SECTION 6 - PROGRAM TECHNIQUES AND MODIFICATION SYSTEM

6.1. Repetitive Sequences and Counting

One of the advantages of EMIDEC is its ability to perform a large number of simple repetitive operations in a very short time. If, however, each operation had to be specified by a separate written instruction, then the number of instructions in a program would very soon become unmanageable. The aim in writing a program is rather to set out a pattern of instructions in such a way that particular groups of instructions - known as "sequences" - may be used repeatedly, as required to deal with the data presented to the machine.

It has been demonstrated how the computer may be made to "jump", by means of a test instruction, so that the next instruction performed is not the one which follows serially in the program, but one specified by the test instruction. So far however we have considered only a forward jump, i.e. to an instruction which appears later in the program. We can however, equally well direct the machine to make a backward jump to a previous instruction, and this technique can be used so that the machine returns to carry out a particular instruction or sequence of instructions as many times as we wish. A repetitive sequence thus takes the form of a group of instructions followed by a test instruction directing the computer to return to the first instruction of the group. Such a sequence is also referred to as a "loop". The computer will continue to operate repetitively in such a loop until the test in the closing test instruction fails, when the next part of the program in series will be entered.

To demonstrate the procedure, let us take an example. Suppose we have a number 'x' in register 20, and we want to calculate the value of say x^{10}. This calculation involves repetitive multiplication by x. We therefore write:-

0	1		0		8				Clear Register 8
1	1		11		9				Put 1 in Register 9
2	9		20		9				Multiply Reg.9 by X
3									
4									

After these three instructions have been carried out we have x in register 9. Now we require to repeat instruction 2 so that x (in. Reg.9) is multiplied by x (in Reg.20) to give x^2 in register 9, then to repeat again to give x^3 in register 9 and so on. Now to return to instruction 2 we could use an "unconditional transfer" i.e. 11 0 2 but clearly since register 0 is always zero the 'loop' then formed by instructions 2 and 3 would be continued by the machine indefinitely. Provision must always be made for the computer to proceed out of the loop, and the way in which this is done must depend on the circumstances of each case. For the present example we note that we wish to perform instruction 2 ten times and then to proceed out of the loop. This introduces another common computer technique, that of counting. What we require is to count from one to ten by unit steps, performing a multiplication at each step, and then proceeding out of the loop when we have reached ten.

This we can do, as follows.

0	1		0		8				Clear Register 8
1	1		11		9				Put 1 in Register 9
2	1		0		7				Clear Register 7
3	7		11		7				Add 1 to Register 7
4	9		20		9				Multiply Reg.20 by Reg.9
5	8	R	10		7				Subtract 10 from Reg.7
6	11		7	R	9				Test Register 7 zero
7	7	R	10		7				If Reg.7 non-zero add back 10
8	11		0	R	3				Return to Instruction 3
9									Proceed if Reg.7 zero above
10	90						1	0	

This procedure is fairly straightforward to follow. The loop is contained in instructions 3 to 8. One is added each time through, to register 7, and each time through, this register is tested to see whether it has reached 10. The test, however, is clumsy since it involves adding back 10 each time after the test has been performed. The programming can be simplified by counting downwards from 10 to zero, thus:

	0	1		0		8				Clear Register 8
	1	1		11		9				Put 1 in Register 9
	2	1	R	7		7				Put 10 in Register 7
	3	9		20		9				Multiply Reg.20 by Reg.9
25	4	8		11		7				Subtract 1 from Reg.7
	5	12		7	R	3				Test Reg.7 Non-zero. Return to 3.
	6									Proceed if Reg.7 zero
	7	90						1	0	

Here 10 is first put into register 7, and then each time a multiplication is effected one is deducted from register 7, until its contents are reduced to zero, when the program proceeds out of the loop.

To effect a further saving in instructions in counting routines EMIDEC is provided with a special count-test function (Function 25) which combines the operations of subtracting from a counter register and testing that register for non-zero.

The count-test function operates only on register 7 so that to use the function it is necessary to use register 7 as the counter register.

In the last sequence, instructions 4 and 5 could be replaced by a single count-test instruction, thus:-

25	11	R3			

Here the address of the register to be subtracted from register 7 is written in the 'a' address and in the 'b' address is written the number of the instruction to which a jump is to be made if the test succeeds.

By starting our counter off at 10 and subtracting one at each operation of the loop we have been able to use the count test. In some cases it may be necessary to count upwards. It is however, still possible to use the count test by first putting a negative value in the modifying register.

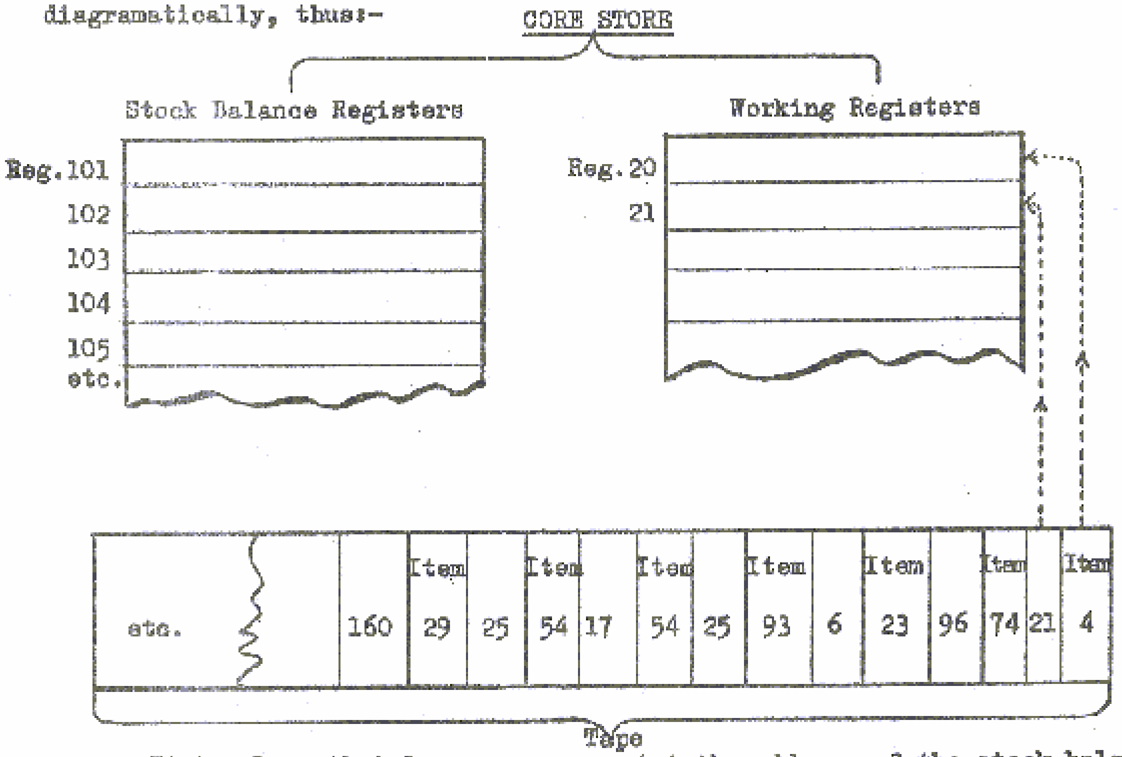
0	1		0		8				Clear Reg.8
1	1		11		9				Put 1 in Reg.9
2	1	R	7		7				Put -10 in
3	9		20		9				Multiply Reg.9 by x
4	25	R	8	R	3				Subtract -1.Test Reg.7
5									Proceed
6									
7	90						-1	0	
8	90						-	1	

By this method the counter is progressively increased from -10 to zero. The count for the number of times that the loop has been performed can then be calculated at any time it is required by adding the counter register to a constant 10.

6.2. Modification

In writing a program we may find that we wish to refer to a particular register whose address cannot be specified either absolutely or relatively, since the address depends upon the data to be read by the program.

To illustrate the position, let us take as a simple example a case where it is required to update balances of stock items which are coded from 1 to 100, the balances being held, in stock code number order, in registers 101 to 200. Details of receipts are read in from punched tape, the information for each receipt consisting of the item number and the quantity. These details are in random order on the tape. The position may be considered diagramatically, thus:-



It is clear that for any one receipt the address of the stock balance to be updated will depend entirely upon the contents of the tape read in, and that the address may be different for each receipt. We cannot therefore write the address as a permanent part of the programme. What is required in such circumstances is some means of writing a basic instruction to affect updating, with a facility for altering the instruction in the course of the actual operation of the job, so that it affects whatever address is appropriate to the data being used.

Suppose that the instructions for reading each item from the paper tape place the code number in register 20 and the receipt quantity in register 21. Now we require to add the receipt quantity in the register corresponding to the code number. The basic form of the add instruction would be:-

7		21		100 + i		

Where "i" is the item code number which is required to be added to 100 to give the destination address. Since instructions are held in registers in binary form, it is possible to do arithmetic work on them, as if they were numbers. The instruction 7/21/100, would be stored in a register in binary as:-

											1	1	0	0	1	0	0						1	0	1	0	1			1	1	1			

The 'b' address here contains the binary for 100, and we want to add it to the binary equivalent of "i", the code number of the particular item with which we are dealing. Now "i" will be contained in register 20, and will appear in the form:-

																															1	0	1	1	1

We could therefore achieve our aim by shifting the contents of register 20 eighteen places to the left, so that they are in the 'b' address position, and then adding register 20 to the instruction register, so that we get 100 + i in the 'b' address of that register. To program such a procedure we would write:-

0	2		20		20	18			Shift Reg.20 left 18
1	7		20	R	2				Add Reg.20 to Inst.2
2	7		21		100				Add Reg.21 to Reg.100 (now amended)

While, however, this procedure would work for the first item it could not be repeated, because the contents of the 'b' address of instruction 2 have now been altered to some unknown value. To make repetition possible it would be necessary to restore the original value of instruction 2 by subtracting register 20. If this were done, a sequence could be built up as follows:-

0									Read item into Regs.20-21
1	2		20		20	18			Shift Reg.20, 18 places left
2	7		20	R	3				Add Reg.20 to instruction 3
3	7		21		100				Add Reg.20 to Reg.100 +
4	8		20	R	3				Subtract Reg.20 from Inst.3
5	11		0	R	0				Return to read another
6									item into Regs.

Now in EMIDEC, in order to improve upon the above procedure where a number is added to an instruction and then re-subtracted, a special modification device has been built which enables an instruction to be amended during operation without effecting any alteration of the stored program.

In EMIDEC, registers 1 - 7 are designated as modification registers and the contents of the appropriate one of these registers are, in operation, added to the contents of an instruction register which has been designated, in writing the instruction, as being subject to modification. It has been mentioned that the final column of a coding sheet has been provided to hold a modifier and it is in this column that the appropriate modification register is specified. In fact, register 0, which is the source of zero should also properly be considered as a modification register, since the computer always carries out the modification procedure and when the modifier column is left blank the machine applies a zero modifier which, of course, leaves the instruction unaffected. It may logically be said, therefore, that there are eight modification registers from 0 - 7.

Reverting to our example, we would program the sequence using modification, as follows:-

0	2		20		3	18			Shift Reg.20 left 18 to Reg.3
1	7		21		100			3	Add Reg.21 to Reg.100 (Modified)
2	11		0	R	⟶				Return
3									
4									

Here instruction 0 shifts the code number into the 'b' address in register 3, one of the modifying registers. The basic add instruction is then written as instruction 1 with 3 in the modifier column. The effect of this is that, after the instruction has been selected, the contents of register 3 are added to it, so that the code number is added to the 'b' address section which specifies the location holding the balance to be updated, and the appropriate destination is therefore given for the adding operation.

It should be clearly understood that the contents of the modification register are NOT added to the modified register IN THE STORE. The computer control centre reads an instruction from the store in order to perform it, and it is only IN OPERATION, at the point in the computer control when the instruction has been read from the store, that the contents of the modifying register are added. The stored instruction remains quite unaffected.

As another example suppose we have a list of prices held in descending order of magnitude from register 101 upwards and we wish to transfer them into registers 201 upwards. We can assume that in this case the sequence will end when we reach a register in the 100 section which contains a zero value.

In this sequence the basic instruction is a transfer, but we do not know absolutely either the 'a' address or the 'b' address. We know, however, that they will commence at 101 and 201 respectively, and will increase by one each time the basic instruction is performed. The programmed sequence would be:-

	0	1		0		1				Clear Reg.1
	1	1		0		2				Clear Reg.2
5 ⟶	2	1		101		201			2	Transfer (Modified)
	3	7		12		1				Add 1 in 'a' address to Reg. 1
	4	7	R	8		2				Add 1 in 'a' & 'b' addresses to Reg. 2
	5	12		101	R	2			1	Test 101 etc. non-zero. Proceed
	6									to next sequence.
	7									
	8			1		1				

Here we are using two modifying registers 1 and 2. The transfer instruction is modified by the contents of Register 2 so that the numbers of the source and destination of transfer are each progressively increased by "one". Register 1 is used to modify the non-zero test instruction so that as the transfers proceed the A section registers are progressively tested, and when one of them is found to be zero the sequence is ended. As the sequence proceeds the modifiers accumulate, but the basic instructions 2 and 5 remain unchanged.

Such a progressive increase in the modifiers is used for all kinds of repetitive sequences. Suppose, for example, we have ten numbers in registers 101 to 110 and we wish to obtain their sum in register 200. This can be done by repetitive adding and to detail the necessary operations we would write:-

0	1		0		200				
1	7		101		200				
2	7		102		200				
3	7		103		200				
4	7		104		200				
5	7		105		200				
6	7		106		200				
7	7		107		200				
8	7		108		200				
9	7		109		200				
10	7		110		200				

Here we have ten instructions basically similar, but varying at each operation by a fixed amount. If the summation of several hundred registers had to be written in this way the programming would become very laborious. The basic instruction of the sequence is, however, to add into register 200 commencing from register 101.

All we require is for the machine to repeat this instruction, but with an amendment, each time, of the 'a' address, so we modify it as follows:-

0	1		0		6				Clear Reg.6
1	1		0		200				Clear Reg.200
2	7		101		200			6	Add into Reg.200 (Modified by Reg.6)
3	7		12		6				Add 1 in 'a' address to Mod.Reg.6
4	11		0	R	2				Jump to Instruction 2.

By this procedure the register holding instruction 2 is never altered, but the 6 in the final column directs the computer to add the contents of register 6 to the instruction after it has been read by the control unit. Register 6 is cleared at the beginning of the program, so that the first time instruction 2 is performed a nil modifier will be applied and the 'a' address will be 101 as written. Then however, a digit in the 'a' address is added, not to the instruction itself, but to the modifier. The jump is then made back to instruction 2 and this time, though the instruction itself remains unchanged it will in performance be supplemented by the contents of the modifying register 6 which now contains 1 in the 'a' address, so that the register addressed will be 102. The operation will then be repeated for registers 103, 104 and so on, the amount in the modifying register being progressively increased by one, while the basic instruction remains unchanged.

In the example shown above register 101 is added, then 102, then 103 and so on. It may have been noted, however, that the program as written would cause the computer to go on indefinitely adding consecutive registers into register 200. To correct our example, we require to add into register 200 ten times and then leave the sequence. This we may do by setting up a counter of 10 in say, register 50.

0	1	R	8		50				Constant 10 in Reg.5
1	1		0		6				Clear Reg. 6.
2	1		0		200				Clear Reg.200
3	7		101		200			6	Add to Reg.200 (Modified by Reg.6)
4	7		12		6				Add one to modifier
5	8		11		50				Subtract one from counter
6	12		50	R	3				Test counter
7	0								
8	90						1	0	

In this sequence, each time the basic add instruction is performed one is deducted from the counter in register 50 and it is tested for non-zero. After ten add operations therefore the counter becomes nil, the test fails and the program proceeds out of the sequence.

It will be seen that, in the above sequences, for each repetition of the basic instruction we had to add one to the modifier and subtract one from the counter. At the end of the ten operations the modifier will have become 10 and the counter zero. We can however, accumulate the registers in the reverse order, starting at register 110, by commencing with a modifier of 10 and reducing it progressively by one, and if we do this we can save instructions by using the same register both for modifying and for counting:-

0	1	R	9		6				Constant 10 in 'a' address in Reg.6
1	1		0		200				Clear Reg.200
2	7		100		200			6	Add to Reg.200
3	8		12		6				Subtract one from Reg.6
4	12		6	R	2				Test Reg.6 Return to Inst.2

Here register 6 is used both as modifier and counter. The first basic add instruction is performed on register 100 modified by 10 i.e. register 110, then on register 109, then 108 and so on until register 6 is reduced to nil, when the program proceeds out of the sequence. It should be especially noted that as soon as register 6 becomes zero the sequence is finished and no return is made to the basic add instruction. Therefore at the time when the last add instruction is performed register 6 contains a one in the 'a' address. For this reason the 'a' address in instruction 2 is written as 100 not 101, for the last instruction performed will be on the basic address plus the one remaining in the modifier, so that register 101 is, in fact, addressed. No operation is performed on register 100 because a nil modifier is never applied in this sequence, which is finished as soon as the modifying register reaches the value nil.

Such a sequence can be especially advantageous because the modifying register is left clear and can therefore be used again at a subsequent point in the program without needing clearing as a special operation.

It has been seen that, by starting at the highest register and counting down, we can combine the modifying register with the counting register. It is however, also possible to combine the modifier with the count even when counting upwards, by using the device of a negative number in the counter register.

	0	1	R	6		6				Constant -10 in 'a' address Reg.6
	1	1		0		200				Clear Reg.200
	2	7		111		200			6	Add to Reg.200
25	3	7		12		6				Add 1 to Reg.6
	4	12		6	R	2				Test counter
	5	0								
	6			-10						

This same technique could also be used with the count-test function 25 by employing register 7 instead of register 6 as modifier and by progressively subtracting in the count test instruction itself a constant of -1, this being equivalent to the progressive addition of 1.

It has been shown that by modification the number of program instructions can be considerably reduced, and this technique would in practice be used in almost all cases of the type illustrated in the last example. The programmer should not however, lose sight of the fact that modification is primarily a device for reducing the labour of program writing, but that it does not reduce the amount of work - and therefore time - required of the machine. Whether modification is used or not the machine must repeat the basic operation for each register being dealt with, and the operation and setting up of the modifying procedure must, of course, take additional time. In the last example 32 program steps are carried out by the computer to obtain the required result. The same result could be obtained without modification by a simple program of repetitive adding as previously shown.

We should then need one instruction to clear Reg.200 and ten addition instructions, making only eleven steps in all, taking only 1/3 of the time of the modified procedure.

The additional time taken by the machine is usually justified by the saving in programming effort, but this cannot be stated as an invariable rule, and the basic principles should be borne in mind so that the appropriate course may be followed in any particular case.

Switching

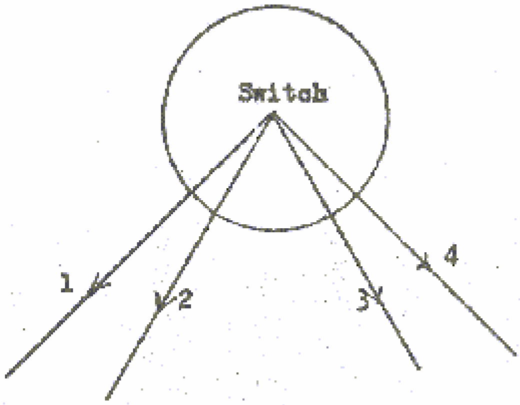
Although variation of instructions is most conveniently and most commonly effected by modification, none the less, it is still possible to do arithmetic on the instruction registers in the way shown on page 5 and this technique may in certain cases be the most appropriate. Not only may instructions be varied in this way, but they may in fact be replaced completely by a different instruction, thus causing a complete change in the program sequence. This substitution of instructions is known as "switching". As an example of switching let us consider the case where we are reading in from different card types, numbered 1 to 4, and for each card we have a different routine to follow. Suppose these routines commence at instructions R27, R41, R52 and R57 respectively and that the card type has been read into register 2, in the 'A' address. Then we would program as follows:

0	1	R	2	R	1			2	Set up switch
1	[0							]	Switch
2	0								
3	11		0	R	27				
4	11		0	R	41				
5	11		0	R	52				
6	11		0	R	57				
7									
8									
9									

Here instruction R 0 transfers into register R1, the contents of R2 + modifier. Thus if the card type is, say, 1 the 'a' address of instruction R 0 will become R 3 and R 1 will be replaced by R 3 so that the next instruction carried out is equivalent to the contents of R 3 i.e. a jump to R27. Similarly if the card type is 3 then instruction R 1 is replaced by R 5 which effects a jump to R 52. So instruction R 1 becomes a "Switch" to the sequence appropriate for the card type.

It should be pointed out that the instructions in R 3 to R 6 are not an integral part of the program, but are instruction constants which are only used if and when they are inserted at R1 by the operation of R 0.

In drawing flow charts and diagrams of jobs, a test function is shown as a box with two alternative flow-lines proceeding from two of its sides. A switch, however, is usually shown as a circle with the appropriate number of alternative flow-lines radiating from it, thus:


Switch
1
2
3
4