THE KDF.9 COMPUTER SYSTEM

A. C. D. Haley
English Electric Co., Ltd.,
Kidsgrove,
Stoke-on-Trent, England

SUMMARY

The English Electric KDF.9 computing system has a number of unusual features whose origins are to be found in certain decisions reached at an early stage in the planning of the system. At this time (1958-59) simplified and automatic programming procedures were becoming established as desirable for all programming purposes, whereas previously they had been regarded as appropriate only to rapid programming of "one-off" problems because of the drastic reductions of machine efficiency which seemed inevitable.

Many early interpretive programming schemes aimed to provide an external three-address language, and for a time it appeared that a machine with this type of internal coding approached the ideal. Increasing interest in translating programs, particular for problem languages such as ALGOL and FORTRAN, showed the fallacy of this assumption. It became evident that efficient translation could only be achieved on a computer whose internal structure is adapted to handle lengthy algebraic formulae rather than the artificially divided segments of a three address machine.

The solution to the difficulty was found in the use of a "nesting store" system of working registers. This consists of a number of associated storage positions forming a magazine in which information is stored on a "last in, first out" basis. It is shown that this basic idea leads to development of a computer having an order code near to the ideal for evaluation of problems expressible in algebraic form.

A number of other significant advantages arise directly from the nesting store principle, chief among them being a striking reduction in the program storage space required. This is due to elimination of unnecessary references to main store addresses and to the implicit addressing of operands in the nesting store itself. Many instructions are therefore concerned with specifying only a function, requiring many fewer bits than those instructions involving a main store address. Instructions are therefore of variable length to suit the information content necessary and on average three instructions may occupy a single machine word (48 bits). This again reduces the number of main store references, allowing the use of a store of modest speed while still allowing adequate time for simultaneous operation of a number of peripheral devices on an autonomous main store interrupt basis.

Fast parallel arithmetic facilities are associated with the nesting store, both fixed and floating-point operations being provided. A further nesting store system facilitates the use of subroutines, and a third set of special stores is associated with a particularly comprehensive set of instruction modification and counting procedures.

Operation of the machine is normally under the control of a "Director" program. A number of different Directors cover a variety of operating conditions. Thus a simple version is used when only a single program is to be executed and more sophisticated versions
may be used, for example, to control pseudo-off-line transcription operations in parallel with a main program, operation of several programs simultaneously on a priority basis, etc.

INTRODUCTION

For almost a decade after the first computers were put into service, developments in system specifications were almost exclusively a by-product of local engineering progress. No radical changes took place in machine structure, except insofar as engineering changes led directly to operational ones. Thus the emergence of the ferrite core store as the most reliable and economic rapid access store for any significant quantity of information led to the abandonment, now virtually complete, of the various types of delay line store. In consequence, "optimum programming" is now used only in certain systems which exchange speed for economy and use a magnetic drum for the working store.

The majority of systems continue to be basically simple single address order code machines, in which most orders implicitly specify an arithmetic register as one participant in every operation. It was the subsequent proliferation of transfers between this register and the main store, purely to allow its use for intermediate arithmetic operations on instructions, which led to the introduction at Manchester University of the "B-tube." This in turn has been extended and elaborated to provide the automatic instruction modification and/or counting features which are now universal.

A further reduction in housekeeping operations, with a consequent increase in speed, can be obtained by providing not a single register associated with the arithmetic unit, but a number of registers. Sometimes all facilities are available on all registers, and in other machines the facilities are divided.

Changes and elaborations such as these have, of course, had the primary aim of increasing the effective speed of the machine. The penalty to be paid is the complexity of programming. The increased quantity of hardware required is, of course, more than compensated by increased computing power.

There is no corresponding compensation for the increased programming costs, particularly for problems of infrequent occurrence. This factor was the provocation for much of the early work on simplified programming schemes. Major programs and those to be used repeatedly were written in machine code, but the remainder were written in problem-oriented languages either obeyed interpretively (as if a set of subroutines) or translated into the necessary machine code program by highly sophisticated translator routines. Typical of the former approach are English Electric "Alphacode" [1] and Ferranti/Manchester University "Autocode" [2, 3] and of the latter method I.B.M. "Fortran" [4] and English Electric "Alphacode Translator" [5, 6]. The penalties of using these schemes are again different in nature. Interpretive routines lead to inefficient use of machine time during every run, factors of 10 to 100 covering most of the systems in common use. The translator routines, in contrast, may ideally produce a 100% efficient program, although a loss of speed by a factor of 1-1/2 to 5 is more usual. The translation operation, performed only once, may however occupy long periods and may swamp the subsequent gain over the interpretive method for a rarely used program.

In the late nineteen-fifties it was evident that a successful new general-purpose system should ideally have two programming features:

(a) It should have an order code designed to allow easy efficient coding in machine language after a minimum training period;

(b) The language should be such as to allow the preparation of a number of rapid translators (for many categories of work) from problem-oriented languages into efficient machine programs.

Studies based on these fundamental requirements culminated in the machine organization selected for KDF.9.

Choice of an Order Code

Many interpretive schemes used up to this time had been of three-address type, where the typical instruction is of the form

\[ C = A \text{ (func. on) } B, \]

and A, B, C are main store addresses. Experience had shown that this type of code was well adapted for design calculations and scientific usage, especially if a wide range of fixed and floating point functions and well-chosen loop counting and instruction modifying facilities were made available.
At a time when potential arithmetic speeds were overtaking storage access rates the use of a three-address machine code had the drawback of requiring four main store references for each operation (including extraction of an instruction), and with the constant demand for stores of 32,000 or more words the necessary instruction length approaches 60 bits when provision for a large number of functions and modifiers is made.

A single address system, on the other hand, requires greater care in programming and a multi-accumulator machine is perhaps worse still from this standpoint.

None of these conventional structures is particularly well suited to preparation of efficient translation routines, and a study was therefore made of a special form of multi-accumulator system using automatic allocation of addresses for the working registers. This depends on presentation of data to the system in a form closely analogous to the "reverse Polish" notation. (This notation merely involves writing the sign of any arithmetic operation after, rather than before, the two operands. Thus $a + b$ is written $ab+$

$$a \rightarrow b \text{ is written } ab+, \text{ etc.}$$

Fundamentally the procedure is to arrange the machine structure in such a way as to allow operations to be performed in a natural order. Thus, in carrying out the operations involved in calculating $E = A.B + C.D$, the natural sequence is

- obtain $A$;
- obtain $B$;
- multiply & retain $A.B$;
- obtain $C$;
- obtain $D$;
- multiply & retain $C.D$;
- add;
- store $E$.

Given a suitable "user code" (i.e., a convenient form in which to prepare a program of instructions), it was evident that such a machine structure, if attainable at reasonable cost, would meet requirement (a) above. The belief in its appropriateness for simplicity of coding was coincidentally supported by the appearance of an interpretive scheme [7], producing a similar problem-language, written for DEUCE. This proposal, however, attracted little attention, mainly because the multi-level storage system of DEUCE prevented attainment of a reasonable working efficiency.

The potential virtues of this type of problem language were again supported by a subsequent proposal to use a reverse Polish notation as a computer common language (APT). For a number of reasons this proposal was not adopted, but two of the reasons advanced in its favor are worthy of note in the present context. They were:

(a) That the language is one in which problems can be formulated with little effort after a short period of training;

(b) That the process of automatic or manual translation from any conventional problem language to most of the existing machine languages requires an effective expression in reverse Polish notation as an intermediate step in the procedure. Using such a machine code therefore eliminates a substantial part of the process.

The Nesting Store

The evaluation of a formula above can be seen to consist of successive operations either of fetching new operands (or storing results) or of performing operations on these most recently named. An intermediate result is immediately reused (as in the seventh step) or temporarily pushed "below the surface," as when the product $A.B$ is superseded by the fetching from store of $C$.

A mechanical analogue of such a system is shown in Figure 1. It consists of a magazine, spring loaded to maintain the contents at the top, with only one point of entry and exit. Objects stored can therefore only enter and leave on a "last in, first out" basis.

The electronic equivalent is shown in Figure 2. There are $n$ separate registers, each capable of accommodating one machine word, and corresponding bit positions of each register are connected so that the assembly can also be treated as a number of $n$-bit reversible shifting registers. Information can be
transferred to the top register N1 from a conventional main store buffer register (a parallel machine is assumed), and simultaneously with any such transfer a shift pulse causes a downward shift of any existing information. Thus the new word appears in N1, the previous content of N1 in N2, etc. This process is called "nesting down." Reversal of the process causes "nesting up" with transfer of the content of N1 towards the main store. "Nesting up" and "nesting down" can occur in any sequence, provided, of course, that not more than n words are in the store at any time.

Associated with the top two registers is a "mill" or arithmetic unit [8]. Serial transfers to and from the mill are shown for clarity, but in a fast machine parallel working is again used.

To allow the evaluation of a formula such as the simple one of section 2, the mill must be made capable of the two arithmetic operations needed. One of these is addition, and at this point it must be noted that in general no operand is considered as necessary after being used by the mill. The operation "add" therefore uses the words in N1 and N2, and places the sum back in N1. Since N2 is now unoccupied, "nesting up" occurs, the content of N3 moving to N2, etc.

The natural effect of most of the conventional arithmetic operations can now be visualized. Thus, "multiply" produces a double-length product in N1 and N2 (the more significant half in N1 for obvious reasons). No nesting is involved in this operation, but the more elaborate "multiply and round" produces a single length result in N1 and therefore requires nesting up.

Two important points emerge at this stage. The first is that an operation sequence written as

YA, YB, x, YC, YD, x, +, = YE

evaluates the formula given earlier (YA is to be interpreted as "fetch from the main store address containing A into N1," and = YE as a converse operation). This sequence is the one involving a reduction to a minimum number of main store operations.

The second important point is that arithmetic operations have implied addresses, so that only the function need be specified. On the other hand, all instructions referring to a main store address require many bits for this purpose unless flexibility and convenience are sacrificed by addressing relative to a local datum.

Variable Length Instructions

It is clearly advantageous to economize in instruction storage space, and hence, for reasons stated above, to allow instructions to vary in length according to their function and the additional information they require. Obviously any instruction must carry within itself a definition of the number of bits included in it. Analysis shows that complete variability is unprofitable (as well as complicating the hardware), since five bits would be needed to specify the length. Three possible lengths of 8, 16 and 24 bits are therefore made available, including in each case bits to designate length. In connection with instruction length, a unit of eight bits is referred to as a "syllable." Most arithmetic operations are therefore one-syllable instructions; memory fetch and store operations together with jumps are three-syllable, while two-syllable instructions include a number requiring parameters of less length than memory addresses (shift instructions, input/output instructions, etc.).

The word length of the computer is 48 bits, and this is the smallest unit in which information can be extracted from the main store. Instructions are stored continuously and obeyed serially; i.e., the store area concerned is regarded as a continuous series of eight-bit syllables, rather than of 48-bit words, and two or three-syllable instructions may overlap from one word to the next. Associated with the main control there is
therefore a two-word register. This at any time holds the word containing the current instruction together with that from the next higher main store position (if the current instruction overlaps a word boundary both words are of course in use). As soon as all syllables in one word have been used, this word is replaced at the first available main store cycle by a further word in sequence from the main store.

Because of the economy in instruction storage space achieved by these means, it is frequently possible to contain important inner program loops in two words of instructions. Provision is made to mark such loops by a special jump instruction, whose effect is to inhibit the extraction of a new instruction word until the condition for leaving the loop is satisfied. This saves two main store cycles (12 microseconds) for extracting instruction words on every circuit of the loop, and incidentally saves a syllable since again no main store address needs specifying completely.

Some additional complexity arises due to the separation of control into two parts, one associated primarily with arithmetic operations (known as "Mill Control") and the other, which runs normally at least one instruction in advance, controlling most other internal operations and in particular controlling access to the main store (this is called "Main Control"). The object of this is, of course, to increase effective speed by allowing, for example, a new instruction word to be extracted while an arithmetic operation proceeds in the nesting store. Such overlaps are completely automatic and no special actions are required of the programmer.

Further Consideration of Nesting Store

At this stage it is convenient to examine the nesting store and the consequence of its use in a little more detail.

It should first be stated that the representation of Figure 2 while possible is uneconomic if more than a two or three words of storage capacity are required. Examination of a large number of programmes shows that only occasionally is storage for more than about eight words needed in the evaluation of an expression, and that a sixteen word limit to capacity will cause inconvenience only on very rare occasions.
addressing levels in the plane. Suppose, for instance, that the whole system is empty, and successive words are fed into N1 (by a series of main store fetch instructions). The transfer paths between N1, N2, N3 and the mill are gated as required by control.

The only complication arising here is that the transfer system does not allow N2, for example, to send to N3 in the same half microsecond period in which it receives from N1. The two buffer registers of the mill are therefore used, and the transfers are, in the first half microsecond, N1 to B1 and N2 to B2.

The write address counter is at zero and the read counter at 15 for a reason which will appear shortly. In the first half microsecond N3 is also allowed to set up bias currents in the vertical core lines where digits are to be written.

The next half microsecond sees the word in N3 written into the top core plane position by operation of the write drive, and the subsequent period covers the operations needed to complete the first fetch. These are main store to N1, B1 to N2, and B2 to N3. Simultaneously the read and write address counters are advanced by one to zero and one respectively.

A second and third fetch may now be executed in exactly the same way, but, as the system was assumed empty originally, blanks (as distinct from data zeroes) are written into levels 0, 1, 2 of the core plane, leaving the cores cleared. Only on the fourth and subsequent fetches does a genuine word appear in level 3.

Continuation of these operations will on the sixteenth occasion fill the lowest level of the core plane, and the write counter has now carried round to address zero while the read counter is at 15. The programme is interrupted if further entries are attempted (except under circumstances mentioned in a later section).

It will have been noted that the read counter is always one position behind the write counter, so that if a nesting down operation is followed by nesting up, the read drive is used and no correction is needed to the counter position in order to read out the last word inserted.

Note also that all operations are either to read out, leaving a clear storage position, or to write into an already clear position. The normal read/write cycle is therefore unnecessary and would be wasteful of time.

Special Nesting Store Instructions

The simple example given earlier illustrates the general nature of the instructions provided. It is soon discovered, however, that a few instructions are desirable which have no real counterpart in more conventional systems.

Straightforward evaluation of an expression in algebraic form will usually produce the desired result without special manipulation. Occasionally it will be found that two operands, for example prior to a division, are in reverse order. An instruction "REVERSE" has the effect of interchanging the contents of N1 and N2.

The instruction "DUPLICATE," which nests down and leaves a copy of N1 in both N1 and N2 has many uses. Followed by MULTIPLY it produces the square of N1, and it also allows an operand shortly to be lost in some other operation to be preserved for later use without requiring a main store reference.

Instructions ZERO and ERASE bring an all zero word into N1, nesting down, and erase N1, nesting up, respectively.

Many instructions are also available in double-length form. Thus double length addition (mnemonic +D) treats N1 and N2 as one 96 bit number, and N3, N4 as a second.
It produces a double-length sum in N1, N2, nesting up two places.

Single and double precision floating point representation is also permitted, the corresponding mnemonic forms for addition being +F and +DF.

With these instructions it is possible to introduce an example showing the power of the system and incidentally the speed and the economy in instruction storage space.

The example to be given is perhaps somewhat artificial, but it has been selected to illustrate not only the essential simplicity of evaluation of any expression from an algebraic statement, but to illustrate also that even in a sophisticated system there is scope for an occasional elegant twist. The formula to be evaluated is

\[ f = \frac{a(a^3 b^2 + 1)}{b + 2c^2 d^2} \]

where a, b, c, d are single length fixed point numbers stored at Ya, Yb, Yc, Yd, non-adjacent addresses in the main store.

The table shows the successive steps in the calculation, together with the contents after each step of the top few cells of the nesting store.

The following points should be noted:

(a) Again main store references are reduced to the minimum practicable.

(b) A count shows that only 30 syllables, or five instruction words, are used (there are no two-syllable instructions in this particular example).

(c) It is advantageous to evaluate the numerator in expanded form in this particular case.

(d) The use of DOUBLE DUPLICATE at step three neatly anticipates future requirements for the parameters. An automatically programmed version would fetch these parameters again or could perhaps use a less satisfactory temporary storage process to avoid this.

The complete evaluation on KDF.9 takes less than 170 microseconds.

It is interesting and instructive to compare this performance with that of a conventional one or three-address machine. The same actual arithmetic and main store speeds, and a single accumulator only for the one-address machine are assumed. It is also assumed that instructions are packed two to a word for the single address system and one to a word for the three address type.

The single-address programme then occupies eight words of storage and takes about 250 microseconds (increases of 50% in both factors). A three-address system uses 9 words of storage and takes 340 microseconds.

This example requires several operations of a "housekeeping" nature and the savings arising from use of a nesting store are less prominent than is frequently the case. On the other hand, it is also possible to find cases where there is little difference between the various systems.

The very poor relative speed of the three-address machine is accentuated because of the assumption of the same basic internal speeds as for KDF.9 (typically 6 microsecond store cycle, 15 microsecond multiplication). It is, however, true that these represent speeds attainable at corresponding levels of economy. A substantially faster store with 1 microsecond cycle time could be used, at a significant cost penalty, and would bring down the problem time to around 160 microseconds. There is, of course, no corresponding saving in storage space.

**Treatment of Sub-Routines**

It will have been observed that in the example above the condition of the nesting store at the end was identical with that at the start. A system of preparing sub-routines can be based on this fact. At any stage in a program there may be information in one or more cells of the nesting store. At this point the parameters required by the sub-routine must be planted by the main program (or of course by a lower order sub-routine). A note must also be made of the next instruction in the program for re-entry purposes. These two points will be considered separately.

Just as an instruction such as MULTIPLY expects to find operands in the top cells of the nesting store and terminates leaving the product in place of the operands (nesting as necessary), so a sub-routine can also be arranged to function. It then becomes in effect a machine instruction in that it does not influence the nesting store contents below the level of the operands made ready for it.

It will also be obvious, as soon as multi-level sub-routines are considered, that the operations involved in storing the return instruction address for use after each
Table 1

<table>
<thead>
<tr>
<th>OPERATION</th>
<th>N1</th>
<th>N2</th>
<th>N3</th>
<th>N4</th>
<th>N5</th>
</tr>
</thead>
<tbody>
<tr>
<td>FETCH</td>
<td>b</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FETCH</td>
<td>a</td>
<td>b</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DOUBLE DUP.</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>-</td>
</tr>
<tr>
<td>DUPLICATE</td>
<td>a</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>MULTIPLY</td>
<td>$a^2$</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>-</td>
</tr>
<tr>
<td>MULTIPLY</td>
<td>$a^2 b$</td>
<td>a</td>
<td>b</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DUPLICATE</td>
<td>$a^2 b$</td>
<td>$a^2 b$</td>
<td>a</td>
<td>b</td>
<td>-</td>
</tr>
<tr>
<td>MULTIPLY</td>
<td>$a^4 b^2$</td>
<td>a</td>
<td>b</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ADD</td>
<td>$a^4 b^2 + a$</td>
<td>b</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>REVERSE</td>
<td>b</td>
<td>NUM</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FETCH d</td>
<td>d</td>
<td>b</td>
<td>NUM</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FETCH c</td>
<td>c</td>
<td>d</td>
<td>b</td>
<td>NUM</td>
<td>-</td>
</tr>
<tr>
<td>MULTIPLY</td>
<td>cd</td>
<td>b</td>
<td>NUM</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DUPLICATE</td>
<td>cd</td>
<td>NUM</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>MULTIPLY</td>
<td>$c^2 d^2$</td>
<td>b</td>
<td>NUM</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DUPLICATE</td>
<td>$c^2 d^2$</td>
<td>$c^2 d^2$</td>
<td>b</td>
<td>NUM</td>
<td>-</td>
</tr>
<tr>
<td>ADD</td>
<td>$2c^2 d^2$</td>
<td>b</td>
<td>NUM</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ADD</td>
<td>DENOM</td>
<td>NUM</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DIVIDE</td>
<td>NUM/ DENOM</td>
<td>= f</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

sub-routine in turn are again of a last in, first out nature. Another nesting store is therefore provided for this purpose, but there is here, of course, no necessity for a number of inter-connected registers or any arithmetic facility. This "sub-routine jump nesting store" (S.J.N.S.) therefore has one register as its most accessible cell, with again a sixteen-word core plane below. As in the case of the main nesting store, only a total of 16 words, not 17 as might have been expected in this case, are available. Since one is reserved for a special purpose (see below), only 15 may be used by the programmer.

The instruction "Jump to Sub-routine" has the effect of planting its own address in the top cell of S.J.N.S. and of causing an unconditional jump to the entry point of the required sub-routine. It will be noted that
S.J.N.S. must receive not only the word address but also the syllabic address of the return point.

Any sub-routine is terminated by one of the variants of the basic EXIT instruction, which transfers control to the instruction whose address is in the top cell of S.J.N.S. (nesting it up one cell). These variants augment this address by units of three syllables before performing the basic EXIT operation, and thus return control to the instruction immediately following the "Jump to sub-routine" instruction (itself three syllables long) or to one of the succeeding three-syllable instructions. These are usually a string of unconditional jump instructions, corresponding to failure exits or multiple exit points preceding the normal return instruction.

The Q-Stores

The computer organization is completed by addition of a further set of storage registers (not of nesting type) known as the Q-stores, and by provision of an input/output system.

The first of these features is based on conventional practices, and will be treated briefly. A set of fifteen registers (formally 16, one of which is identically zero) is used for address modification, counting and a number of other purposes. Each register is a full 48-bit word, but for address modification is considered as having three 16-bit independent sections for modifier, increment and counter.

Any main store reference has associated with it a Q-store number (or an implied Q0), and refers to the address specified, augmented by the content of the modifier section. If the letter Q is added to the mnemonic form of the instruction then after the address has been calculated the modifier is changed by addition of the increment and the counter is reduced by one. Jump instructions testing the counters are, of course, provided.

The Q-stores may also be used if desired as 48-bit registers, or as independent 16-bit registers, in each case with accumulative or direct input. The "counter" part of any Q-store may be used to hold the amount of shift (positive or negative) in shift instructions. There are sufficient facilities of this kind in the machine to remove the need for any kind of programmed instruction modification.

The other main use of the Q-stores is in connection with input/output operations, outlined in the next section.

Input/Output Operations

Provision is made for use of the normal peripheral devices, each one operating completely autonomously and simultaneously with other peripherals and with computer operations. In the standard system up to 16 peripheral devices, with a total transfer rate in excess of a million characters per second, may be handled.

To call any peripheral transfer, an instruction specifies the nature of the operation, referring also to a Q-store in which the other required parameters have been planted. These are the device number, and the limiting addresses of the main store area concerned, since peripheral transfers are of variable length.

Assuming that the device is available (i.e., not already busy), a check is now carried out by the input/output control system that the main store area specified does not overlap that involved in any peripheral transfer already in progress. The parameters are restored within the control, so that the Q-store register is freed for further use by the program.

The transfer now proceeds in a manner which, for economic reasons, differs depending on whether the device concerned is fast (magnetic tape, for example) or slow (punched card or paper tape, etc.).

In the case of the former, six-bit characters are assembled (taking a read operation by way of illustration) into machine words. When a word is complete, it is placed in a single word buffer and a signal to the main control system seizes a store cycle as soon as possible to transfer the word into store. Such calls have priority on the time of the main store, but any one peripheral device may have to wait until other devices have been dealt with.

Since such a buffering system uses a substantial quantity of equipment, more economical procedures are adopted for the slower devices. A common unit is shared by all these, and no assembly into words takes place outside the main store. Instead, as any device has a character ready, the required main store word is extracted, the character inserted in the next character.
location and the word is returned to store. This process is somewhat prodigal of main store time, but the penalty of handling devices with character rates up to a few kilocycles per second in this way is quite negligible.

An attempt by the computer to make use of a peripheral device or any part of a main store area concerned in an uncompleted peripheral transfer results in a "lock-out." The programme operation is interrupted until the prior operation is completed. The user is thus freed from any obligation to check and guard in his program against such conflicting operations.

KDF.9 "User Code"

Up to this point it has been implied that programs are written directly in machine code, albeit in mnemonic form. A very simple compiler routine can clearly translate the mnemonics into instructions. This would, however, leave the programmer with certain tedious tasks, such as calculation of addresses for jump instructions. The syllabic instruction form increases this problem.

The opportunity is therefore taken to incorporate in the compiler a number of additional features to eliminate such difficulties. One or two specific facilities will be mentioned.

The mnemonic code handled by the compiler is called "User Code" and has the following important characteristic: every User Code instruction is either a directive to the compiler, not appearing explicitly in the compiled version of the program, or is compiled into one machine instruction proper. Thus there are no macro-instructions in User Code which become sequences of instructions in basic machine code, the correspondence between User Code and machine code instructions being essentially one to one.

Calculation of jump addresses is handled by the compiler, the user being required only to label the entry point with an arbitrary reference number and to specify "jump to reference number... if..." (for example JrCqZ is the mnemonic meaning "jump to reference r if the counter section of Q-store q is zero").

Further elaboration of this process allows a similar labelling of sub-routines. When it encounters an instruction JSLi, the compiler ensures that a copy of the library sub-routine i will be made from the library tape and incorporated into the program. It also writes the appropriate entry address into the jump instruction in the main program.

It will be remembered that in section 3 an instruction YA was used as a mnemonic for "Fetch into N1 the word in main store address A." It is clearly possible for the programmer to treat as his data store a continuous main store area to be addressed as Y0, Y1, etc. The compiler is again used to convert these relative addresses to absolute addresses, allocating as the starting point the first word available after assembly of the programme.

An obvious extension is to allow the programmer to use a number of groups designated YA, YB, etc., each having an area commencing at zero. Thus YA0, YB75 are permissible addresses.

Other areas of store are similarly allocated by the compiler. Constants appear to the user as a set of stores known as V-stores. To incorporate a required constant for a program requires only the user code statement that, for example, V27 = F3.14159. This has the effect of converting the numeric value in standard floating binary form and allocating a store word to it. Any subsequent program reference to V27 fetches this constant into the nesting store.

Similarly the programmer can refer to main store areas reserved by the compiler for working store (storage of intermediate results or data). Exactly as for Y and V stores these are referred to with prefix W.

When the compilation is performed, the resultant program commences with the transcribed main program, followed by any sub-routines used. Areas for V and W stores (and for other functions of this type) are provided up to the highest numbered of each type in the original program. Finally the remaining area becomes the Y store area, thus allowing maximum generality.

Clearly it is possible to prepare special versions of the compiler with any degree of elaboration or desired characteristics. The incorporation of a routine for conversion of constants (which may be expressed in a number of ways) is a requirement for all uses, but it is evident that the principle can be extended as desired.

Thus the standard compiler, which is appropriate for preparation of programs for a variety of applications, may be supplemented by additional special purpose compilers.
The KDF.9 Computer System

Interruption and the Use of a Director Routine

KDF.9, like other computers of its generation, has built-in interrupt facilities. Specifically, this means that at virtually any time the normal sequence of instructions may be suspended and control transferred to a fixed address (syllable 0 of word 0 for obvious practical reasons). Such a transfer of control is called an Interruption. It is arbitrary only in the sense that it is generally outside the control of the programmer. It will take place only when one of a certain number of quite clearly defined situations arises in the machine. When an Interruption occurs, the interrupted program is left in such a state that it may be subsequently resumed and will then continue exactly as if nothing had happened - unless the reason for the interruption was some obvious abuse by the program of the facilities of the machine.

The purpose of the Interruption facility is to make "Time-sharing" possible. Here it is necessary to distinguish between "parallel operation" and "time-sharing." The former implies that the machine is doing more than one operation at a particular moment. On KDF.9 such occasions include the ability of one or more peripheral transfer operations to proceed at the same time as computation as long as no lock-out violation occurs (see above). If such a violation does occur, a transfer of control is necessary in order to enter some other sequence of instructions which can proceed unaffected by the lock-out.

This switch of control is automatic, and so implies the necessity of interruption, in order that the programmer shall not be required to anticipate and take action over lock-out violations. The ability to switch from one instruction sequence to another is called "Time-sharing"; it will be evident that time-sharing and parallel operation go hand-in-hand, the former enabling the most efficient use to be made of the latter.

There are two versions of the KDF.9 system. One of them has an elaborate time-sharing facility which enables up to four independent programs to be stored within the machine at once (together with a supervisory program or "Director"). They are obeyed on a time-shared priority basis - that is, each program is allocated a priority, and the hardware and the supervisory program together ensure that the program of highest priority is always operating subject only to peripheral lockout conditions.

The other version of the KDF.9 system only permits one program to be stored, in addition to a supervisory program. This version can be converted to "Full Time-sharing" by addition of equipment. The extra equipment includes:

(a) Extra core planes which enable each program to have its own Nesting Store, Subroutine Jump Nesting Store and Q-Store. To switch Nesting Stores, for instance, it is necessary only to nest down three places, so that the entire contents of the nesting store are in the bottom sixteen cells. These are all in the core plane, which can then be disconnected from the top registers and replaced by another core plane. This is a very satisfactory compromise between loss of time during changeover and volume of extra equipment required.

(b) A register corresponding to each priority level, which is set whenever that program is held up by a peripheral lock-out and which records the details of the lock-out. Interruptions are caused whenever a hold-up occurs and also whenever a lock-out is cleared which was holding up a program of higher priority than the one currently operating. For this purpose a register noting the current priority level is also necessary.

Features common to both types of machine provide for relative addressing and for address checking. The relative addressing feature causes the contents of a "Base Address" register to be added to any main store address used by a program before access to the main store is made. This allows programs to be coded always as if stored at location 0 onwards, but to be stored and obeyed in any segment of the store.

The address checking actually precedes augmentation of the relative address by the base address, and includes a check that the relative address is not negative and does not exceed the size of the store area allocated to that particular program. (Core storage allocation is completely flexible and is in the hands of the supervisory program. Programs may be moved about bodily in the store and may have their priorities interchanged).

If a program tries to go outside its allocated storage area a "Lock-in Violation" is said to have occurred, and an interruption follows. The same thing happens if a program tries to use a peripheral device which
has not been allocated to it - a register of currently allocated peripheral devices is automatically referred to every time a peripheral transfer is called.

In addition, interruption will occur if an ordinary program tries to use one of a number of instructions which are reserved for use by the Director. Such instructions provide access to the various hidden registers concerned with the interruption facilities.

The over-riding objective is to ensure that no program is capable of doing anything which can upset the operation of any other.

Other interruptions can be instigated by the machine operator, in order to allow input of control messages on the console typewriter. The program itself may also include instructions which cause entry to the Director in order, for example, to ask for allocation of peripheral units.

One other reason for interruption, of particular significance, is that which occurs whenever a peripheral transfer called by the Director itself comes to an end. This enables a certain amount of "programmed time-sharing" within the Director. There are many interesting and valuable possibilities. Thus, the feature is used to allow programs to output "lines of print" which the Director can send direct to a printer or to a magnetic tape for subsequent off or on-line printing - the latter again Director controlled. This allows standard programs to be written so as to be capable of running on systems having different output device configurations.

The only limitation on the facilities offered by such supervisory routines lies in the size of program which can be allowed without restricting the "main program" unduly. It is therefore expected that in addition to the "standard Directors" a number of others will appear for various ranges of use.

Any Director must satisfy a number of requirements. It will be stored throughout normal operation of the machine in word 0 onwards of the main store and will be entered at this point by any interruption. On each such entry, it must:

(a) preserve, as far as is necessary, the state of the interrupted program;
(b) discover the reason for interruption and take appropriate action;
(c) return to program.

These requirements will be considered in turn.

(a) This involves very little work on the Director's part. The address to which control must be returned when the program is resumed is automatically planted in the S.J.N.S. by the interruption process. This makes use of one of the "spare" cells of this store, leaving one for use by the Director itself.

Similarly the Director makes use of the three spare cells of the Nesting Store and therefore does not have to do anything to preserve the sixteen cells used by programs. It must, however, store away the contents of any Q-stores which it is going to use itself (usually three) and of two special registers provided to record arithmetic overflow and peripheral device states.

(b) The Director has access to a "Reason for Interrupt" register, different digits of which are used to indicate the different reasons. It is, in fact, the setting and subsequent detection of non-zero bits in this register, as various situations arise, which triggers off the interruption sequence. Once interruption has occurred it cannot occur again until control has been returned to program; however, digits corresponding to reasons for interruption may appear in the register at any time (they are cleared out as soon as the register is read by the Director).

(c) In the case of a KDF.9 system with full time-sharing facilities, this requires the Director first to determine the priority level to which control will be returned, and then to arrange for connection of the appropriate Nesting Stores, etc.

On any machine, the Director must also restore any Q-stores which were temporarily parked away, and must reset the two special registers mentioned in (a) above. The Base Address register, which was at zero while the Director was in operation, must then be set before finally using a special version of the EXIT instruction to return to the main program (this special instruction removes the inhibition preventing interruption while the Director operates).

CONCLUSIONS

In the space available it has been possible only to outline some of the distinctive features of the KDF.9 system. Many of these arise from the novel arrangement of working registers, whereas others are unspectacular
The KDF.9 Computer System

in terms of hardware but are the product of close investigation of operational needs. No attempt has been made to catalogue the performance factors of the KDF.9 system or even to include a specification. It is, however, confidently anticipated that it may point the way to even more striking improvements in the ratio of performance to cost.

Apart from the standard versions of the Director already mentioned, efficient use of a system of the proper KDF.9 presupposes the availability of an adequate "software package." Prominent among the individual items associated with the system are the following:

(a) A fast-compiling "load and go" ALGOL Compiler [9].
(b) A fast object-program ALGOL translator [10, 11].
(c) ALGOL program testing and operating systems.
(d) An ALGOL procedure library.
(e) Translators for FORTRAN, COBOL and other languages.

Detailed descriptions of some of these items appear elsewhere.

ACKNOWLEDGMENTS

In preparing this paper, the author is privileged to report the work of enthusiastic teams of designers and users led by R. H. Allmark and C. Robinson respectively. Most of the members of these groups have made significant original contributions and mention of individuals in such a team enterprise is out of place. The writer wishes to express his thanks to them and also to the Manager of the Data Processing Division of the English Electric Company, Ltd., for permission to publish this paper.

REFERENCES