THE UNIVERSITY OF NOTTINGHAM

FACULTY OF APPLIED SCIENCE

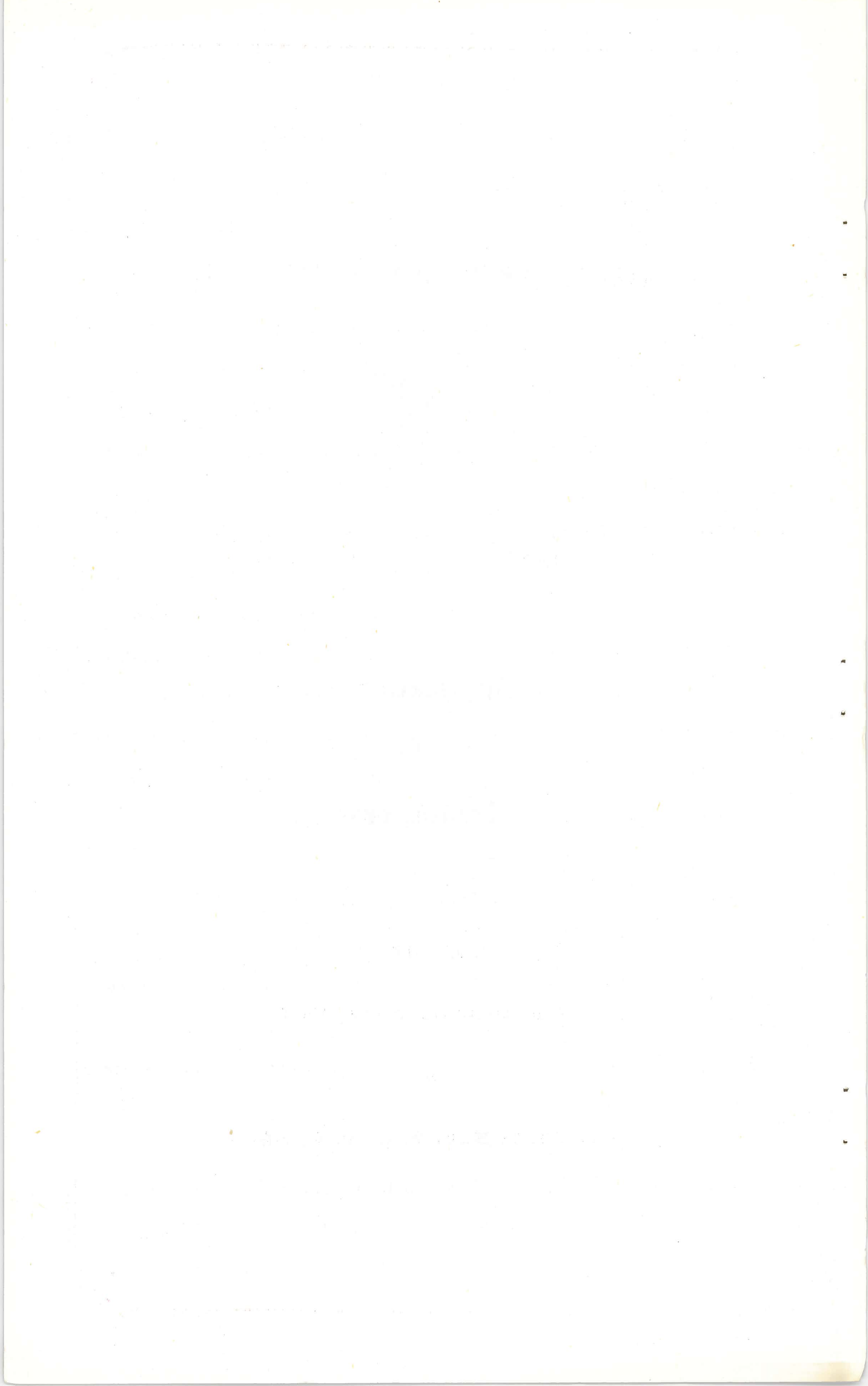
Applications of Computers

LECTURE 6.

"BASIC PRINCIPLES - CONTROL"

by

A. S. DOUGLAS, B.Sc., M.A., Ph.D., A.Inst.P.



"BASIC PRINCIPLES - CONTROL"

INTRODUCTION

Fast arithmetic and rapidly accessible storage naturally require rapid control if the resulting operating speeds are to take full advantage of the components. Such a control can readily be achieved by making suitable connections between stores and the arithmetic unit, on the principle of a telephone exchange system. This approach was in fact followed in the first electronic calculator, the ENIAC. However, the necessary plugging up for a particular job takes a long time, and it is desirable that there should be a high degree of flexibility in operation. As was shown in the report of Burks, Von Neumann and Goldstine of 1946, this can best be achieved by controlling the sequence of operations by 'instructions' held on rapidly accessible storage, the actual set being employed at any time being able to be changed rapidly when required. We shall now indicate how such a set of instructions may be stored on elements similar to those used for number storage.

CONTROL OF ARITHMETIC OPERATIONS

Standard arithmetic operations such as addition, subtraction, multiplication and division involve the processing of two numbers to produce a single resultant number. When we specify such an operation, therefore, we normally supply two numbers, a and b say, and, denoting the result by c, write, for instance:

$$a + b = c$$

Any such specification involves two numbers and a function, and must also determine what is to be done with the result. Except in certain limited contexts this is not convenient within the computer, since, among

other things, each instruction would require considerable space. It is customary, therefore, not to specify the numbers themselves but only where they are to be found within the computer. Our fundamental instruction thus becomes:

$$C(A) \quad + \quad C(B) \rightarrow C,$$

where A, B and C are positions or 'locations' in the computer storage, and C(A) stands for 'contents of A'.

It is easily seen that the locations can be labelled with numbers, called 'addresses', according to some convenient scheme, and that these labels can be stored in the same way as numbers in the computer. Furthermore we may designate the various operations which an instruction can cause to be carried out by a numbering scheme also, 01 standing for + for instance, and these 'function digits' can also be stored readily.

If we store instructions in the computer in the form indicated, then control must also know in what sequence the instructions are to be called upon, and thus our complete instruction must also include information as to where the next is to be found. A full explicit instruction must thus be of the form A, B, C; F;N, where A, B, C are numerical addresses, F consists of one or more function digits, and N is the address of the next instruction. Such an instruction would be termed 'four address' or 'three plus one address'. On reasonable assumptions about the length of number-words and of the addresses necessary in such an instruction it is usual for one instruction to occupy about the same number of digits as a number. Very few computers in fact employ this system.

By adopting suitable conventions we may reduce the number of addresses required in the typical instruction. Thus we may automatically extract instructions from addresses in some determined sequence, e.g. from consecutively numbered locations, thus eliminating N altogether. Further

we may impose the condition that either A or B is the same as C. If this convention is adopted, then C is termed an 'accumulator', since it may readily be used to accumulate totals. It is in fact adequate, and even in some cases desirable, to provide a single accumulator in a computer. In this case C itself need not be specified at all, and only the address A(or B) need be given. A control operated in this way is termed a 'one-address' system. In this case it is common for two or more instructions to be stored in the same number of digits as a number-word.

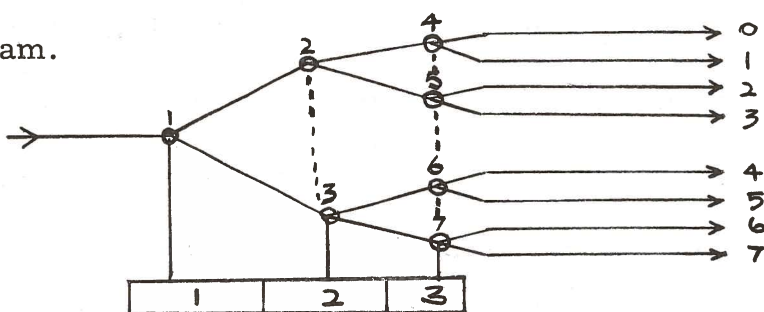
Between the extremes of one-address and four-address systems lie a large number of compromises, which allow, for instance, for more than one accumulator, or for a limited selection of locations from which the next instruction may be drawn. Machines with fairly large working stores are commonly one-address or nearly so, whereas those with a small working store and a large backing store are often two address, although one of the two addresses is usually restricted. Machines with a working store with anisomeric access, such as a drum or long delay lines, normally use a system embodying a timing number or an explicit address for the next instruction, so that successive accesses can be suitably adjusted to give a minimum of delay.

In the analysis given above no greater degree of complexity in one instruction than of the simple operations of arithmetic has been considered. Any more elaborate system would involve more addresses per operation, and would greatly complicate the control for carrying out arithmetic. Nevertheless in future machines more elaborate individual instructions may be provided, and we shall discuss how this may be done.

FUNDAMENTAL CONTROL OPERATIONS

Within the computer it is possible to regard each location in the working store as being connected to one of two or more inputs of the arithmetic unit, and also to the one or more outputs provided. Information in a location remains there and takes no part in operations unless it is fed to the arithmetic unit. This may be done by opening a 'gate' on the output of the location which is normally kept closed. Similarly information in a location can be replaced by opening a gate into the location from one of the arithmetic unit outputs, allowing new information to flow in and destroy the old. Control of the store and thus of computer operation is therefore done by selecting and opening, in a suitable sequence, gates throughout the computer, coupled with specification to the arithmetic unit by means of the function digits what operations are to be carried out within it.

Selection of the requisite gate, given an address A is done by means of a decoding 'tree'. To illustrate the operation of this device we consider the problem of constructing a switching system such that when a pulse is fed in it comes out on one of eight output lines, the line being selected by a 3-bit number. Let us use two-way switches, the state of each of which is controlled by one of the 3 bits, arranged according to the following diagram.



We suppose that switch 1 is operated by the state of bit 1, and connects with 2 if bit 1 is 0 and with 3 if bit 1 is 1. Similarly 2 and 3

are controlled by bit 2, the connections being 2 → 4, 3 → 6 for bit 2 = 0, 2 → 5, 3 → 7 if bit 2 = 1. Switches 4, 5, 6, and 7 are controlled by bit 3 in a similar way. We note that, if the number in the controlling position is 010, for instance, switch 1 is connected to 2, which is connected to 5, which is set so that output is on line 2 (the settings of 3, 6 and 7 are immaterial, since 1 is not connected to 3). Inspection assures us that a binary number in control selects the output appropriate to its value. In this way, therefore, we may express an address number in terms of a selected gate. The time of opening of the gate will, of coure, depend upon the timing of a pulse through the tree reaching the gate, and this is determined by the phasing of operations in the control. Clearly several gates can be attached directly, or through delays, to the output wire and their opening thus controlled in a suitable sequence.

A similar system can be used for selection of a particular function to be performed within the arithmetic unit, and, by a suitable arrangement- of connections the output wires can alter the setting of the controlling number. In this way a whole sequence of elementary gating instructions can be carried out automatically without further reference to the store for instructions. The details of construction of such a system are complicated, but the sequences of gating operations are analogous to sequences of instructions in the computer. These sequences are, therefore, sometimes known as 'microprograms', after the terminology of Dr. M. V. Wilkes, who first introduced the subject.

It is seen from the foregoing that the designer of the control has a wide choice of possible systems. The simplest control to engineer is one which throws the whole burden of timing of gating operations and selection of positions directly upon the programmer. Such a system would, however, require a considerable knowledge of engineering on the part of

the programmer and would require much detail in programming. The programmer, however, is mainly concerned with using the machine to solve problems in arithmetic, logic, or organisation and these already involve much detail in terms of the arithmetic operations described above. Greater complexity in operation is possible, and, by using techniques such as micro-programming, may even be made relatively easy to engineer. But it involves much equipment which is both expensive and liable to failure, this liability becoming increasingly important the more equipment is used. There are, therefore, powerful arguments for engineering simplicity, and it is usual to adopt a compromise, which combines reasonable simplicity in instruction coding with minimal equipment and maximum speed in control.

SEQUENCING OF INSTRUCTIONS

It was pointed out above that some indication, either explicit or conventional must be given to the control of where instructions are to be found and in what sequence. If no explicit indication of where to find the next instruction is included in each instruction, then certain special instructions are necessary which enables breaks in sequence to be made. These are called 'jump instructions'. Such jumps enable a group of instructions to be obeyed repeatedly and thus increase the power of the computer without increasing the storage demand. A system repeating a group of instructions without cessation would be of little practical value, however, and it is necessary to provide a method of getting out of the sequence upon the occurrence of some event, such as the completion of a specified number of repetitions. Instructions are therefore provided which break sequence except upon the occurrence of such a condition; these are called 'conditional jumps'. The condition tested is often the

setting of a particular bit in the store, such as that representing the sign of a number in a selected location, or that indicating that overflow has occurred, but any condition for which a test can be devised is acceptable. As well as simplifying the use of repeated sequences, conditional jumps enable us to select one of two (or more) possibilities, according to some criterion, and thus give the machine a function of decision, provided that the criterion applied can be suitably formulated. This is of the utmost importance in control applications.

MODIFICATION OF INSTRUCTIONS

Whilst the repetition of fixed sequences of instructions is useful under many circumstances, it is more usual to wish to carry out the same operations upon a sequence of numbers. For instance, let us form the sum of the set of integers 1 to 100. We take 1 and 2 and add to form 3, next we take 3 and the preceding result (also 3) and add to form 6; the latter procedure of adding the result to the next number may then be repeated until 100 is reached, and theresult then given is the required total. Using machine language let us put the numbers 1 to 100 in locations 0 to 99. We will store the result each time in 100. Our sequence of operations reads:

$$C(0) \quad + \quad C(1) \longrightarrow 100$$

$$C(100) \quad + \quad C(2) \longrightarrow 100$$

$$C(100) \quad + \quad C(3) \longrightarrow 100$$

etc.

Obviously the latter instruction could be contracted to a repeated sequence of the form $C(100) + C(i) \rightarrow 100$ if we can arrange that the value of i is increased by one at each repeat. This requires only that the address be increased by one, and, since the address is

held in numerical form, this can be done by adding a suitable number to the instruction. Such an addition can be done through the arithmetic unit or by a separate adding system and can take place either by changing the instruction in the store, or during the processing of the instruction within the control unit. The simplest system engineering-wise is to store instructions as if they were numbers in the computer and to process them through the arithmetic unit when it is desired to change an address. In this case any address in the instruction or the function digits themselves can be changed by adding suitable constants to the instruction-word. Integrating storage of numbers and instructions in the way indicated has the advantage that the best use can be made for a particular problem of the storage available, an appropriate division being possible between instructions and numbers. In this system, however, the machine makes no distinction between the two in storage, and a number sent by mistake to the control unit will be treated as an instruction, with possibly interesting results. This, in fact, causes little embarrassment as a rule, but may make fault finding in programmes slightly more difficult.

From a programming point of view direct alteration of the instruction in the store by arithmetical processing is less desirable than modification of an address within control. It is frequently found that several instructions in a sequence are to be altered by similar amounts throughout a calculation, although each refers to a different position initially. Consider the operation of multiplication of two vectors each of 100 elements. This involves forming $x_i \times y_i$ for successive i values from 0 to 99. Let the x_i be stored in $100 + i$ and the y_i in $200 + i$. Our repeated sequence uses the instruction $C(100 + i) \times C(200 + i) \rightarrow 300 + i$, say. Thus three addresses are to be tied to the counter i as it proceeds from 0 through 99. It would

clearly be wasteful to process each address separately, and it is better to keep i in a special index counter, the contents of which can be added automatically to the address or addresses of any instruction suitably marked. If such an automatic system is provided, the index location is called a 'modifier', and we speak of 'modification' of an instruction. This method was originated in Manchester on the first machine built there. The marking of an instruction to select a particular modifier may involve a further address or addresses being included in the full specification. If the machine is fundamentally one-address then a single modifier is appropriate, and if more addresses are included it is still usual to provide modification facilities for one of them only.

It will be seen that modification methods applied to arithmetic instructions might also be applied to jumps or indeed to any other type of instruction. Furthermore the index counts are themselves sufficient to provide criteria for the conditional jump instructions, since control numbers can be compared with the counts, and jumps made conditional upon the count reaching these control numbers. In our example we would increase the count, i, after each multiplication operation and then test whether i had reached 99 or not. If not, then the operation would be repeated with the increased value of i; if so then the repetition would be regarded as finished. It will be seen that this technique implies that the instructions within the store are not altered throughout the repetitions. If i is held in location I, the instructions would consist of the sequence

(1) C(100) x C(200) → 300; all addresses modified by C(I);

(2) Increase C(I) by 1 ;

(3) Test C(I) and jump back to (1) if not equal to 99.

The contents of I are altered, and this 'index' or 'counter' position must have an adder (at least) associated with it or must form part of the working

store for numbers. Whilst there are still advantages in being able to process instructions arithmetically and in having a unified storage system for numbers and instructions, this is no longer essential if modification is used. It is thus possible to use special storage for instructions which has very rapid access for reading only. A particularly interesting possibility, first attempted on EDSAC II, at Cambridge, is that of storing certain frequently used instruction sequences on permanent storage of this nature, thus extending the effective instruction code of the machine to cover complicated operations without complicating control, and also reducing the actual storage requirement for those sequences.

ADDRESSING THE BACKING STORES

So far we have been concerned only with arithmetic operations and the techniques of controlling the working store and of handling instructions relating to them. The control system of the computer has, of course, the wider task of co-ordinating the operation of all sections of the computer. This is complicated by the varying methods and times of access to the storage and other devices involved. If every control operation is completed before a new one is begun much time is necessarily wasted in waiting for the slower moving parts to complete their operation. By suitable autonomous arrangements much of this waiting time can be eliminated, but the central control must then be capable of initiating operation of the autonomous systems and of sorting out the signals of completion issued by them or of timing their operation suitably. One method of carrying out this which has been discussed elsewhere by Dr. Gill is that of arranging a system of priorities for dealing with such signals which can then 'break-in' to the fundamental operation of the computer. This may well be suitable for the operation of very high speed systems which have to control much slow peripheral equipment.

The backing stores are typical examples of equipment which are normally autonomous or partly so. Thus the drum or tape mechanism will arrange to extract information, of which the address is given, into a buffer quite independently of the behaviour of the rest of the computer. However, before the information can be used we must allow time for location of the address and for reading out to the buffer. Unless some break-in procedure is employed, the time allowed would normally have to be the maximum possible, or else the timing would have to be left to the programmer.

In order to simplify this problem it is usual to arrange the transference of information in blocks large enough that the waiting time is not a large proportion of the total transference time. Block transference of this kind poses a difficulty in arranging instructions such that they are commensurate in length with other types of instruction. A typical transference instruction will comprise an address referring to the backing store block and another referring to the working store (or other backing store if direct transference is possible between them). The number of blocks in the backing store may be very large, and thus the address of this will be long compared with working store addresses used in arithmetic instructions. Various techniques have been used to overcome this difficulty, should it prove impossible to keep a transference instruction within the normal compass. Sometimes special provision has been made for linking pairs of instructions so that the double length is available for a transference specification. Another system used has been to use the modification facility to extend the effective instruction in the control, the modifier being added to the instruction, but shifted relative to the end of the instruction so that the total length is greater than that of a single instruction. Yet another method is to arrange that the transference instruction is of normal type, specifying an address, the contents of this

address containing a special word which contains any additional addresses required. Any of these techniques or all of them may be included in a particular machine.

One of the effects of the use of block transfers is to make it convenient to refer to parts of the working stores in terms of the blocks which can be transferred to them. Thus we may divide our working store into blocks of 64 words, each of which can be fed from one of the half-tracks on the drum. The transference instruction will thus be of the form 'read half-track A onto block B'. However, for purposes of arithmetic we require to refer to the individual words of block B. This can most conveniently be done by specifying the word in the form B. p, where p lies between 0 and 63, rather than considering the working store as a set of words numbered serially from 0 upwards. Naturally the two numbering systems are equivalent in principle, but the actual system in the control will often be most conveniently arranged according to the block and position scheme and this is frequently adopted. Translation from one system to the other can be arranged during input if desired in a manner similar to that described for decimal to binary conversion.

Instructions need not, strictly speaking, be of fixed length, but if not then each instruction must contain some indication of its own length in a conventional position, and the start of the next instruction must be indicated. Such a system overcomes the difficulty of transference instructions, but introduces complications in the decoding system for addresses, especially if numbers too are of flexible length. An address must consist of a reference to the starting point of the word required and an indication of its length, unless the latter is conventionally contained within the number-storage. Whilst machines using variable word length do exist, the advantage of this over fixed length systems has not been

clearly established as yet.

PROGRAMMING AS AN EXTENSION OF THE INSTRUCTION CODE.

It has been pointed out above that the fundamental control operations of the computer consist of selection and operation of gates throughout the machine, whereas the operations required by a user are often complicated sequences of these fundamental motions. Some discussion has been given of the design balance between carrying out such sequences automatically, however complicated, and carrying out only fairly simple sequences automatically, leaving the sequencing of these simple operations to be done by the programmer so as to produce the required result.

The point at which automatic operation leaves off and programming begins is to a certain extent arbitrary, and we may thus think of programming as alternative to hardware in building up an instruction code for the user. For instance, in certain work the trigonometric functions, cos and sin are frequently evaluated. This can be done by hardware or by programming the operations of multiplication and addition on numbers stored in the computer. A sequence prepared for a specific purpose such as this, and capable of being called upon for some simple instruction or pair of instructions is called a 'subroutine', and is capable of returning control to the original sequence on completion of its operation. It thus behaves in every way as if the instructions used to call it in were a part of the instruction code of the machine devised to evaluate the function specified. The subroutine differs from the hardware required only in speed and in the occupation of storage. Nevertheless the loss of speed by programming may not be great, and the storage occupied may in fact take less equipment than the hardware. There is certainly a degree of complexity in a subroutine for which the price in hardware would be too high to pay in order to obtain additional speed, unless the computer is to be a special purpose machine.

Using programming techniques to supplement the formal machine code, the aim is to make specification of a problem as natural as possible for the user. Experience shows that a user with engineering or mathematical training takes kindly to a mathematical notation, and it is on this assumption that a system of 'autocodes' have been built up by Glennie, Brooker and others (but notably by Brooker at Manchester), based on the translative techniques described for input (originally developed by D. J. Wheeler at Cambridge). Brooker's autocode which will be dealt with in a later lecture consists of an instruction code utilising minor changes from and additions to standard mathematical notation together with a 'symbolic addressing' system within the computer (location addresses are referred to not as numbers, but symbolically). The computer is used to translate this code in terms of a number of subroutines and of the basic machine code so as to perform the operation specified. For convenience such codes are always applied to numbers in 'floating point' form, thus concealing any difficulties due to magnitude, and they are thus peculiarly suited to machines providing floating point facilities in the basic code.

For uses not directly connected with mathematical work other types of codes have been developed, though none yet so universally satisfactory to users. These employ various techniques, notably those of 'interpretation', 'generation', 'compilation' and 'assembly'. In interpretation systems the machine is programmed to simulate a control system with a simulated arithmetic unit, working stores and so on. The system simulated has an instruction code different from (or the same as) the computer itself, and may provide additional facilities not offered by that code. It is possible to provide an autocode in this way, but it would be slower than the translative system usually employed, which

does not involve simulation of a machine but only translation of the given instructions in terms of the basic code. The techniques of compilation and generation are used to avoid excessive programming preparation where basic code is to be used. By the supply of a few symbols it may be possible to specify a required operation the instruction for which can then be 'generated' by a simple process determined by a subroutine. In a similar way we may 'compile' a complete operation by bringing together its constituent parts arranged as small sequences of instructions stored in various places. The complete operation required can then be carried out by 'assembling' the sequences compiled or generated. The details of these techniques have been highly developed in the United States by Dr. Grace Hopper for application to commercial work.