SECTION 3 - INSTRUCTION CODE AND PROGRAM

3.1. Structure and Coding of Instruction

The computer is operated by a 'program' which is the list of instructions required to effect a complete job. Each instruction directs the machine to carry out one of a range of comparatively simple operations, the type of operation being defined by its 'function'.

Normally there are 31 basic functions (0 - 30) on an EMIDEC machine, and a table of the instructions for these functions is given in Appendix II.

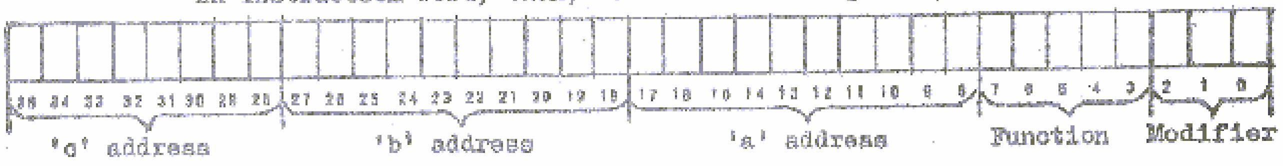
Programs are stored in the computer registers in exactly the same way as other working data. Each instruction is held in one register, and a further code is employed whereby the 36 bits of a register are used to represent the parts of an instruction.

One essential part of every instruction is the function number, mentioned above, which determines the type of operation to be carried out. In addition we must specify register 'addresses', the 'address' being the number of the register containing the numerical data upon which the operation is to be performed. The immediate access registers are numbered from 0 to 1023 and the instructions operate on the data in these registers by incorporating a reference or 'address' of the particular register or registers whose contents are required for the current calculation.

EMIDEC is a 2 address machine. That is to say it is so designed that one instruction is sufficient to carry out an operation involving the contents of two registers. Basically the two registers affected are the 'source' register and the 'destination' register. Thus to take the case of a simple transfer or copy instruction, it is required to reproduce the contents of register 'a' in register 'b'. The instruction must then specify the function 'transfer', which is function 1, the address of the source register from which the contents are to be copied, and the address of the destination register into which they are to be transferred. The parts of the instruction relating to these addresses are conveniently called the 'a address' and the 'b address', and this terminology is applied throughout to all instructions, though it will be found that with more complex functions these two addresses do not invariably relate to 'source' and 'destination'. It will be found too that for some functions additional information is required which cannot be accommodated in the 'a' or 'b' address. For instance with a shift instruction, in addition to specifying the source and destination registers, it is also necessary to indicate the number of places to be shifted. A further section of the instruction word is allocated for this purpose, and is termed the 'c' address', although it does not in fact represent the address of a register, but only additional numerical information.

One other final element is required in the composition of an instruction, and this is the modifier. The operation of modifiers will be explained later (Section 6), and it is sufficient now to say that space must be allowed for this purpose in the instruction word.

An instruction word, then, is coded in five groups, thus:

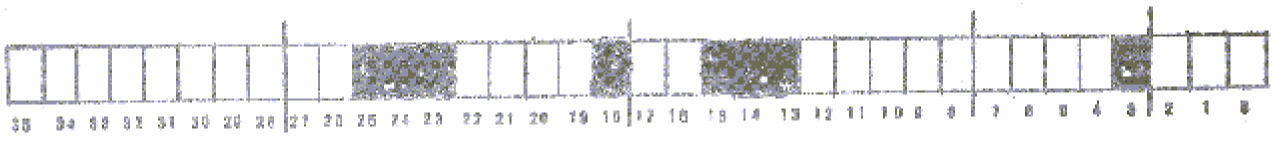


It will be noted that the modifier occupies three digits, from 0 to 2, the function occupies five digits from 3 to 7, and the 'a' and 'b' addresses each occupy ten digits from 8 to 17, and 18 to 27 respectively, and the 'c' address occupies eight digits from 28 to 35.

Thus taking, as before, a simple transfer instruction, suppose it is required to copy the contents of register 224 into register 225. The transfer function is 1 and no modifier or c address details need be specified, so that the instruction will be:-

c address	0	=	binary		0
b address	225	=	"	11100001	
a address	224	=	"	11100000	
Function	1	=	"		1
Modifier	0	=	"		0

In binary coded form this is 00000000/0011100001/0011100000/00001/000 so that when this instruction is itself stored in a register it will appear as:-



Again it will be seen that this is equivalent to the alphanumeric grouping 000000/000011/100001/001110/000000/001000/ and also equivalent to the pure binary 11100001001110000000001000 or 59,039,752.

Since words stored in registers may represent either numerical data or program instructions, it might perhaps be as well to reiterate the concept of 'address' to ensure that it is quite clear. The address of a register is its reference number and an instruction can only operate on the contents of a register by 'addressing' the register by its number. Part of the instruction word is used to indicate the addresses of registers whose contents are to be operated on. Confusion may arise because this part of the instruction word is itself referred to as the address part, and is divided into the 'a' address, the 'b' address and the 'c' address. It should be remembered that the term ''a' address' is really an abbreviation for "the part of the instruction word in which the address of the 'a' register (i.e. source register) is written", and similarly with the 'b' address. The ''c' address' however, does not indicate a register address at all and is so called only for convenience in referring to the parts of an instruction.

Finally it should be noted that the instruction word itself will be stored in a register and so will have an address i.e. the address of that register. This is quite distinct from the address part of the word itself which normally refers to the addresses of other registers.

It has now been explained how both program instructions and working data are represented by binary words which may be held in the immediate access store registers. For more extensive storage these words are held on the drum. They are held in exactly the same form since the magnetic surface of the drum also simulates binary 'ones' or 'zeros', and this surface is divided up, for ease of storage, into 256 'tracks' each occupying a full circuit of the drum each containing 64 words. More will be said later (see Section 4) about the arrangement of information in the store and the way in which it is extracted and used by the machine. It is now proposed, however, to consider some of the functions themselves, and the methods of writing them and using them in program sequences.

3.2. Writing Programs

It has been shown that, within the computer, program instructions are stored in binary code in the form:-

'c' address	'b' address	'a' address	Function	Modifier:

The machine is, however, provided with automatic conversion so that it is unnecessary for the programs to be actually written in binary form. The machine, in fact, is not only able to convert the numbers read in, but also to alter their order, within defined limits. Thus the written program to be presented to the computer, may be completed in ordinary decimal figures and the parts of an instruction may be set out in the order most convenient for writing. In practice this has been found to be as follows:-

Function	'a' address	'b' address	'c' address	Modifier

Coding sheets for preparing programs are therefore ruled in this form. The C address is divided into two columns since for some functions it is necessary to indicate two unrelated numerical values in this address. The two columns are known as the C1 and C2 addresses. Each half of the 'C' address occupies four of the instruction digits, from D_{28} to D_{31} and from D_{32} to D_{35} respectively. Dotted lines in the 'a' and 'b' addresses are provided for the purpose of an address referencing system to be explained later.

Generally speaking the 28 functions provided on EMIDEC may be divided into three groups, i.e.

1.	Inter-register arithmetic functions.	Functions 0 to 10, 26, 27, 29 & 30.
2.	Test functions.	Functions 11 to 14 and 25
3.	Functions transferring between registers and other units.	Functions 15 to 24 and 28

Within the first group there may be distinguished certain functions which are used not only for arithmetic but also for 'organisation'. That is, for the general handling and arrangement of information in registers.

We will deal first with the arithmetic functions, and certain other matters which arise in connection with their use. Then we will consider the 'organisation functions' (page 11) and the test functions (page 17). The functions used for transfers to and from the drum and the peripheral units will be explained later in the sections where the various units are described. (Sections 4 and 7).

3.3. <u>Inter-Register Arithmetic Functions</u>

<u>Register to Register Transfers - Function 1</u>

This is the simplest instruction and is used to copy the contents of one register into another. The contents of the first register are not altered in any way. Thus to transfer the contents of register 100 to register 101 we write:-

	1		100		101			

The contents of register 100 remain unchanged, but any previous contents of register 101 are cleared before the new value is transferred in. It will be noted that the source register is specified in the 'a' address and the destination register in the 'b' address.

We will next take the four normal arithmetical operations, add, subtract, multiply and divide.

Add - Function 7

This instruction adds the contents of one register to the contents of another, so that the sum appears in the second register. The contents of the first (source) register remain unchanged. So if we have 5 in register 300 and 3 in register 301, and we wish to add them together, we write:

	7	300	301			

The contents of register 301 will become 8, and register 300 will remain at 5. The original 3 in the register 301 will, however, have been lost.

Subtract - Function 8

Function 8 subtracts the contents of one register from the contents of another, placing the difference in the second register. The contents of the first register are unaltered.

To subtract 5 in register 100 from 12 in register 101, therefore, we should write:

	8	100	101			

The contents of register 101 are reduced to 7 and the original 12 is lost. Register 100 will, however, remain at 5.

Multiply - Function 9

This instruction multiplies the contents of one register by the contents of another, and places the result in registers 8 and 9. For the purposes of this function registers 8 and 9 operate as one register of double length i.e. 72 bits. The contents of the source registers for the multiplication are unaltered.

Thus, suppose we have 5 in register 200 and 6 in register 201 and we wish to multiply them together. We write:

	9	200	201			

The 5 is multiplied by the 6 and any previous contents of registers 8 and 9 are replaced by the product 30. The double length register 8 and 9 is arranged as the destination of all multiplication operations. Double length is provided since when the two numbers, each of which may contain up to 36 binary places, are multiplied together, the product may contain up to 72 binary places and in many cases therefore a single destination register would not be adequate to hold the result. Even if the product being calculated does not, in fact, exceed the capacity of one register, still both registers 8 and 9 are cleared and replaced by the product (the product in such a case being in register 9, and register 8 being reduced to zero).

Both addresses specified in the instruction indicate source registers and could as well be written in reverse order. It should be noted, too, that either register 8 or 9 (but not both as one unit) may be specified as a source register.

<u>Divide - Function 10</u>

In the same way that double-length capacity is provided for the product of a multiplication, so with division, in order to get a more significant result the dividend is always contained in registers 8 and 9 - again used as one double-length register - and the dividend must be placed in these registers before the division function is used. Instruction 10 therefore does not specify the address of the dividend, but the addresses of the divisor source and the quotient destination, in the 'a' and 'b' addresses respectively. The divisor and quotient are each to be contained in a single register only, and it must be ensured that the quotient, in particular, is not likely to exceed this capacity for should it do so all the results of the division will become meaningless.

After completion of the operation the quotient will appear in the register specified, and the remainder will be left in the double-length register 8 and 9. Since, however, the divisor is limited to a single register all the remainder must, in fact, be contained within register 9. The divisor remains unaltered after the division.

Suppose we wish to divide 30 by 4. The 30 must first be placed in register 9 and since so small a number will not extend into register 8, this latter register must be cleared so that the double-length register 8 and 9 contains only 30. If the 4 is in register 100 and we wish the quotient to appear in register 101 we write:

	10		100		101			

The 4 will remain in register 100, 7 will replace the contents of register 101 and 2 will remain in register 9.

It is possible to divide with a negative dividend or a negative divisor, or both. The remainder however, is always positive so that if the dividend is negative the quotient is in effect rounded up to the next highest whole number. Thus for instance if -367 is divided by 4, the answer given will be -92 with a remainder of 1. Or if -592 is divided by -7 the answer will be 85 with a remainder of 3. If however, only the divisor is negative the quotient will be rounded down in the usual way, so that 672 divided by -5 will give -134 with a remainder of 2.

Double-length Add - Function 26

To give greater facility for working with double-length numbers, EMIDEC has been provided with a double-length add function. By use of this function the contents of any two adjacent registers may be added to any other two adjacent registers, the two registers in each case being treated as one double-length register of 72 bits. The register addressed is, in each case, the lower numbered or more significant one of the pair. In other respects this function operates in the same way as a single-length addition, the sum appearing in the registers addressed in the 'b' address, and the contents of the 'a' address registers remaining unchanged.

Thus if we had one double-length number in registers 20 and 21 and we wished to add it to another in registers 30 and 31, we would write:

	26		20		30			

The sum would appear in registers 30 and 31, and registers 20 and 21 would remain unchanged.

Double-length Subtract - Function 27

This function also operates on two pairs of adjacent registers, treating each pair as one double-length register of 72 bits. Thus to subtract one double-length number in registers 20 and 21 from another in register 30 and 31 we would write:-

	27		20		30			

The difference would appear in registers 30 and 31 and registers 20 and 21 would remain unchanged.

Halt - Function 0 (and Program Alarm)

Cease operation and await manual restart. This function can be made conditional on the state of ten Halt Condition switches on the Console Desk.

If D18 of the instruction is ZERO, the halt is absolute.

If D18 of the instruction is ONE, the machine will halt only if there is a ONE in at least one of the digits D8-D17 AND the corresponding Halt Condition switch is DOWN.

A function 0 instruction can also be used to sound the alarm by making D28 ONE and D29-D35 ZEROS. No other combinations of D28-D35 will cause the alarm to sound.

Once sounded, the alarm can be silenced manually by pressing the Clear Alarm button on the Console Desk, or by program by means of a function 0 instruction, with D29 ONE and D28, D30-D35 ZEROS. A stop instruction with D28-D35 all ZEROS leaves the condition of the alarm unchanged. Certain other combinations of D28-D35 either leave the alarm unchanged, or silence it, but the ones quoted above are recommended.

Whether a stop is conditional or unconditional, and whether the condition is satisfied or not, is irrelevant to the operation of the alarm. The content of the 'a' and 'b' addresses is likewise irrelevant.

3.4. Special and Constant Registers

Before dealing with any further inter-register operations, it should be mentioned that a few of the immediate access store registers, numbers 0 to 16, have special uses. It has already been stated that registers 8 and 9 are used for products and dividends, and register 10 is similarly used as the destination of the collate operations to be described shortly. Registers 1 to 7 are employed for modification (see Section 6) and register 16 is the console register (see Section 8).

Registers 0 and 11 to 15 are the constant registers which differ in construction from the other registers and cannot be used as destinations, but only as the sources of the particular constants which they contain. The contents of the constant registers are as follows:-

Register	0	Zero				
"	11	One in position		D0	(i.e.	2^0 or 1)
"	12	" "	"	D8	"	2^8
"	13	" "	"	D18	"	2^{18}
"	14	" "	"	D28	"	2^{28}
"	15	" "	"	D35	"	2^{35}

These contents are provided so that they may be readily accessible in registers as they represent some of the values most commonly used in programming. Registers 12, 13 and 14 of course, in addition to their absolute arithmetical value, represent a 'one' in the first digit of the 'a', 'b' and 'c' addresses respectively.

3.5. Program Constants

Many cases will, of course, arise where figures are required which are neither included with the working data, nor provided by the constant registers. Such numbers can be formed by calculating from the constant registers, but normally it is more convenient to introduce them as part of the program. It has been shown that, within the computer, program instructions are stored in binary code so that inside the machine a program takes the form of a number of binary words stored consecutively in a group of registers. Similarly the information to be worked on will be stored in another group (or groups) of registers elsewhere in the machine. Now if we require to use a constant in a calculation it must be stored in registers, and if it is neither in a constant register nor in the working information registers then we must introduce it into a register by including it with the program. Such a constant is not to be considered as an instruction (though it may be one) but as a constant numerical value which is included with the instructions merely for the purpose of introducing it into registers. Care is in fact taken to see that such a constant is not carried out as an instruction, and it is usually written at the end of a program stage after the instructions themselves.

Alphabetic constants such as, for example, the word CREDIT are also introduced into the computer in this way, by inclusion in the program, so that they are stored in registers and are available as required for printing out.

3.6. Simple Arithmetic Operations

Using the functions and constants explained above it is now possible to construct simple program sequences for arithmetic operations. Suppose, for instance, we want 10 in register 100. The figure 10 may be introduced as a constant in the way explained above, but to illustrate the inter-register functions we will assume it is to be produced from the constant sources. It may be done by simply adding into register 100 as follows:-

0	1		11		100			
1	7		11		100			
2	7		11		100			
3	7		11		100			
4	7		11		100			
5	7		11		100			
6	7		11		100			
7	7		11		100			
8	7		11		100			
9	7		11		100			

Register 11 (Constant 1) is transferred to register 100, thus clearing the previous contents and setting up 1. A further nine 1's are then added to it.

The first column in the above sequence contains the serial numbers of the instructions from 0 upwards. These serial numbers are always printed down the left hand side of the coding sheets.

For a further example let us assume that we want to divide 100 by 3. We can obtain 100 by multiplying 10 (in Reg. 50) by itself, and we can find 3 by the process of adding. So we write:

0	9		50		50				100 in Regs. 8 & 9
1	1		11		101				)
2	7		11		101				{3 in Reg. 101
3	7		11		101				)
4	10		101		102				Divide

The result of the multiplication (instruction 0) will automatically appear in registers 8 and 9. In fact, since the product is only 100, this amount will be in register 9 and register 8 will be clear. For the division (instruction 4) the dividend must be in register 8 and 9, but, by virtue of the previous multiplication, it is already there. In the result we get 33 in register 102 (as directed by instruction 4) and the 1 remainder in register 9. Register 8 will still be clear.

Now suppose we want to divide 33 again by 3, then first we must place it in register 9:-

0	1		0		8				Clear Reg. 8 (see below)
1	1		102		9				33 in Reg. 9
2	10		101		103				Divide
3									

Since the division instruction invariably operates on both registers 8 and 9 it is necessary, when the dividend is transferred to register 9 only, to ensure that register 8 is clear, otherwise digits remaining in 8 from a previous operation may destroy the meaning of the division. Thus in the above exercise register 8 is cleared as a first step. (In fact 8 would have been cleared by the multiplication in the previous exercise, but the 'clear' instruction is included to illustrate the general principle). Next 33 is transferred into register 9, and then we can perform the division which will place the quotient 11 in register 103 and leave zero (the remainder) in register 9.

3.7. Packing

Before dealing with any further functions it will be helpful to explain the concept of 'packing'. It has been said that the basic storage unit within EMIDEC is the word of 36 bits, and any numerical or alpha information is contained within such words. The capacity of a 36 bit word is 34,359,738,367, and it is clear that in normal operation it will be rare to use numbers of such magnitude. It is therefore often the practice, in order to conserve storage space, to 'pack' into one word, two or more numerical values.

Suppose, for instance, that we are storing information concerning employees, and for each employee we have to hold a clock number of from 1 to 5000, a grade of 1 to 20 and a total of normal weekly hours worked from 1 to 60. Then, to accommodate the clock number we require 13 binary digits, for the grade 5 digits, and for the hours worked 6 digits. So we may allocate say D0 to D7 to the grade, D8 to D17 to the hours and D18 upwards to the clock number. All the information for one employee can then be held within one word. It should, however, be evident that while storage may be arranged in this way in sections of a word, yet for any arithmetic operations a complete word must be worked on, so that it is necessary for information to be 'unpacked' when it is to be handled arithmetically, so that any number occupies a complete word.

Both for 'packed' information and for the holding of instructions themselves, it is required that a word should be considered in sections. The functions which we are to examine next are used not only for arithmetic results, but also for the 'organisation' and handling of particular word sections.

3.8. <u>'Organisation' Functions</u>

<u>Collate - Function 6</u>

The collate function compares the contents of two registers and produces a number consisting of a 'one' in each binary place in which <u>both</u> of the registers compared have a one. The number so produced replaces the contents of register 10. Register 10 is always the destination of a collate operation and the instruction addresses are used to specify the two registers to be compared. To show the operation of this function let us suppose that in register 100 we have 16, which in binary is 10000, and in register 101 we have 21, which is 10101. If we then collate these registers we shall get, in register 10, 10000, that is, a 'one' in the D4 position, where both 10000 and 10101 have a one. If we had 31 in register 100, that is, in binary, 11111, then on collating with 10101 we should get a reproduction of the latter i.e. 10101. This facility is used for transferring sections of a word. Thus, if we were holding an employee's data in a single register as suggested in paragraph 6 above, with the grade in D0 to D7, hours worked in D8 to D17 and clock number in D18 upwards, we could, by use of the collate operation separate out one of these sections from the packed register as the multiplication would act on all 36 digits and the result would be meaningless. So we collate the packed register, say register 200 with another register containing 'ones' in each of the positions occupied in register 200 by the hours worked, i.e. D8 to D17. The instruction would be:

	6		200		201			

where register 201 contains the constant of 'ones' in positions D8 to D17. The result of the instruction is to replace the contents of register 10 with a copy of the contents of D8 to D17 of register 200, but with zeros in the remaining sections of the register. The number in register 10 can then be used for further arithmetic work.

It should be noted that register 10 may be used, if required, as one of the <u>sources</u> of a collate instruction. It may also be used for other operations as a normal register.

<u>Shift Left - Function 2</u>

The effect of this function is to take the contents of a register and shift them left by a specified number of binary places. The result may be placed in another register or may replace the contents of the source register. The source register is written in the 'a' address and the destination register in the 'b' address. The number of places to be shifted is specified in the 'cl' address. If the 'b' address is different from the 'a' address, then the contents of the 'a' address remain unchanged by the operation.

One of the main purposes of a shift instruction is to enable sections of a word to be moved to the appropriate places within a register. For example, when considering the collate functions we showed how the contents of a register could be 'unpacked'. Suppose now we wish to perform the converse process, and pack a register with these numbers ranging upwards from the D0, D8 and D18 positions. The three numbers are at present held in three separate registers (say registers 21, 22 and 23) in the usual position from D0 upwards. The first can be left at D0, but we require to add to it the second in D8, so before adding we must shift the contents of the second register 8 places to the left. So we write:

	2		22		22	8		

This shifts the contents of register 22, 8 places to the left, zeros being introduced from the right for the 8 additional places. The result appears in the same register. Register 22 can then be added into register 21.

Similarly, before being added to register 21, the contents of register 23 would be shifted 18 places, thus:-

	2		23		23	18		

There are of course other methods of packing but the above example illustrates the general principles involved.

The Shift instruction may also be used arithmetically for since the computer works in binary the effect of shifting a number one place to the left is, of course, to multiply it by 2.

For an illustration of this use, suppose we have 5 in register 200 and we require 20 in register 300. The instruction is

	2		200		300	2		

Thus in register 200 we have 5, which in binary is 101. This number is shifted two places left, so that it becomes 10100. 10100 is the binary equivalent of 20, so we have in effect multiplied 5 by 4 to give 20, which result is placed in register 300. The contents of register 200 remain at 5, but any previous contents of register 300 are replaced by the value 20.

Any number of places may be specified with this instruction (or instruction 3) from 36 (which has the effect of clearing the destination register) down to 0 which has the effect of a simple transfer from register 'a' to 'b'.

<u>Shift Right - Function 3</u>

This function operates in exactly the same way as function 2, except that the shifts are made to the right, thus dividing by 2 for each place shifted. Whereas in instruction 2 however, zeros were introduced from the right, in the present instruction the new places must proceed from the left. If the D35 digit in the source register is a zero, then the digits introduced will be zeros but if the D35 is a one then, it will be remembered, this indicates a negative number, and in order to preserve this sign value, the computer will automatically provide for further ones to be introduced.

In illustrating the collate function it was shown how the contents of D8 to D17 could be transferred from a packed register into register 10. To get the true arithmetic equivalent in register 10, however, we should require the numbers in the positions D0 upwards. This we would do by shifting the contents of register 10 8 places right into the same register, thus:

	3		10		10	8		

Function 3 is also used arithmetically. For instance, if we have 32 in register 400 and we want 4 in register 401 we write:

	3		400		401	3		

The 3 place right shift has the effect of dividing by eight so that the previous contents of register 401 are replaced by 4. The 32 remains in register 400.

Double-length shift left - Function 4

It will be remembered that for the multiplication and division functions it is required to use registers 8 and 9 as one single register of double-length i.e. 72 bits. In order to facilitate working with double-length numbers, two shift functions have been provided which operate on any two consecutive registers as though they were a single register of double-length. In using these double-length functions the address to be given is always the lower number of the two registers concerned, and this is the register which contains the more significant half of the double-length number. The double-length shift operates in exactly the same way as a single length shift, with the same maximum shift of 36 places.

To shift the contents of registers 500 and 501 18 places left into registers 551 and 552 we would write:

	4		500		551	18		

It should be noted that there is no double-length <u>transfer</u> function, Such a transfer may, however, be effected by using a double-length shift instruction specifying zero as the number of places to be shifted.

Double-length Shift Right - Function 5

This function operates in exactly the same way as the previous ones except that the shifts are made to the right instead of the left. The value of D35 in the lower numbered register will determine whether ones or zeros are introduced from the left.

The instruction for shifting the contents of registers 551 and 552 8 places right into registers 560 and 561 would be written:-

	5		551		560	8		

Shift Functions 2, 3, 4 and 5

The number of shifts performed is normally as specified in the instruction but, if the specification exceeds 39 but is less than 64, then 39 shifts will be performed. If the specification is greater than or equal to 64, then the effect will be that of specifying the difference between the greatest multiple of 64 less than the number specified and the number specified, e.g.

Specifying 195 = specifying 3 (i.e. 195-192)
performs 3 shifts;

Specifying 234 = specifying 42 (i.e. 234-192)
performs 39 shifts.

3.9. Fractions and Rounding Off

When we normally work in decimal fractions we are, in effect, by using additional decimal places, multiplying our number by 10 or 100 or 1000 to increase the scale of the number in order to get a more significant result. We put the decimal point so as to indicate how many times our number has been multiplied by 10. When working on the computer we may similarly obtain a more significant result by increasing the scale of the original number upon which we want to work. We cannot of course use a decimal point notation, but we can still multiply by 10 or 100 and so on, so long as we remember the extent to which we have magnified our numbers. Alternatively, since we are working in binary arithmetic on the computer we can increase our original numbers by binary multiples instead of by decimal multiples to obtain the degree of accuracy necessary. The latter method has the disadvantage that the results cannot be printed out in the conventional decimal form, but on the other hand it has the advantage that in the machine each multiplication or division by a binary multiple of 2 can be effected by a shift operation, and the position of a number in a register will indicate the number of powers of 2 to which it has been raised.

As an example of working in binary fractions, let us consider the process of rounding off an answer to the nearest whole number. If we refer again to the example in Paragraph 3.6 where we divide 100 by 3, it will be remembered that the answer obtained in register 102 was 33, while a remainder of 1 was left in register 9. Similarly 100 divided by 6 would give 16 in register 102 with a remainder of 4 in register 9. Sometimes it may be possible to ignore the remainder, but on the other hand it may be required to know the quotient to the nearest whole number. 100 divided by 6 for instance is - to the nearest whole number - 17 rather than 16. Now to round up to the nearest whole number we only require to know whether the remainder is more or less than $\frac{1}{2}$ of the divisor, so to obtain the necessary degree of accuracy it is sufficient to work in halves i.e. to one binary place. To express 100 in halves we multiply it by 2 or - what is equivalent - shift it one place to the left. We then have 200 which we divide say by 6, giving an answer of 33 - in halves. Since 33 contains an odd half, we add 1 to make 34 and then divide by 2 to bring the result back to whole numbers i.e. 17. Had the result in halves been 32, rather than 33, then we could still have added the half to make it 33, since on dividing by two the result would still be 16. The rule for rounding off is therefore: shift one place left at the beginning of the operation, add one to the result, and then shift back one place to the right. Let us now follow the procedure through in binary. 100 in binary is 1100100, so when we shift this one place left it becomes 11001000. Then when we divide by 6, instead of getting 10000 (16) we get 100001 (33). The final place is the rounding off digit and since it is a 'one' it indicates that the fractional part of the quotient is greater than one half. If it were 'zero' the fractional part would be less than one half. So in either case we now add one in the final place. Adding one to 100001 we get 100010. Then we reverse the effect of the original left shift by shifting one to the right and we get 10001. This is the binary for 17. Had the rounding off digit been zero i.e. had the number been 100000 the one added to the final place would give 100001 which, shifted one place right gives 10000 i.e. 16.

It should be remembered that a left shift of one place is equivalent to a multiplication by 2, so it may be possible to avoid the preliminary left shift by dividing by only half of the true divisor. Thus to divide 100 by 6 we in fact divide 100 by 3 only, which gives us 100001 (binary for 33) and then add one and shift right as before. Assuming that we have 100 in registers 8 and 9 and 3 in register 101, the instructions for the division would be:

0	10		101		102				Divide 100 by 3
1	7		11		102				Add 1 to Reg.102
2	3		102		102	1			Shift Reg.102 right 1
3									
4									

The answer 17 then appears in register 102.

It has been said that if we require an answer which is significant to a number of decimal places we multiply it by the appropriate multiple of 10. Suppose, for instance, we wish to multiply 9.53 by 7.64. This is equivalent to

$$\frac{953}{100} \quad x \quad \frac{764}{100}$$

We therefore simply multiply 953 by 764. (i.e. we have multiplied each number by 100). To get an answer to two places of decimals we would then divide by 100, but to get the final digit correct we would also apply the rounding off procedure already explained, so we should divide by 50, add one and then shift one right. So we get

$$\frac{\frac{953 \times 764}{50} + 1}{2} = 7281$$

and we know that this answer, to two decimal places is 72.81.

If we required an answer correct to the nearest whole number only, we would divide by 10,000 rather than by 100. If only a whole number result is required an alternative method could be adopted of multiplying the original 9.53 and 7.64 not by 100 but by 128 i.e. 2^7. We should then multiply 1220 by 978 and we would not have to use the divide function but instead we would effect a division by 2^{14} by shifting 14 places to the right - which is a quicker operation.

The program for the first method, assuming that we have 953 in register 100, 764 in register 101, and 50 in register 102, would be:-

0	9	100	101				Multiply
1	10	102	103				Divide by 50
2	7	11	103				Add 1
3	3	103	103	1			Shift right 1
4							

Let us assume now that we have the same constants, 953 in register 100, 764 in register 101 and 50 in register 102, and we want to divide 9.53 by 7.64. To obtain an answer to two decimal places we must multiply 953 by 100, and to apply the rounding off procedure to the last digit we require also to multiply by two. We can achieve the same result by multiplying by 50 and then multiplying by 4 by shifting two places left. So we write:

0	9	100	102				Multiply 953 by 50
1	2	9	9	2			Shift left 2 places
2	10	101	103				Divide by 764
3	7	11	103				Add 1
4	3	103	103				Shift right 1 place
5							
6							

There are, of course, other possible methods of rounding-off and working in fractions. The main points to be borne in mind however, are that any required number of significant figures may be retained by appropriate multiplication and shifts, and that a binary number may be rounded up at any point by considering whether the next lowest place is "one" or "zero".

3.10. Test Functions

Consideration of the test functions involves a new and important principle, that of a logical choice by the machine between two alternatives. If the machine is to be completely controlled by program it is necessary that it be able to make these "decisions" and follow the course of action appropriate to the data presented. The way in which the computer varies its course of action is by effecting a "jump" in the sequence of program instructions.

It has been stated previously that the computer reads and carries out instructions serially in the order in which they are stored, unless it is instructed to do otherwise. With a test instruction a register is inspected to see whether it fulfills certain conditions, and if it does not then the usual serial order is continued. If, however, the conditions are fulfilled, then the computer is instructed to "jump" and start carrying out instructions from some specified new location in the store. In each of the test instructions the 'a' address is used to specify the register to be tested, and in the 'b' address is written the number of the instruction to which a jump is to be made if the test succeeds. It has been mentioned that the instructions on a coding sheet are serially numbered, and the way in which the instruction serial numbers are referenced to registers within the machine will be explained later (see Section 5). In the following examples however "absolute" addresses are used. That is, the instruction numbers represent the actual register numbers in which the instructions will be held.

The test functions are as follows:-

Function 11 - Test Zero
Function 12 - Test non-zero
Function 13 - Test Positive (i.e. test D35 is zero)
Function 14 - Test Negative (i.e. test D35 is one).

Each of the above four test functions operates in the same way, the only difference being in the conditions required to be fulfilled by the contents of the register tested. These functions may best be explained by programming an example. Let us take a practical case where a man is paid on piece-work rates at 1/- per article, but with a guaranteed minimum wage of 40/- per day. Assume that we have 40 in register 100 and in register 101 the number of articles produced in a day. We require to put the day's pay in shillings in register 102. As a first approach we might write:-

	50	8		100		101			
	51	14		101		55			
	52	7		100		101			
	53	1		101		102			
	54	11		0		56			
51 $\longrightarrow$	55	1		100		102			
54 $\longrightarrow$	56								

We have to put in register 102 the greater of the contents of registers 100 (which we may call '<u>a</u>') and 101 (which we may call '<u>b</u>'). So first we subtract '<u>a</u>' from '<u>b</u>' (instruction 50). Then we test whether register 101 is negative. Register 101 now contains <u>b</u> - <u>a</u>, and if this is negative then <u>a</u> is greater than <u>b</u> and we wish to transfer <u>a</u> to register 102. If, on the other hand, <u>b</u> is greater than <u>a</u>, then register 101 will be positive and we wish to transfer <u>b</u> to register 102. In this latter case the test in instruction 51 will fail and the computer will proceed to the next instruction in sequence. So in instruction 52 we add <u>a</u> back to register 101, so that it again contains <u>b</u> and then in instruction 53 we transfer <u>b</u> to register 102. If however, the test in instruction 51 succeeds, the instruction directs the machine to jump to instruction 55 and here we write a transfer from register 100 (which contains <u>a</u>) to register 102. Thus the appropriate transfer is made to register 102 according to whether <u>a</u> or <u>b</u> is the larger.

Instruction 54 is also a test instruction of a special kind, known as an "unconditional transfer of control". Clearly if the machine carries out instruction 53 we do not want it then to carry on and also perform instruction 55. We require to make it jump round instruction 55 to instruction 56 where the next part of the program will commence. There is no question of a choice here. When instruction 53 has been performed we want, in every case, to jump to instruction 56. So as instruction 54 we write a zero test, directing the machine to jump to instruction 56 if register 0 is zero. We know however, that register 0 is a constant source, and is always zero. In effect, therefore, the jump is unconditional.

This sequence could be programmed more neatly without the use of the unconditional jump, as follows:-

50	8	100	101			
51	13	101	53			
52	1	0	101			
53	7	100	101			
54	1	101	102			

Here we deduct a from b and then test b - a positive. If it is positive then b is greater than a and the computer jumps to instruction 53 which adds a to register 101 to give b which is then transferred to register 102. If, however, a is greater than b the test fails and the computer proceeds to instruction 52 which clears register 101. The computer then goes on to instruction 53 which adds a to register 101, now making its contents a, which is then transferred to register 102. This clearly, is the better method which should be adopted in practice.

Count Test - Function 25

The count test is a specialised function which combines the functions of subtracting and testing for non-zero. The non-zero test is applied to register 7, which must therefore always be used with the count test function and need not be specified in the instruction. The effect of this function is to subtract the contents of a specified register from register 7 and then to test register 7 non-zero. The register to be subtracted is specified in the "a" address, and in the "b" address is written the register to which a jump is to be made if the test succeeds i.e. if the subtraction does not reduce register 7 to zero. After the operation the contents of register 7 remain reduced by the amount subtracted. The use of the count test function is explained in Section 6.

3.11. Additional Functions

Function 28 - Sterling Output Conversion with Spaces

1. Until now, the available instruction for the output of information with a sterling conversion has required that spaces between the pounds and shillings columns, and the shillings and pence columns, should be produced by appropriate wiring of an E.C.U. Sterling output to the paper tape punch has required programmed conversion to provide the spaces.

A new facility is now available by the inclusion of another order - function 28 - in the instruction code. This function enables a sterling output conversion to be produced with these spaces included.

2. When coding, function 28 may be regarded as similar to function 23, but with the difference that the automatically inserted spaces must be included in the number of characters specified to be sent out. Since the maximum number of characters remains at 12, the largest sterling amount which can be dealt with is £9,999,999 : 19. 11d. (With function 23, the limit is £143,165,576 : 10 : 7d., determined by the capacity of a 36-bit word). Should the binary amount exceed this value the most significant characters will be lost.

3. Due to the introduction of function 28 and the fact that certain microroutines are common to Output Decimal and Output Sterling the following times now supersede those already published.

Function 22	-	OUTPUT DECIMAL	- 2,990 microseconds
23	-	OUTPUT STERLING	- 3,220 "
28	-	OUTPUT SPACED STERLING	- 3,720 "

4. This function may be used for either Line-Printer or Paper Tape output, but it should be noted that, since no corresponding Input Spaced Sterling conversion exists, Paper Tape prepared using function 28 may not be read as Input to the computer using Sterling Input conversion, but instead will require a short routine (of about seven instructions) to convert it.

5. The function will not apply on any machine using Sterling halfpenny conversion.

Function 29 - Set Link and Jump Instruction

This instruction stores the number given in its own A address in the B address position of register 7, and then jumps unconditionally to the location specified in the B address. Thus for example the instruction

29	50	325			

causes '50' (NB. not the contents of register 50) to be set up in register 7 in the D18 position, and transfers control to the instruction in register 325.

The instruction has three principal uses.

1. To enter a subroutine from several points in a program, leaving in register 7 an indication of where to re-enter the main program.

2. To set a constant in register 7; e.g.

0	29		25	R	1			
1								

sets 25 x 2^{18} in register 7.

3. As a quick unconditional transfer of control.

Time: 110 microsecs.

N.B. Two further points should be noted in this connection:-

1. Zero in the 'a' address will have the effect of clearing register 7.

2. Negative numbers may not be set in register 7 using this function.

Function 30 - Negative Sum

This instruction calculates the sum of a group of sequential registers, and places the complement of this sum in a specified address.

The number of registers involved is specified in the 'C1' address. The first register is specified in the 'a' address and the complement is placed in the register specified in the 'b' address.

Up to 39 registers may be specified. If, in fact, no registers are specified the effect is that one will be taken (i.e. the same effect applies if the C1 address contains '0' or '1').

This function is primarily intended for use in connection with check totalling on Magnetic Tape.

e.g. To form a negative check total in register A16 for registers A1 - A15 inclusive we would write:

```
30 . A1 . A16 . 15 . 0 . 0
```

and to check the 'input' block in registers W1 - W16 inclusive the instruction would be

```
30 . W1  . W17 . 16 . 0 . 0
12 . W17 .  R  .  0 . 0 . 0 → Error
```

Other uses can be found easily for this function, e.g. Accumulate Registers B0 - B25 inclusive in B73, could be stated as

```
30 . B0  . B73 . 26 . 0 . 0
30 . B73 . B73 .  1 . 0 . 0
```

(It will be noted that the last instruction shown, in fact, merely complements the contents of B73).

The timing of this function depends on the number of registers specified and may be shown as follows:

$t = 130 + 30n$ μSecs $(1 \leqslant n \leqslant 39)$ <u>or</u>

$t = 160$ μSec $(n = 0)$

<u>Allocation of Instruction Digits</u>

<u>Function 0 - Halt</u>

Cease operation and await manual restart.

This function can be made conditional. If D18 is ZERO the stoppage is absolute. If D18 is ONE the machine will halt only if there is a ONE in any of D8 - D17 and the corresponding switch of a set of ten switches on the monitor desk is also set to ONE.

Digits 28 and 29 are used to control an alarm signal. A D28 switches the alarm on and a D29 switches it off.

<u>Function 1 - Register - Register Transfer</u>

Replace the contents of register <u>b</u> by those of register <u>a</u>, leaving register <u>a</u> unchanged.

Function 2 - Shift Left

Take the contents of register a, shift them left by the number of places specified in c, and replace the contents of register b with this result, leaving register a unchanged.

Function 3 - Shift Right

Take the contents of register a, shift them right by the number of places specified in c, and replace the contents of register b with this result, leaving register a unchanged. According as the most significant digit of the contents of register a is ZERO or ONE, the new digits brought in at the more significant end by shifting are zeros or ones.

Function 4 - Double-length Shift Left

Take the contents of registers a and a + 1, shift them left as one unit, with a the more significant half, by the number of places specified in c, and replace the contents of registers b and b + 1 with this result, leaving registers a and a + 1 unchanged.

Function 5 - Double-length Shift Right

Take the contents of registers a and a + 1, shift them right as one unit, with a the more significant half, by the number of places specified in c, and replace the contents of registers b and b + 1 with this result, leaving registers a and a + 1 unchanged. According as the most significant digit of register a is ZERO or ONE, the new digits brought in at the more significant end by shifting are zeros or ones.

Function 6 - Collate

Replace the contents of register 10 by a number having ONE in each place in which the contents of registers a and b both have ONE. The contents of a and b remain unchanged after this operation.

Function 7 - Add

Add the contents of register a to the contents of register b, placing the sum in register b. The contents of register a remain unchanged after this operation.

Function 8 - Subtract

Subtract the contents of register a from the contents of register b, placing the difference in register b, the contents of register a remaining unchanged.

Function 9 - Multiply

Multiply the contents of register a by the contents of register b and place the product in registers 8 and 9 (the less significant part in register 9). The contents of registers a and b are unchanged after this operation.

Function 10 - Divide

Divide the double-length number contained in registers 8 and 9 and having its more significant half in register 8 by the contents of register a, placing the quotient in register b and leaving the remainder in register 9. The quotient is positive or negative dependent upon whether the signs of dividend and divisor are the same or different; the remainder is always positive. The contents of register a are unchanged after the operation.

Function 11 - Test Zero

If the contents of register a are zero proceed to the instruction in register b; if not, continue with the next instruction in sequence.

Function 12 - Test Non-Zero

If the contents of register a are non-zero proceed to the instruction in register b; if not, continue with the next instruction in sequence.

Function 13 - Test Positive

If the contents of register a are positive or zero, (i.e. if D35 is ZERO), proceed to the instruction in register b; if not, continue with the next instruction in sequence.

Function 14 - Test Negative

If register a contains a negative number, (i.e. if D35 is ONE), proceed to the instruction in register b; if not, continue with the next instruction in sequence.

Function 15 - Transfer from Drum

Transfer from the drum the number of 4-word blocks specified in D32-D35. Not more than 16 blocks (i.e. one track) may be transferred; zero in D32 - 35 indicates a 16-block transfer. Only one track can be read in one instruction, i.e. information on two different tracks (but having consecutive block numbers) requires two instructions to transfer. The first block of the transfer is specified in D18 - 31, and the first destination register in a; the transfer proceeds sequentially until the required number of blocks have been transferred.

Function 16 - Transfer to Drum

Transfer to the drum the number (up to 16 as above) of 4-word blocks specified in D32 - 35. The first source register is specified in a, and the first destination block in D18 - 31; the transfer proceeds sequentially until the required number of blocks have been transferred.

Function 17 - Input Block Transfer

Transfer the contents of the buffer of the input unit specified in D32 - 35 to 16 consecutive registers, starting at register a.

Function 18 - Output Block Transfer

Transfer to the buffer of the output unit specified in D32 - 35 the contents of 16 consecutive registers, starting at register a.

Function 19 - Convert Decimal Input

(a) Punched card or magnetic tape input

Take from the buffer of the peripheral unit specified in D32 - 35 the number of characters specified in D28 - 31. Convert these characters from decimal (most significant digit first) to a binary number, and place the result in register a. Characters are taken sequentially from the buffer.

A unit of input information may be less than a complete half-bufferful (i.e. 96 characters). A ONE in D18 position is used to indicate that after the conversion specified in the instruction the next information unit is required.

(b) Punched tape input

Take from the buffer of the tape reader specified in D32 - 35 the next number, i.e. the characters up to the next "end of number" sign, convert it from decimal to a binary number and place the result in register a.

Function 20 - Convert Sterling Input

(a) Punched card or magnetic tape input

Take from the buffer of the peripheral unit specified in D32 - 35 the number of characters specified in D28 - 31. Convert from sterling to binary pence and place the result in register a.

(b) Punched tape input

Take from the buffer of the tape reader specified in D32 - 35 the next number, i.e. the characters up to the next "end of number" symbol, convert it from sterling to binary pence and place the result in register a.

Function 21 - Alphanumeric Input

(a) Punched card or magnetic tape input

Take from the buffer of the peripheral unit specified in D32 - 35 the number of characters (not more than twelve) specified in D28 - 31, and stack in registers a and a + 1 in six-bit coded form. The least significant (or right hand) character appears in the less significant end of register a + 1. Unused character positions are filled with the code symbol for 'no character' (100000).

(b) Punched tape input

Take from the buffer of the tape reader specified in D32 - 35 the characters up to the next 'end of number' sign and stack them in registers a and a + 1 in six-bit coded form. The least significant (or right hand) character appears in the less significant end of register a + 1. Unused character positions are filled with the symbol for 'no character' (100000).

Function 22 - Convert Decimal Output

Take the contents of register a and convert them to a decimal number. Transfer to the buffer of the output unit specified in D32 - 35 the number of decimal digits specified in D28 - 31, these digits being counted from the less significant end of the converted number. Not more than 12 digits may be specified.

Function 23 - Convert Sterling Output

Take the contents of register a and, regarding them as binary pence, convert to a sterling number. Transfer to the buffer of the output unit specified in D32 - 35 the number of sterling digits specified in D28 - 31, these digits being counted from the less significant end of the converted number. Not more than 12 digits may be specified.

Function 24 - Alphanumeric Output

Transfer from registers a and a + 1 to the buffer of the output unit defined by D32 - 35 a number (up to twelve) of alphabetic characters, the numbers being defined by D28 - 31 and counted from the less significant end of register a + 1.

Function 25 - Count Test

Subtract from register 7 the contents of register a. If the contents of register 7 are reduced to zero by this operation, continue with the next instruction in sequence; if register 7 is non-zero, proceed to the instruction in register b.

Function 26 - Double-length Add

Add the contents of registers a and a + 1, taken as one number with a the more significant half, to the contents of register b and b + 1 similarly regarded. Place the sum in registers b and b + 1, leaving a and a + 1 unchanged.

Function 27 - Double-length Subtract

Subtract the contents of registers a and a + 1, taken as one number with a the more significant half, to the contents of registers b and b + 1 similarly regarded. Place the difference in registers b and b + 1 leaving the contents of registers a and a + 1 unchanged.

Function 28 - Convert Sterling Output with Spaces

Take the contents of register a and, regarding them as a binary number of pence, convert to a sterling number. Insert 'space' characters between pounds and shillings and between shillings and pence. Put out the number of characters (up to 12) specified in D28 - 31 to the channel given in D32 - 35; the number of characters includes the spaces and is counted from the less significant end of the converted number.

Function 29 - Set Link and Change Sequence Instruction

Transfer the ten binary digits of the a address to the b address of register 7. Proceed to the instruction in register b.

Function 30 - Negative Sum

Commencing with register a, accumulate the number of sequential registers specified in c. Place the complement of the accumulation in register b.