The DEUCE Alphacode Translator

By F. G. Duncan, and D. H. R. Huxtable

A description is given of a recently completed program for translating from a single-level pseudo-code (Alphacode) to a multi-level machine code (orthodox DEUCE code). The chief point of interest is the allocation of the single-level addresses among the three levels of the real computer to obtain an efficient final program.

1. Introduction

At the Cambridge Conference of The British Computer Society in June 1959, Dr. S. Gill, reviewing progress in automatic programming, stated (Gill, 1959):

> "One thing which has caused trouble in the past, and which will probably continue to be a nuisance, is the problem of utilizing multi-level stores . . . . It turns out to be very difficult to make a multi-level machine imitate a single-level machine with high efficiency over a large class of problems. If, therefore, one is planning an autocode for a multi-level machine, the only simple solutions of this problem are:
>
> 1. To present the user with a multi-level hypothetical machine, with store levels corresponding to those in the real machine, and to let him decide how to allocate the available space for them.
> 2. To present the user with a single level of storage corresponding to the slowest level in the real machine."

Some examples of the second solution as applied to DEUCE were discussed in a recent paper (Robinson, 1959). These "autocodes" (G.I.P., T.I.P., Alphacode) are, it was pointed out, complementary to each other in that each is suitable for its own certain class of problem. For example, G.I.P. is best suited to parallel operations on blocks of data, as in matrix manipulation, while Alphacode (which is not unlike Pegasus Autocode in many ways) is best for problems of single variables.

A G.I.P. program is very nearly as efficient as the corresponding DEUCE program. The reason is that, although the interpretation cycle is slow ($\sim \frac{1}{2}$ sec), the instructions are powerful and call into operation highly efficient hand-made subroutines which are well buffered in the fast store. The interpretation time overall is negligible compared to the calculation time, and although the data are organized in the slowest part of the store, this happens to be the most efficient place. Alphacode, on the other hand, can be rather wasteful in its use of machine time. The interpretation cycle is quite fast ($\sim$ 17 msec), but this is by no means negligible when compared with the times for obeying instructions, which are normally concerned only with single operations. Also, whereas Alphacode is confined to the slow backing store, an orthodox program for a problem suitable for Alphacode can make very good use of the faster levels of store for numerical quantities. As a result of this wide difference in efficiency, we find that G.I.P. is used both for "one-off" and for repeated jobs, whereas Alphacode tends to be limited to "one-off" work. In all other respects, however, Alphacode has been a highly successful user code, being very popular among "non-professional" programmers.

It is clear that if the inefficiency of Alphacode can be overcome in some way, the usefulness of the scheme will be much increased, and it will be able to cope with a much wider range of problems. Two difficulties are involved in this. One is the elimination of interpretation time, which could be achieved by taking the pseudo-code instructions one at a time and "transliterating" them into machine instructions. This is a fairly trivial matter. The other, more serious, difficulty is that mentioned by Dr. Gill, namely the efficient use of the multi-level structure of the real computer.

The present paper is concerned with an attempt to overcome these difficulties. A program, called the Alphacode Translator, is described which accepts an Alphacode program and produces an equivalent, more efficient, program in orthodox DEUCE code. The translator has been in use since the end of 1959. The object program is often between five and six times as fast as its original; the cost in machine time is between 20 and 50 minutes for a translation, according to the size and complexity of the program. (These figures relate to the magnetic-tape version of the Translator; for the punched-card version the translation time is between 30 and 75 minutes.) The Translator itself has about 22,000 instructions (cf. 24,000 for FORTRAN) and took between four and five man-years to make. As far as the authors are aware, it represents the first successful attempt to program the efficient allocation of storage in a multi-level machine.

2. The Translator in use

Before describing the Translator it is convenient to describe how it is used. A program for translation is written in Alphacode and tested and corrected in the established manner, using the Alphacode Interpreter. The Interpreter has several useful program-testing facilities, such as optional punching of intermediate results, optional stopping, provision for altering instructions, and so on, and there is thus every hope of the program being right before translation is attempted. This is most important, since the translated (DEUCE)

program, being very economically constructed, will have no special facilities, and though it will be always accompanied by a printed DEUCE flow diagram, the latter may not be readily understood by the original Alphacode user. There should normally be no need for interference with the DEUCE program. If modifications are necessary after translation, it is much easier for the user, even the sophisticated user, to alter the original Alphacode pack and do a complete translation again, than to attempt to alter the DEUCE program. It is also probably less wasteful of machine time.

Forms of input and output, and details of arithmetic, are exactly the same for the DEUCE program as for Alphacode. Therefore it is possible to check the work of the Translator by running both Alphacode and DEUCE programs with the same case data, and comparing the results mechanically; this in fact is standard practice and seems quite satisfactory.

3. The Pseudo-Computer

(For a complete description of Alphacode, the reader is referred to the *Alphacode Manual*, English Electric, 1959.)

The pseudo-computer (that is, the imaginary machine for which the user thinks he is writing) has a uniformly accessible store of 2,200 or so addresses for floating-point numbers. These are called X1, X2, . . ., X2200, . . .

The normal form of the instruction is "three-address-plus-function":

$$X3 = X7 \div X29$$
$$X5 = X2 - X3$$

for functions of only one argument the second address is blank:

$$X4 = \log X5$$
$$R23 \quad X7 = \cosh^{-1} X40$$

An instruction can be labelled with a reference number, as in the case of the last one above, and a discrimination can lead to it:

(If) X3 equals X4 (jump to) R23.

The basic form is distorted to accommodate more powerful orders:

12 DATA X5

(i.e. Read 12 data into X5 to X16 inclusive).

There are 64 functions available, including floating-point arithmetic on real and complex numbers, circular and hyperbolic functions and their inverses, input and output in binary and decimal, Simpson's Rule integration, linear interpolation, and the solution of ordinary differential equations.

There are 63 counting registers, called N1, N2 . . . N63 (with the same access time as the X-stores), for fixed-point integers. Arithmetic may be performed on these, but their main use is for counting round loops and for modifying X-addresses; for example, if N4, N7, N10 happen to contain at a particular time the numbers 30, 27, 15 respectively, the effect of the pair of instructions:

N4 N7 modify N10
X3 = X75 ÷ X125

is the same as that of the single instruction:

X33 = X102 ÷ X140.

Instructions are obeyed serially, except at discriminations and jumps. Subroutines in Alphacode may be included in a program; special functions are available for planting and obeying links to these. Up to 27 subroutines may be used; they can be nested in any way and used in any order.

4. The Real Computer

The DEUCE store (more fully described by Haley, 1956) is made up of mercury delay lines and a magnetic drum as follows:

1. *Fast Store*

4 delay lines of one 32-bit word each (TS13, TS14, TS15, TS16).
3 delay lines of two 32-bit words each (DS19, DS20, DS21).
2 delay lines of four 32-bit words each (QS17, QS18).

The access time to any of these words is less than 0·1 msec.

2. *Main Store*

12 delay lines of 32 words each (DL1–DL12).

The access time to any word here is never more than 1 msec; it is usually much less, particularly if "optimum coding" (see, for example, Wilkinson, 1955), is used.

There are functional differences between the long delay lines. DL1–DL8 are the only part of the store in communication with control. Instructions stored in any other part of the machine must therefore be copied into these delay lines before they can be obeyed.

DL11 is the buffer delay line through which all information to or from the magnetic drum must pass.

3. *Backing Store*

Sixteen blocks of 16 tracks of 32 words each (8,192 words). Information is passed between the main store and the backing store through DL11 in units of a complete track. The transfer time for one track is about 15 msec, but if the mechanism has previously been ordered to operate on a different block a further 35 msec are required to change blocks, making 50 msec in all. Transfers to and from the drum are called *writing* and *reading* respectively.

Since transfers are done in complete tracks it is clear that in order to write a single word it is first necessary to read the appropriate track, then to amend it in the main store, and finally to write it. This point is important. Another important point is that, although a magnetic

operation takes time to complete, this time need not be wasted, for the computer will proceed with calculation, or any other logically independent operation, becoming interlocked only when another (dependent) magnetic transfer is called before the time has elapsed.

It follows from these points that in an efficient program references to the drum, particularly for writing, are cut down to a minimum, and those that remain are spaced as far apart from each other as possible.

The basic idea in the DEUCE instruction is of a transfer from a source (S) to a destination (D). Thus, for example, 13–14 means "put a copy of the contents of TS13 into TS14." Transfers for anything up to 32 word lengths can be specified. Thus 11–8 (32) means "transfer the whole of DL11 (the drum buffer) to DL8," while 10_{14}–21(2) means "transfer words 14 and 15 of DL10 to DS21."

Each instruction must specify its successor, for DEUCE instructions are not obeyed as they are stored; rather they are stored so that each one is accessible as soon as possible after the completion of its predecessor. (This is "optimum coding".)

DEUCE is essentially a fixed-point computer, so floating-point operations are carried out by means of subroutines. Now a subroutine needs to have its data and its link ready in standard positions, and it gives its result in a standard position. Floating-point numbers occupy two words each (one for the exponent, the other for the normalized mantissa). Therefore, all the ordinary functional subroutines, for use in translated programs, have been standardized so that the arguments are assumed in DS21 and DS20 (DS20 for a single argument) and the results are given in DS21. The links are assumed in TS16. Since complex numbers occupy four words each, the complex subroutines use QS17 and QS18 instead of DS21 and DS20. The more elaborate subroutines ("differential equations" and the like) are, unavoidably, exceptions to these rules.

It is now possible to see how Alphacode instructions could be "transliterated" into DEUCE instructions. For example,

$$X3 = X7 \div X29$$

could become:

(Read the X7 track to DL11)
11–21 (2)
(Read the X29 track to DL11)
11–20 (2)
(link) –16
(Enter and obey division subroutine)
(Read the X3 track to DL11)
21–11 (2)
(Write the X3 track from DL11)

The immediate reaction to a program written like this is one of horror. A simple Alphacode instruction to divide two floating-point numbers has given rise to a whole string of DEUCE instructions, including four magnetic transfers—more if the division subroutine is not already in the main store. This illustrates the futility of the "transliteration" approach by itself.

5. The use of the Fast Store

Let us consider a very simple Alphacode program, for finding the (real) roots of a quadratic equation

$$ax^2 + 2bx + c = 0.$$

This program has eleven instructions, as follows:

1		3	DATA	X1	(Reads a, b, c to X1, X2, X3)
2	X4	= X1	×	X3	(Calculates ac)
3	X5	= X2	×	X2	(Calculates b^2)
4	X6	= X5	−	X4	(Calculates $b^2 - ac$)
5	X7	=	ROOT	X6	(Calculates $\sqrt{(b^2 - ac)}$)
6	X8	= 0	−	X2	(Calculates $-b$)
7	X9	= X8	−	X7	(Calculates $-b - \sqrt{(b^2 - ac)}$)
8	X10	= X8	+	X7	(Calculates $-b + \sqrt{(b^2 - ac)}$)
9	X11	= X9	÷	X1	(Calculates one root)
10	X12	= X10	÷	X1	(Calculates the other root)
11		2	RESULTS	X11	(Punches roots from X11, X12)
			FINISH		

The computer, when it has the translated program, will be dealing with two distinct types of quantity—numbers and instructions. Now we have mentioned that instructions to be obeyed must be in part of the main store, so we can exclude them from consideration for fast-store use. The numbers are of two types—floating-point (X-addresses) and fixed-point (N-address counters). For several reasons, counters are relegated to the main store. This leaves only the floating-point numbers to be considered.

All operations on floating-point numbers are done by means of subroutines, and all of these use the single-length stores, TS13, TS14, TS15. They use DS20, DS21, and TS16 for purposes mentioned in the last chapter. Many of them use DS19. None of them use QS17 and QS18. Thus we have eight words of fast storage which are free to carry information through subroutines. We regard these as four floating-point stores which we name D, E, F, G. (They are, respectively, $17_{0,1}$; $17_{2,3}$; $18_{0,1}$; $18_{2,3}$.) Let us now see how these stores can be used to eliminate many of the magnetic transfers implied in the Alphacode program.

In instruction 4 there is clearly no need to fetch X5 from the drum. The number required is still in DS21 as a result of the previous instruction, and this is precisely where it is needed. Similarly for X8 in instruction 7. In instruction 3 there is no need to fetch X2 twice from the drum. It can be fetched into DS21, and then simply copied into DS20 by means of the instruction "21–20(2)."

The subroutines for all the functions in this particular program have been written so as not to disturb the contents of DS20. It follows that X7 in instruction 8 and X1 in instruction 10 need not be fetched from the magnetic drum since they are already in DS20. In instruction 5, X6 is in DS21 as a result of the previous instruction.

Already six drum references have been eliminated merely by considering what each instruction can use of the information left by its predecessor, and none of D, E, F, G have yet been touched. To make use of these, we look through the program from the bottom. We first note that the result of instruction 8 is wanted in instruction 10. Therefore, we put it in D, and alter instruction 10, so that the address "X10" now reads "D." The result of 7 is wanted in 9; it cannot be carried in D since D is accounted for over part of the way, but E can be used. By the time we consider the result of 6, wanted in 8, D is free again. We carry on up the program in this way; this process eliminates five more drum references, at the cost of two fast-store references each.

We then note that X2 is wanted in 6 as well as in 3; once it has been fetched into DS21, from the drum, for instruction 3 it can be put into D and fetched from there for instruction 6. Similarly, X1 can be carried in F from 2 to 9. Finally, we note that there is no point whatever in writing the results of instructions 2 to 8, inclusive, on the drum, since all reference to them elsewhere has been eliminated. Therefore we do not write them, and so the program, partially translated, now looks like this:

```
1.          3   DATA     X1
2.  E   = X1, F  ×       X3
3.  .   = X2, D  ×       DS21
4.  .   =  .     —       E
5.  E   =        ROOT    DS21
6.  D   = 0      —       D
7.  E   =  .     —       E
8.  D   = D      +       .
9.  X11 = E      ÷       F
10. X12 = D      ÷       .
11.         2   RESULTS  X11
            FINISH
```

We can go no further with fast-store allocation in this example, since the DATA and RESULTS subroutines do not refer directly to the fast stores, as do the others. (They are designed to cope with strings of numbers of any length, and so refer always to the drum.) Even so, remembering that each result for the drum represents two magnetic transfers, we have only 7 magnetic transfers in instructions 2 to 10 as against the 34 which would result from a crude transliteration.

The processes outlined above are all amenable to programming, and are all incorporated in the Alphacode Translator. The example used here was particularly simple; the Translator must cope with complicated programs containing loops, branches, modified addresses and so on. These features make for difficulties in storage allocation, only some of which can be mentioned in this paper. The part of the Translator for allocating the fast stores has about 5,000 instructions; only a quarter or so of these would be required if all programs were as simple as the example.

In an earlier paper on this subject (Duncan and Hawkins, 1959) these processes were described with a slightly different emphasis, and without specific mention of DEUCE. Block diagrams of parts of the system were also given.

6. Flow-Diagram Analysis

The effect of a storage allocation process, such as that described in the previous section, on any instruction is a function not only of the instruction itself but of the whole context in which it is considered. In the example, the third address in instruction 10 was affected by the second address in instruction 2. Here the whole program was considered as a single unit; this was because its flow diagram was trivially simple—merely one "block."

Fig. 1(*a*) shows a simple type of non-trivial flow-diagram, which is very common. (In these diagrams we assume that each block stands for a simple string of instructions with a unique entry point and no branches within itself; it may or may not end with a jump or discrimination instruction.) From the shape of the flow-diagram alone it is evident that II is obeyed at least as often as I and III (i.e. once if I–II–III is the whole program); the probability is that it is obeyed much more often. Hence II deserves priority of treatment over I and III. In practice this means that I and III should not be allowed to prejudice the storage allocation in II, and so the allocation is done for each of I, II, III independently of the others. This ensures that II starts with a clean sheet. We now have the problem of reconciling the differences in the contents of the store between the end of one section and the beginning of its successor. Clearly, priority in this should be given to the return path from the end of II. A practical example of the use of this scheme is where II calculates a quantity by iteration. Suppose it is calculating y, where $y = y_n$

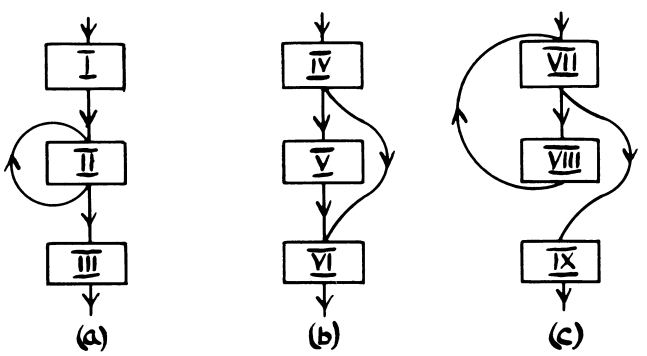

Fig. 1.—A simple flow diagram

and $y_1, y_2, \ldots y_n$ are successive values calculated by II such that $y_r = f(y_{r-1})$ and $|y_n - y_{n-1}| \leqslant \varepsilon$. In this sort of situation, it is clear that only the final result, $y_n = y$, is significant, and there is no point in writing the intermediate values on the drum. Thus, around the loop tie, the result will not be written; along the tie II–III it will be written if necessary. Also, along the tie I–II the initial value will be (probably) read from the drum, but intermediate values, carried along the return arm of the loop, will be taken from a fast store. This situation would not be achieved if the whole program I–II–III were considered at the same time from the beginning, and the loop tie put in afterwards.

In Fig. 1(*b*), however, the situation is somewhat different. If anything, V has lower priority than IV and VI, but there is little to be gained here by taking the sections separately. We take the three sections as a whole for storage allocation, and then insert instructions, on the tie IV–VI, to adjust the contents of the faster stores to the assumptions made by VI as a result of the overall allocation.

This leads us to the idea of "stages" in storage allocation. In Fig. 1(*a*) there are three stages; in 1(*b*) there is one, while, in 1(*c*), VII and VIII form one stage and IX another. There is an arbitrary division in Fig. 2, where the third section forms a stage separate from the others. This is inevitable when stage division becomes systematized. The division of the program into sections, the compiling of lists of ties between sections, and the grouping of sections into stages accounts for about 1,000 instructions in the Translator.

The system seems to cope adequately with any shape of flow diagram. One of the ugliest specimens dealt with so far is shown in Fig. 3. Stage division is indicated by broken lines. This program, which was one of the first to be translated, was produced by a physicist who had learnt Alphacode *ad hoc*, and this was his first program. This emphasizes the point made earlier, that the Translator is intended for programs produced by casual users; this means that it must be prepared to deal with all kinds of shapes of program. No restrictions have been put on programmers in this respect.

Fig. 4 shows a more typical flow diagram, also divided into stages. Here the stage division is much more reasonable.

7. The use of the Main Store

The problems of main store utilization are potentially greater than those for the fast store, since there are several types of quantity which must be accommodated. Let us consider these in turn.

(i) Floating-point quantities (numbers in X-stores). The allocation process for the fast stores is not capable of eliminating all the X-addresses, although in many cases it absorbs 75% or so of them. Those that remain should be considered for a place in the main store.

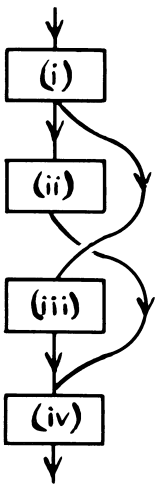

Fig. 2.
Stage Division

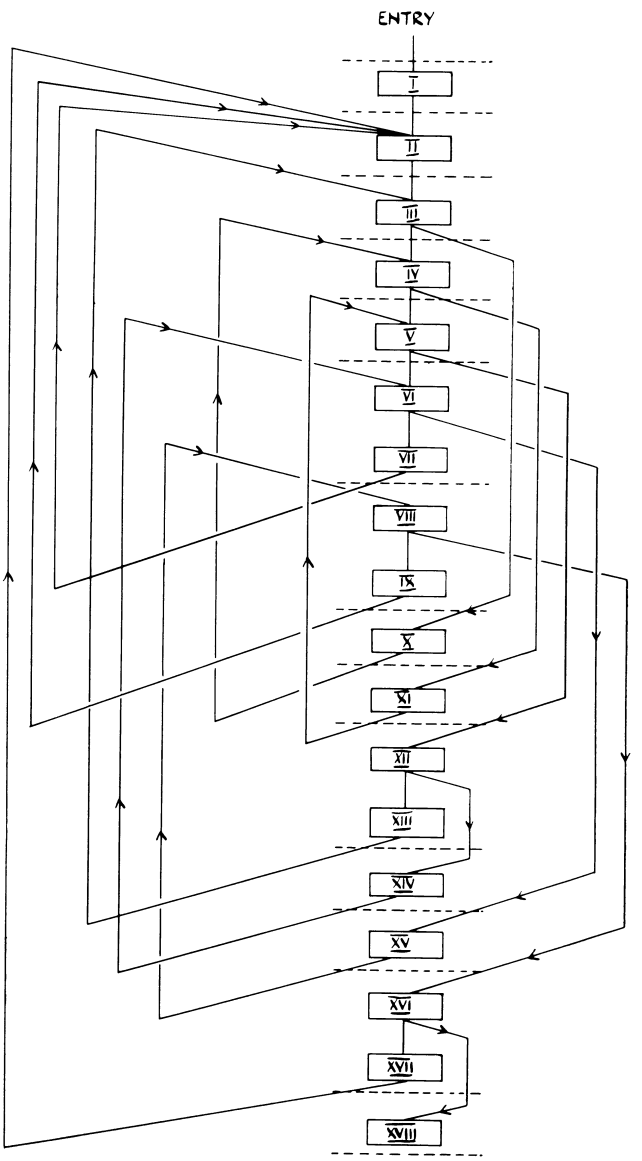

Fig. 3.—A "difficult" flow diagram

(ii) Fixed-point quantities. These are counters, specified by N-addresses.
(iii) Subroutines, corresponding to Alphacode functions.
(iv) Main program. The DEUCE equivalents of translated Alphacode instructions, to be generated by the Translator.

To simplify the problems, these types have been allocated to particular delay lines, as follows:

DL1 to DL4 are reserved for standard subroutines (floating-point arithmetic, fetch and store routines for modified addresses) and certain useful constants.
DL5 to DL7 are used primarily for other subroutines, as required. Changes are organized in a manner to be described later.
DL8 is reserved for the main program. The program is allowed to extend into DL5–DL7 if these happen to be free of subroutines.
DL9, DL10 are used for floating-point numbers.
DL11 is the buffer for the magnetic drum, although if there is a lull in magnetic transfers, it is used like DL9 and DL10.
DL12 is used for the 32 most frequently used counters (N-addresses).
This division is admittedly arbitrary, but it seems to produce good results with most programs and is in accord with the practice of most programmers.

(i) *Floating-point Quantities*

The object is to replace by addresses in DL9 and DL10 as many as possible of the drum addresses remaining after fast-store allocation.

Each delay line holds 32 words, that is 16 floating-point numbers, and this is the same as one drum track. The X-stores are consecutive word pairs on the drum. Thus X1 to X16 are in the first track, X17 to X32 in the next, and so on. If there are several references in one stage to addresses in the same track, the thing to do is to read this track into DL9 or DL10 before the first reference is made. The drum references become delay-line references; only if a result is put into the delay line does it have to be written back on to the drum. In practice, several tracks will be competing for places in the two delay lines, so the allocation process has to decide which tracks are to have priority.

Allocation is done a stage at a time. For each stage a table is made showing, for each track concerned, a list of the instructions in which it is mentioned, and whether for operand or result. From this a "priority table" is made, giving the limits of the span of instructions over which the track is required, and a priority number. This number is the number of magnetic transfers which would be saved if the track were treated as outlined in the last paragraph. It is the number of times the track is wanted for operands, *plus* double the number of times for results, *less* one (for the first "read" transfer) *less* one more (the final "write") if the number of results is non-zero.

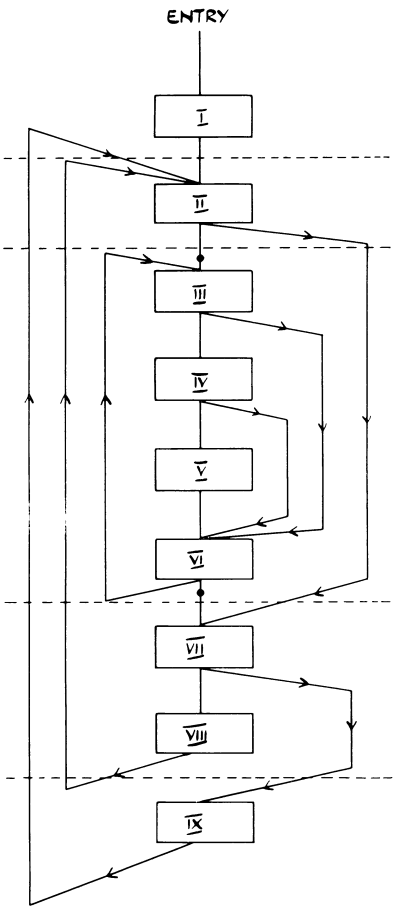

Fig. 4.—A more typical flow diagram

The track with highest priority is then considered. Since this is the first time, it will probably be accommodated, although it is possible that the stage contains an odd instruction that uses the delay lines for itself (as, for example, "Solve Differential Equations"). If, however, the attempt is successful, a delay line is marked as occupied over the appropriate span, and the track is removed from the priority table. The table is then again searched for the remaining track of highest priority, and an attempt made to accommodate this. Whenever a span cannot be fitted in directly, instructions obeyed while the delay lines are in use are removed from the table, and priority for the track re-assessed.

The process is thus iterative in nature. If both delay lines are free at a particular step in the process, the choice falls on that with the shorter free path. This is probably clearer in a diagram. Fig. 5 shows that three allocations have been made to DL9, and two to DL10. The third column shows a span which is being considered. It can go into either delay line, but DL10 is chosen, so as to leave the longer free span in DL9 for later use.

The final part of the data delay-line allocation is a set of runs through the program, changing drum addresses to main store addresses where appropriate, inserting markers to show where tracks must be read and written to and from the delay lines, and making insertions in the flow diagram description.

Reading and writing markers are so placed as to ensure that a track is read as early as possible, and written as late as possible, in order to take advantage of the carry of delay-line information between stages.

A difficulty common to fast-store allocation and to this process has not yet been mentioned. This is the presence of modified addresses. Unlike some other schemes, Alphacode does not require the user to specify ranges of modifiers, so the Translator must assume they are infinite. The only restriction is that they must be positive.

The effect of a modified address, as far as the Translator is concerned, is to introduce an ambiguity. The address X15 modified may mean anything from X15 upwards, though X1 to X14 preserve their unique meaning. It follows that a fast store containing X20, say, may no longer contain the latest version of X20 after an instruction in which a result is put into "X15 modified" (i.e. if the modifier happens to have the value 5). Modified result addresses are in fact partial barriers to the flow of information through the faster stores. Modified operand addresses have far less effect, although clearly they must be distinguished from unmodified addresses.

(ii) *Fixed-point Quantities*

The 32 most frequently used counters (N-addresses) are allocated to DL12. Any others—up to 63 are allowed in Alphacode—are given addresses in a drum track. A simple run is made through the whole program, changing the addresses, and a table is made, to be punched out at the end, relating these new addresses with the programmer's N-addresses.

(iii) *Subroutines corresponding to Alphacode Functions*

The subroutines for floating-point arithmetic (addition, subtraction, multiplication, division, fixed-to-floating and floating-to-fixed conversion) are required very frequently in almost all programs. Therefore they have a permanent place in the main store, in DL1–4, together with a "magnetics fetch and store" routine, for reading and writing quantities specified by modified addresses, and for use in other subroutines such as "Sum Series," which operate on strings of data.

Other subroutines are given places in DL5, DL6, DL7. It has been pointed out that instructions must be in the main store when they are obeyed. There is thus no question of a selection of subroutines being allocated places in the main store—all must be accommodated. This makes the problem of allocation radically different from that for numerical quantities, and in fact much simpler.

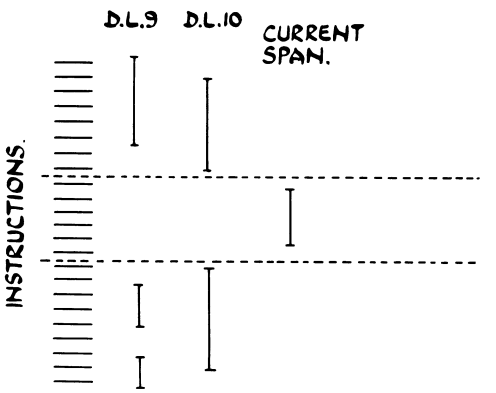

Fig. 5.—Allocation of delay lines

Allocation is first done section by section. It is at first assumed that copies of all subroutines exist on the drum, and that transfers to the main store are to be made for each entry to every subroutine. Then unnecessary transfers are eliminated. If a sequence of cosines is to be calculated, for example, all transfers of the cosine subroutine are eliminated except the first. Since "sine" and "cosine" are substantially the same subroutine, there is no need to read one if the other is known to be in the main store. Again, if "log" is in the main store, and "$\sinh^{-1}$" is required, only one extra track is required, since "$\sinh^{-1}$" implies "log." Conversely, if "$\sinh^{-1}$" is down and "log" is required, no transfers are necessary.

The first transfers into each of the three delay lines are moved right up to the beginning of the section, and note is made of the contents of each delay line at the end of each section. When the sections are joined together into stages, this may enable further transfers to be eliminated, and, where loops are involved, it may allow subroutines to be fetched once for the first entry into the loop and then merely assumed for subsequent entries. When combined with storage allocation for numbers, this process often enables quite big loops—perhaps 30 Alphacode instructions—to be contained entirely within the fast and main stores.

There are a few subroutines too big to be contained in three delay lines. These are allowed to displace part of the standard four, as necessary. The standard delay lines are restored automatically when they are next required, or, in any case, at the end of the section.

(iv) *Main Program*

By "main program" is meant the instructions linking together the subroutines, fetching subroutines, and fetching and storing numbers, and so on. One delay line, DL8, is always reserved for this. In addition, if a stage does not require any of DL5, DL6, DL7, that delay line, or delay lines, becomes available, with DL8 for main program, and is used if the stage requires more than 32 instructions.

8. Generation of DEUCE Instructions

During the whole of the processes described so far, the basic unit of the operand program is the Alphacode instruction. The first operation of the Translator is to read the whole operand program on to the drum. The instructions, originally two words each, are rearranged and given four words (128 digits) each, the extra space allowing for markers and extra addresses to be put in during the course of translation. Up to 512 Alphacode instructions are allowed, and so 64 tracks are allocated. (This is the same limit as for the Interpreter; facilities exist for linking programs together.) For each "section" (see previous section of this paper) a set of "section parameters" is set up; this is essentially a description of the section in terms of the flow of information. Sixteen words are allowed for each section; 128 sections are allowed, and so 64 further tracks are accounted for. Another 16 tracks are required for various tables and indexes, leaving 112 tracks for the Translator itself and for working space. The Translator operates by being read in one part at a time; each part on completion calls the next from the program tape or the card reader.

By the time the processes for flow diagram analysis, fast-store allocation, and main-store allocation for numbers and subroutines have been applied (in that order), the instructions have ceased to bear much resemblance to their original forms. The significance of a four-word group is now not so much one Alphacode instruction as a string of DEUCE instructions. At this stage, therefore, it is convenient to generate the DEUCE instructions explicitly. The part of the Translator for this has about 3,000 instructions. About 2,000 of these are concerned with the actual instruction generation, the remainder with controlling a referencing system whereby the first instruction of each section and tie section is labelled uniquely, and references to these labels are given at the end of each section to indicate the continuation of the flow diagram.

Owing to the lack of space in the computer, the generated instructions are not stored, but are either written on magnetic tape or punched out, with check sums in either case. Each stage has an indication of what delay lines are available for coding (DL8 always, sometimes some of DL5, 6, 7), and the whole program is preceded by an indication of which tracks have been reserved for subroutines. This information is required for the coding process which follows.

At this point, the translation is about two-thirds complete. The output tape or pack is used as input for the next part, which can be operated on the machine independently of the first.

9. Main Program Organization

The partially translated program which is given to the second part of the Translator still gives opportunity for improvement. Improvements are made in two ways. The first is in merging together as far as possible branches

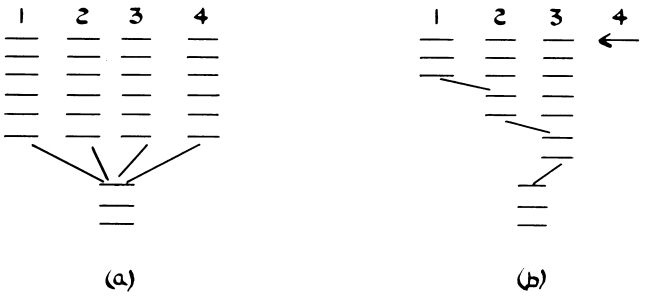

Fig. 6.—Merging of branches

of the flow diagram which converge to the same point. For example, consider the case where four branches converge, Fig. 6(*a*).

In practice we might have the last four instructions of branch 1 identical with those of branch 2; the last two of branch 2 identical with those of branch 3; and the last seven of branch 3 identical with those of branch 4. In this case Fig. 6(*b*) shows all the instructions that are strictly necessary; thirteen redundant instructions of 6(*a*) have been eliminated. (It should be pointed out, perhaps, that unconditional jump instructions are not necessary in DEUCE, since, to make optimum coding possible, each instruction specifies its successor.)

This process does not directly cause any increase in speed, for along each path the same amount of work as before has to be done. However, the saving of instructions may very well enable a loop, for example, to be coded within the main program space in the main store, thus doing away with the need to have program changes from the drum during the execution of the loop. This indirect saving of magnetic transfers may be quite significant in effect.

The second way of improving the program is exclusively concerned with magnetic transfers. It is usually the case that a high proportion of the magnetic transfers remaining in the program are redundant, because they merely repeat what has been done by the preceding magnetic transfer. A simple example of this is of a track of numerical data which has failed, because of insufficient priority, to find a place in DL9 or DL10. Instructions will have been generated to read this track for each time it is wanted. But (if there are no other magnetic transfers to confuse the issue) only the first is necessary; the others are eliminated, and so DL11, the drum buffer, behaves like DL9 and DL10. This elimination of redundant magnetic transfers always saves time as well as instructions.

Concurrently with the latter process, the program is divided into blocks for detailed coding, and room for these blocks is reserved on the magnetic drum. The simplest case is of a stage with less than a trackful of instructions. This becomes one block, and is coded in one delay line, even though more may be free. Similarly, a stage which can be coded within the delay lines available is one block; it will not take more of these delay lines than are necessary. If a stage has too many

instructions for the delay lines, it must be split into two or more manageable blocks, and instructions generated to make each block replace its predecessor in the delay lines. This involves some rather tricky logic, particularly since elimination of redundant transfers is also taking place. In all cases, spaces are reserved on the drum for each block. The whole process takes about 2,000 instructions.

The next task is detailed coding. The basic problem is that of coding a block of instructions within four or less delay lines. The subject of DEUCE coding has been dealt with elsewhere (e.g. Haley, 1956) so there is no need to outline the action of the Translator's coding routine. In generating the instructions of the translated program most of the sophisticated "gimmicks" beloved of DEUCE programmers have been thoughtfully avoided, so that the coding routine is much simpler than might be supposed (about 1,500 instructions). The next instruction space chosen is always the first available, after the completion of the transfer specified by the current instruction.

After coding, the Translator is able to punch out, in binary form, the flow diagram of the translated program. This punching out is optional, and can be suppressed. Conversion to alphanumerical form, and tabulation can be done if required as independent operations.

Finally, the Translator punches out the complete translated program as a pack of cards, which can be used entirely by itself. It contains

(i) DEUCE standard initial routines.
(ii) A copy of each subroutine required, selected and reproduced from the library contained within the Translator pack.
(iii) The coded main program.

As stated before, at the beginning of this paper, data forms for the translated program are identical with those for ordinary Alphacode programs, so no conversion of data is necessary.

10. Conclusion

It is extremely difficult to obtain a fair comparison between the performances of a translated program and a hand-written program for the same job. It is intended to obtain some direct comparisons by writing Alphacode versions of existing DEUCE programs. In the meanwhile, examination of translated programs seems to suggest that they operate at something like two-thirds the speed of the programs that would normally be produced by hand.

The allocation of storage is of course done more systematically by the Translator than by the human programmer, but the latter has the advantage of *ad hoc* knowledge of the problem, whereas the Translator, being designed for a wide range of problems, suffers by being forced to "play safe." In particular, the Translator, working only from the Alphacode program, has no information as to the dimensions of arrays, and so is prevented from accommodating them in the faster stores; it knows nothing about relative frequencies of alternative paths in the flow-diagram, and so has to make "intelligent guesses" as to priorities in storage allocation; and finally, it suffers from the stage division.

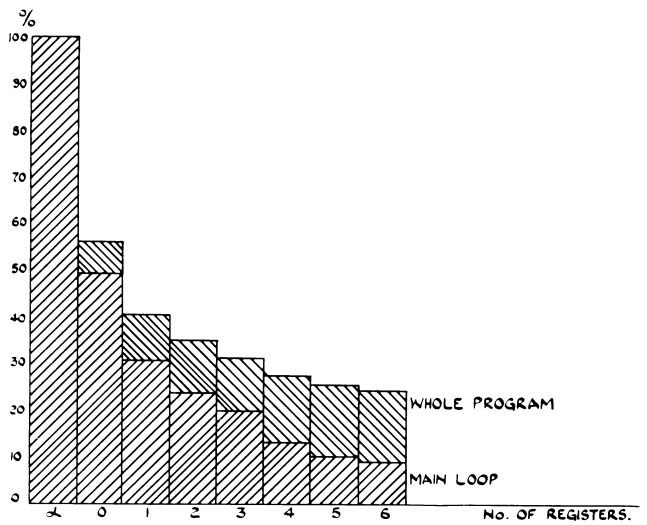

Fig. 7.—Percentage of slow store transfers remaining after the use of fast registers

For practical reasons the stages are treated almost independently. Ideally, the program should be treated as a whole throughout, but this would mean a much more complicated, and slower, Translator.

In the section describing fast-store allocation, it was stated that about 75% of the number addresses were changed into fast store addresses. This was with four ordinary fast registers, and two fully addressable arithmetical registers (DS20, DS21). The system briefly described there has also been applied to a few programs assuming up to six ordinary fast registers. For each of these programs a histogram was drawn. They were quite consistent with each other, and one is shown in Fig. 7. The percentage of slow-store transfers remaining after allocation is plotted against the number of fast registers available. In the first column (α) there are no addressable fast stores; in the second column (0) the two arithmetical registers (called "accumulator" and "multiplier register" in some computers) are addressable; in the next columns there are, in addition, from one to six fast registers.

Two plots are superimposed on the diagram; the upper one is for the program as a whole, the lower (better) one for its main calculating loop. This illustrates the effect of the priority system described above.

11. Publication

It is clearly impossible to describe the Translator fully in a short paper. However, it is being made freely available to all DEUCE users as part of the normal DEUCE library service, and the report issued with it,

which will be in several volumes and contain several hundred diagrams, will contain a complete description, detailed block diagrams, and coding.

12. Acknowledgements

Much of the initial planning and investigation of possible procedures, as well as the programming of considerable parts of the Translator, is due to Mr. E. N. Hawkins. Part of the programming of the storage allocation process was done by Mr. W. P. Gillott.

The paper is published by permission of The English Electric Co. Ltd.

13. References

1. BRIGHAM, R. C., and BELL, C. G. (1959). "A Translation Routine for the DEUCE Computer," *The Computer Journal*, Vol. 2, p. 76.
2. BROOKER, R. A. (1960). "Some Techniques for Dealing with Two-level Storage," *The Computer Journal*, Vol. 2, p. 189.
3. DENISON, S. J. M. (1960). "Further DEUCE Interpretative Programmes and Some Translating Programmes," *Annual Review in Automatic Programming*, Vol. 1.
4. DUNCAN, F. G., and HAWKINS, E. N. (1959). "Pseudo-Code Translation on Multi-level Storage Machines," *Proceedings of the International Conference on Information Processing, Paris*, 1959, p. 144.
5. THE ENGLISH ELECTRIC CO. LTD. (1959), "DEUCE Alphacode Manual."
6. GILL, S. (1959). "Current Theory and Practice of Automatic Programming," *The Computer Journal*, Vol. 2, p. 110.
7. HALEY, A. C. D. (1956). "DEUCE, a High-speed General Purpose Computer," *Proc. Inst. Elect. Eng.*, Vol. 103, Part B, Supplement 2, p. 165.
8. ROBINSON, C. (1959). "Automatic Programming on DEUCE," *Annual Review in Automatic Programming*, Vol. 1, p. 45.
9. ROBINSON, C. (1959). "DEUCE Interpretive Programs," *The Computer Journal*, Vol. 1, p. 172.
10. WILKINSON, J. H. (1955). "An Assessment of the System of Optimum Coding used on the Pilot A.C.E. at the N.P.L.," *Phil. Trans. Roy. Soc. A*, Vol. 248, p. 253.

Book Review

Mathematical Methods and Theory in Games, Programming, and Economics, by SAMUEL KARLIN, 1959; 2 volumes, 819 pages. (London: Pergamon Press, 75s. 0d. each volume.)

The present review is written under the assumption that readers of this Journal are mainly interested in contributions to computing theory and practice. It would be most valuable to have a book on those subjects in the title, dealing with their computing aspects, but these two volumes do not constitute such a text. This should not detract from their value, which will certainly be praised in other reviews; the author did not set out to write a book on computing, and he cannot be blamed for it.

The two volumes deal, on an advanced level of mathematical abstraction, with the theory of matrix games, of linear and non-linear programming, and of mathematical economics. Volume 2 is entirely devoted to infinite games, i.e. games where at least one of the players can choose from an infinity of strategies. The wide scope of this theory is well illustrated.

Chapter 6 is entitled Computational Methods for Linear Programming and Game Theory. It presents the Simplex Method with a brief illustration, some slight modifications of it, a computation of network flow—without making clear the position of this algorithm in relation to other methods mentioned—and finally a differential-equation method for determining the value of a game. No numerical methods for solving non-linear problems are exhibited, although such methods exist.

On its own advanced level the book is excellent, and so is the contribution of the publishers, concerning type, display of formulae, paper, etc. The two volumes are worthy of a place in any library of modern mathematical texts.

S. VAJDA.