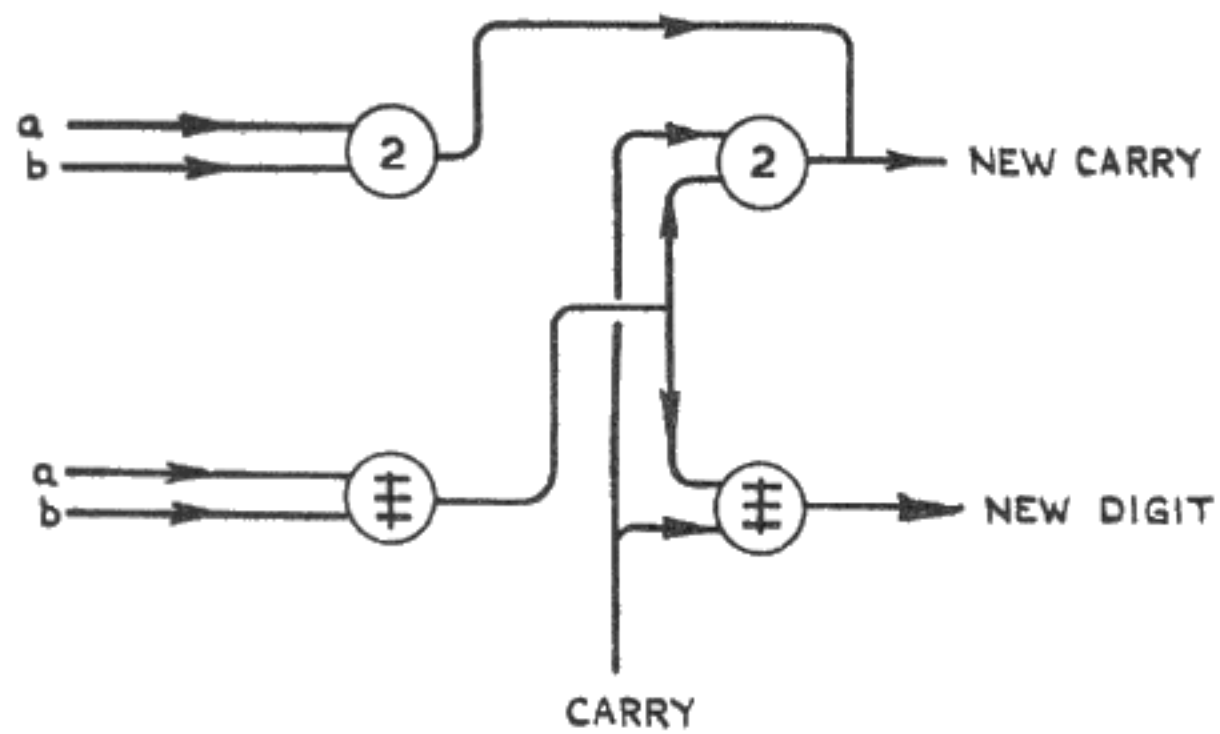


(c) A DELAY LINE

FIG. 5. (a-c).



(a) AN ADDER.



(b) AN ADDER COMPOSED OF TWO HALF-ADDERS.

FIG. 6.(a & b).

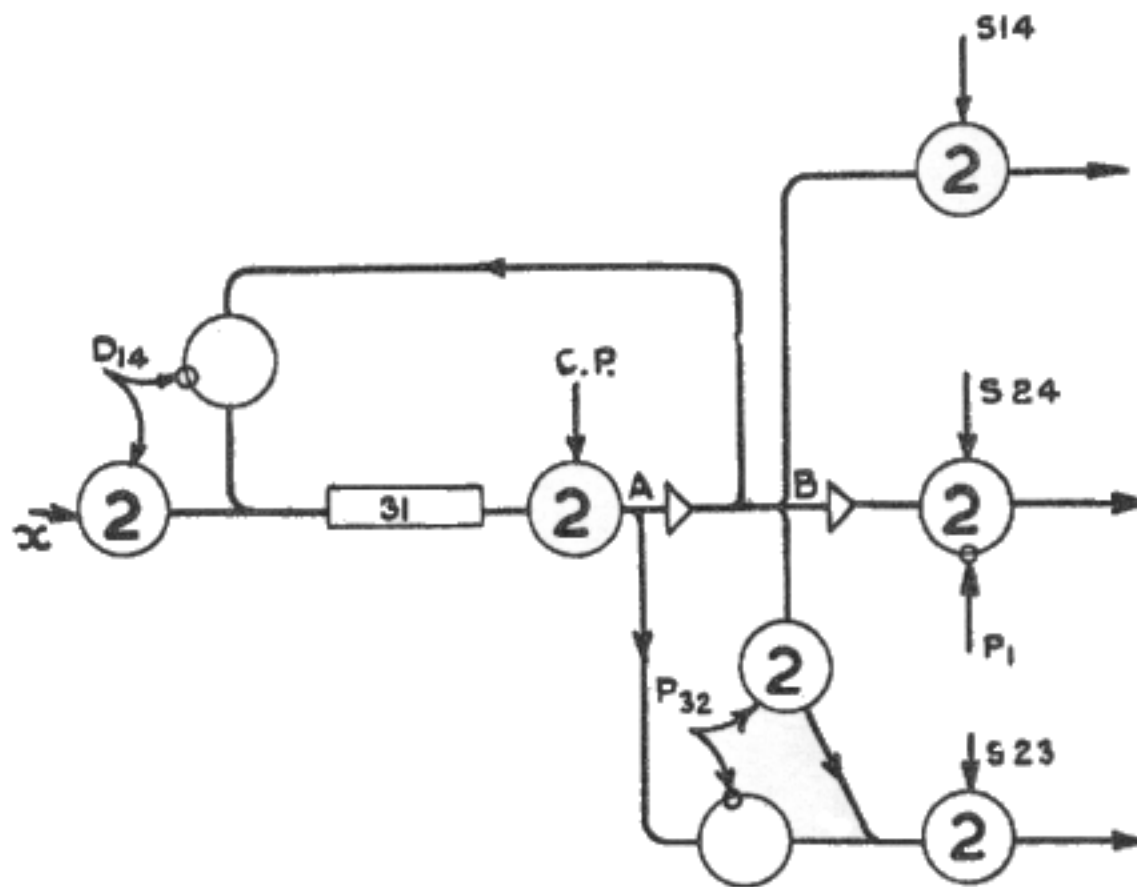


FIG. 7. SOURCES 14, 23 AND 24.

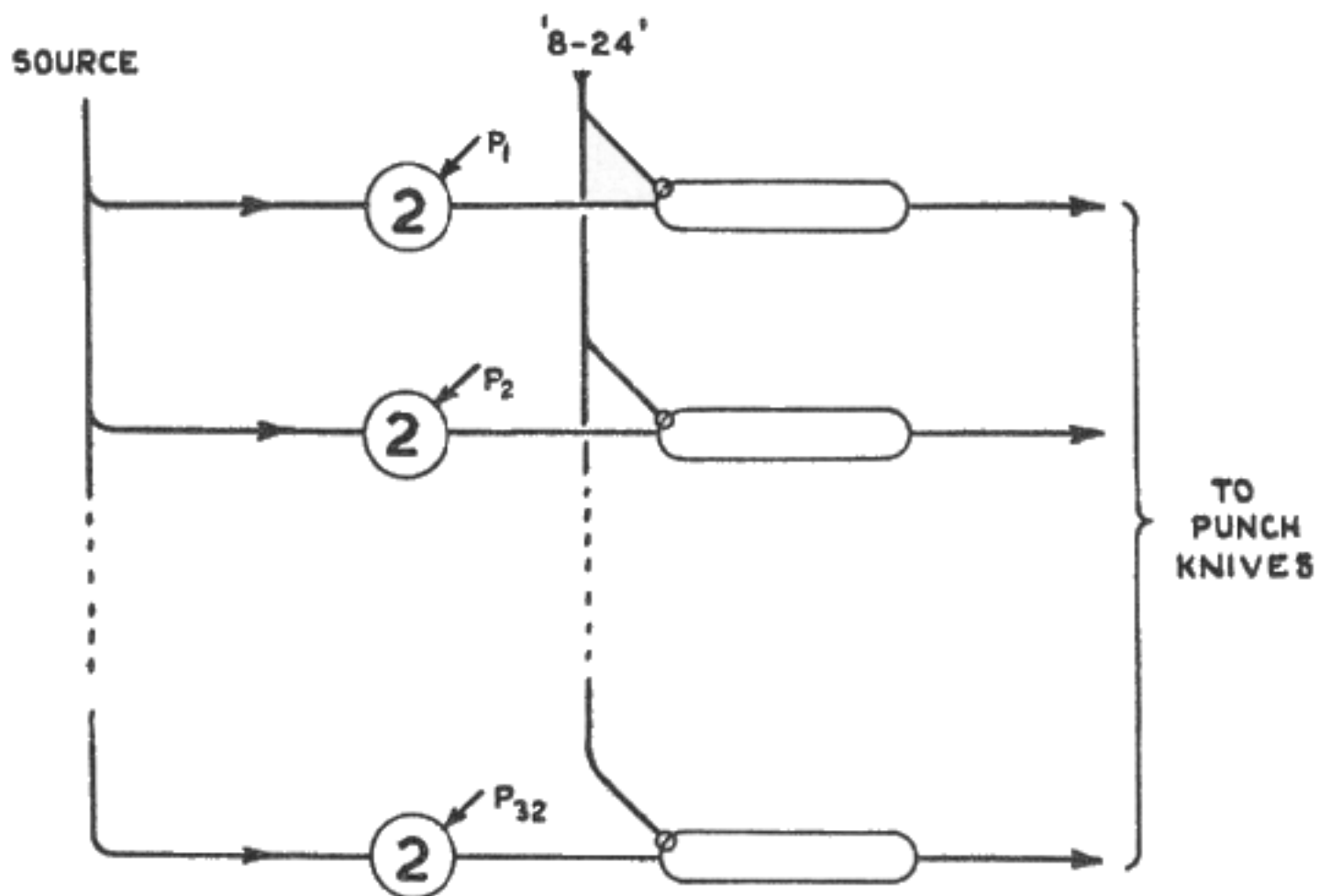


FIG. 8. DESTINATION 29,
THE OUTPUT STATICISER.

DETACHABLE ABSTRACT CARDS

These abstract cards are inserted in Reports and Technical Notes for the convenience of Librarians and others who need to maintain an Information Index.

Detached cards are subject to the same Security Regulations as the parent document, and a record of their location should be made on the inside of the back cover of the parent document.

UNCLASSIFIED

Technical Note No. M.S.38
Royal Aircraft Establishment
518.5
531.791
A PROGRAMMING HANDBOOK FOR THE COMPUTER DEUCE. Burnett-Hall, D.G. and Samet, P.A. April, 1959.

The technique of programming the digital computer DEUCE is described.

UNCLASSIFIED

UNCLASSIFIED

Technical Note No. M.S.38
Royal Aircraft Establishment
518.5
531.791
A PROGRAMMING HANDBOOK FOR THE COMPUTER DEUCE. Burnett-Hall, D.G. and Samet, P.A. April, 1959.

The technique of programming the digital computer DEUCE is described.

UNCLASSIFIED

UNCLASSIFIED

Technical Note No. M.S.38
Royal Aircraft Establishment
518.5
531.791
A PROGRAMMING HANDBOOK FOR THE COMPUTER DEUCE. Burnett-Hall, D.G. and Samet, P.A. April, 1959.

The technique of programming the digital computer DEUCE is described.

UNCLASSIFIED

UNCLASSIFIED

Technical Note No. M.S.38
Royal Aircraft Establishment
518.5
531.791
A PROGRAMMING HANDBOOK FOR THE COMPUTER DEUCE. Burnett-Hall, D.G. and Samet, P.A. April, 1959.

The technique of programming the digital computer DEUCE is described.

UNCLASSIFIED