CHAPTER 12

PROGRAMMING

The methods used for programming a calculator so that it will proceed through the desired sequences of operations depend greatly on the way in which the calculator has been organized. In particular, programming methods are quite different for externally programmed, plugged-program, and stored-program machines. While interesting and important techniques have been worked out for certain externally programmed and plugged-program machines, the outstanding advances in programming methods have been made in connection with calculators of the stored-program variety. For this reason, the term, "programming," frequently implies that the program is stored. Further, it is generally assumed that with a stored program the calculator will be able to perform arithmetic and other operations on instructions in the program as well as on items of data, although a few stored-program machines have been built where the program and data have been kept separate. In this chapter, the subject will be confined to stored-program machines that store instructions and data interchangeably.

For each stored-program calculator there is a list of instructions that the calculator is capable of executing. When preparing a program to solve any given problem, the programmer must be familiar in some detail with the steps the calculator will take when following each instruction in the list. While the instruction lists of most calculators contain certain basic instructions such as add, multiply, shift, and others, the details in the steps taken to execute the instructions can vary considerably from one machine to the next. Also, there are many miscellaneous instructions which may or may not be incorporated into any given machine. Because of the extensive variations which may be found in the instruction lists, the program used to solve a given problem on one machine may seem to bear little resemblance to the program used to solve the same problem on another machine. However, the same principles of programming can be applied to practically all calculator organizations where the instructions are stored interchangeably with the data. These principles will be explained and illustrated through the use of a "specimen" machine. The instructions in the instruction list for the specimen machine have been chosen mainly for their usefulness in illustrative programs, but at the same time there has been an attempt to make the list realistic.

Instruction List for the "Specimen" Machine. A 5-digit, single-address, decimal, organization has been chosen for the "specimen" machine. Each word will consist of five decimal digits with sign, and any word may represent either an item of data or an instruction. When a word is used to represent an instruction, the first two digits will serve as a code to indicate the operation to be performed, and the last three digits will represent the address; the sign will not be used in the case of an instruction. Normally, the calculator will start by executing the instruction found at address 000 and will continue by executing instructions found at sequentially numbered addresses except when a jump-type of instruction is encountered. Besides the one thousand storage locations (designated by addresses 000 through 999) in the main storage unit, there will be one other storage register, called the accumulator, which will enter into the various operations as described in the instruction list. A knowledge of the material in the previous chapter, which explains how a calculator can be made to proceed through a sequence of instructions, should be helpful but not necessary.

Instruction	Code	Description
STOP	00	The calculator stops regardless of the digits in the address part of the instruction.
RESET AND ADD	01	The accumulator is reset to zero, and the number at the indicated address is then placed in the accumulator.
ADD	02	The number at the indicated address is added to the number in the accumulator, and the sum is left in the accumulator.
SUBTRACT	03	The number at the indicated address is subtracted from the number in the accumulator, and the difference is left in the accumulator.
MULTIPLY	04	The number at the indicated address is multiplied by the number in the accumulator, and the product is left in the accumulator. (This instruction is somewhat irregular in that products of more than five digits will not be possible.)
DIVIDE	05	The number in the accumulator is divided by the number at the indicated address. The quotient is left in the accumulator, and the remainder is lost.
SHIFT RIGHT	06	The number in the accumulator is shifted to the right a number of places indicated by the number in the address part of the instruction. Digits shifted to the right from the units position are lost.
SHIFT LEFT	07	The number in the accumulator is shifted to the left a number of places indicated by the number in the address part of the instruction. Digits shifted to the left from the tens thousands position are lost.
STORE	08	The number in the accumulator is placed in the storage location indicated by the address part of the instruction. The previous contents of this storage location are lost; the number in the accumulator is unchanged.

Instruction	Code	Description
STORE ADDRESS	09	Same as STORE except that the number in the accumulator is assumed to be an instruction, and only the three digits corresponding to the address part of the word are sent to storage. The digits corresponding to the code in both the accumulator and address location are uneffected.
JUMP	10	The next instruction is taken from the address indicated by the address part of this instruction instead of the next sequentially numbered address position.
JUMP IF MINUS	11	Same as JUMP if the number in the accumulator at the time is negative (zero is assumed to be positive); otherwise, the next instruction is taken from the next sequentially numbered address position.
READ	12	One word from the input mechanism is placed in storage at the indicated address. The previous number at this address is lost.
PRINT	13	The word at the storage location indicated by the address is recorded by the output mechanism. The number in storage remains unchanged.

Basic Programming Technique. To illustrate the basic procedure which is used to cause a calculator to proceed through a sequence of operations, the instructions required for the "specimen" calculator to calculate $xy + z^2$ and print the result will be shown. Assume that means have been provided for placing the program and the data in the main storage unit of the calculator. The first instruction will appear at the storage location designated by address 000, and the other instructions are placed in the storage locations designated by successively higher numbered addresses. The data, x, y, and z, may be stored at any convenient addresses not needed for instructions; in this example, locations 090, 010, and 011 will be used. Recall that when a storage location is used for storing an instruction, the sign is not used, the first two digits represent the operation in coded form, and the last three digits indicate the address part of the instruction. All five digits with sign are used as one number when storing an item of data.

Address	Contents of Address		Remarks
000	RESET AND ADD	009	Places x in the accumulator.
001	MULTIPLY	010	Forms xy in the accumulator.
002	STORE	012	Places xy at address (location) 012 for temporary storage.

Address	Contents of Address		Remarks
003	RESET AND ADD	011	Places z in the accumulator.
004	MULTIPLY	011	Forms z^2 in the accumulator.
005	ADD	012	Forms $xy + z^2$ in the accumulator.
006	STORE	012	Places $xy + z^2$ at address 012 for temporary storage. (The previous contents of 012 are lost.)
007	PRINT	012	The number representing $xy + z^2$ is printed.
008	STOP	---	Calculator stops. The address part of this instruction is of no consequence.
009	x		
010	y		
011	z		
012			Reserved for temporary storage.

In the above program the purpose and function of each instruction is explained through the comments in the "remarks" column. Several different variations in the program may be worked out. One minor variation is that location 012 need not be reserved as a place for the temporary storage of intermediate results. Location 009, for example, would serve this purpose just as well, because after the first instruction x does not enter into the problem again so that location 009 is no longer needed for its storage.

Note the desirability of stopping the calculator on step 008. If the calculator is not stopped, it will automatically proceed to interpret the items of data as instructions. Since the digits of x, y, and z could represent any of the instructions, all manner of unpredictable and unwanted operations might result.

Elementary Logical Program. In the example of the previous section, the calculator proceeds uniformly through sequentially numbered address locations to obtain its instructions. In many problems it is necessary or desirable to alter the sequence of operations in accordance with the data. A simple illustration of a problem of this type is the finding of the largest of the three numbers, x, y, and z, and printing the result. Such a problem is more a problem of logic than a problem of arithmetic, although certain arithmetic operations are employed in its solution. When the calculator compares two numbers (by subtracting one from the other) JUMP instructions are used to cause one sequence of instructions or another to be followed in accordance with which of the two numbers was the larger. The finding of a program which employs as few instructions as possible is an intriguing puzzle; a program using one or two less instructions than certain "obvious" programs will be explained. As before, the instructions will be found in consecutive address locations, but in this case the calculator will not necessarily follow them in sequence because of the JUMP instructions. The locations, 013, 014, and 015, will be used for storing x, y, and z, respectively, and location 016 will be used for temporary storage.

Address	Contents of Address		Remarks
000	RESET AND ADD	013	Places x in the accumulator.
001	SUBTRACT	014	Subtracts y from x.
002	JUMP IF MINUS	005	Calculator takes next instruction from address (location) 005 if y > x; otherwise, it proceeds to 003.
003	RESET AND ADD	013	Places x in the accumulator again.
004	JUMP	006	Causes calculator to take next instruction from 006.
005	RESET AND ADD	014	Places y in the accumulator.
006	STORE	016	Note that the calculator will arrive at this step from 005 when y > x, but from step 004 when y < x with the result that the accumulator contains the larger of x and y. This number is stored in 016.
007	SUBTRACT	015	Subtracts z from the larger of x and y.
008	JUMP IF MINUS	011	If z is largest, next step is taken from 011; otherwise, from 009.
009	PRINT	016	The larger of x and y (which is now known to be the largest of the three) is printed.
010	JUMP	012	Causes calculator to take next instruction from 012.
011	PRINT	015	Causes z to be printed. The calculator arrives at this step only when z is largest.
012	STOP	---	Calculator stops

Address	Contents of Address	Remarks
013	x	
014	y	
015	z	
016	[ ]	Reserved for temporary storage.

Again, several variations in the program are possible. For example, a STOP instruction instead of a JUMP instruction could have been placed in location 010. With this change, the calculator would stop on step 010 or 012 depending on the relative sizes of x, y, and z. In this elementary example the change would be trivial, but in cases where a program of this type is a part of a larger program, the termination of the part is more likely to be of consequence.

Program Loops. In many problems of a highly repetitive nature, a program prepared in a "straightforward" manner would consume an unduly large number of instructions. Through the use of JUMP instructions it is possible to prepare a relatively short program (program "loop") through which the calculator will proceed over and over again the desired number of times. The loop may comprise substantially the entire program, but in most practical examples a loop would pertain to only a small portion of it. Any one program may contain many loops which may interlock one another in a complex manner. As a simple example of a loop, a program for preparing a list of the squares of the intergers from 1 through 125 will be presented.

Address	Contents of Address		Remarks
000	RESET AND ADD	010	The number at address (location) 010 is increased by 1, (from +00000 to +00001 the first time through the loop).
001	ADD	011	
002	STORE	010	
003	MULTIPLY	010	The number (+00001 the first time) is squared and printed.
004	STORE	013	
005	PRINT	013	
006	RESET AND ADD	010	The quantity, +00125, is subtracted from the number with the result that the program will be repeated until the number is increased to +00125.
007	SUBTRACT	012	
008	JUMP IF MINUS	000	
009	STOP	---	
010	[+00000]		
011	+00001		
012	+00125		
013	[ ]		

In this program, storage locations 010, 011, and 012 were used for the storage of constants which were not data in the usual sense of the word. The brackets around the number in 010 signify that this particular quantity is changed during the course of the calculations. Note that in this example it was not necessary to store all of the numbers from 1 through 125 as data; instead, it was possible to generate them by means of the program.

Modification of Instructions. As has been mentioned, one of the outstanding features of most stored-program calculators is their ability to modify an instruction (particularly the address part of an instruction) by mans of the same arithmetic circuits that are used for performing calculations on the data. A frequently encountered application of this facility is the performing of the same operation on a series of numbers which are located in sequentially numbered addresses. For example, assume that it is desired to accumulate and print the sum of the squares of a series of numbers, x_1, where i varies from 1 to 100. Assume, also, that these numbers are stored in addresses, 300 through 399, inclusive. One program which may be used for this purpose is shown below. Again, brackets are used to signify quantities that will be changed during the course of the program.

Address	Contents of Address		Remarks
000	RESET AND ADD	[300]	These two program steps cause the square of the specified number to be placed in the accumulator. The first time through the loop the specified number is the one found at address 300.
001	MULTIPLY	[300]	
002	ADD	016	Location 016 is used to store the accumulated sum of the squares.
003	STORE	016	
004	RESET AND ADD	018	In 018, a number representing the instruction RESET AND ADD 398 is stored, and this number is placed in the accumulator. Recall that the code for RESET AND ADD is 01.
005	SUBTRACT	000	The number in 000 (which is an instruction) is subtracted. The difference will not be minus unless the address part of the number in 000 has been increased to 399.
006	JUMP IF MINUS	014	
007	RESET AND ADD	000	The address part of the number (instruction) in 000 is increased by 1.
008	ADD	017	
009	STORE	000	
010	RESET AND ADD	001	The address part of the number (instruction) in 001 is increased by 1.
011	ADD	017	
012	STORE	001	

Address	Contents of Address		Remarks
013	JUMP	000	The process will be repeated. Each time through the loop the next successive number in the series will be squared and accumulated because of the altered addresses of the instructions in 000 and 001.
014	PRINT	016	The calculator will arrive at step 014 (from 006) only after all squares have been formed and accumulated.
015	STOP	---	
016	[+00000]		
017	+00001		
018	+01398		

An important extension of programs of this type is in generalizing them so that they will function properly by merely inserting one number representing the length of the series. One variation in the ways by which the preceding program can be generalized will be presented. The number representing the first address of the series (300 in this example) is placed in address 023 and the number representing the length of the series (100 in this example) is placed in address 024. Address 024 is used as a counter; the number stored there is reduced by 1 each time the program loop is traversed, and when the count becomes negative, the process is terminated.

Address	Contents of Address		Remarks
000	RESET AND ADD	023	The address parts of the instructions in 010 and 011 are set equal to the number in 023.
001	ADD	010	
002	STORE	010	
003	RESET AND ADD	023	
004	ADD	011	
005	STORE	011	
006	RESET AND ADD	024	The number in the counter is reduced by 1.
007	SUBTRACT	026	
008	STORE	024	
009	JUMP IF MINUS	021	
010	RESET AND ADD	[000]	The square of the appropriate number is formed and accumulated with the sum being placed in 025.
011	MULTIPLY	[000]	
012	ADD	025	
013	STORE	025	
014	RESET AND ADD	010	The address parts of the instructions in 010 and 011 are increased by 1.
015	ADD	026	
016	STORE	010	
017	RESET AND ADD	011	
018	ADD	026	
019	STORE	011	

Address	Contents of Address		Remarks
020	JUMP	006	The program is repeated, except for the first group of instructions, which are needed only once.
021	PRINT	025	The calculator will arrive at 021 (from 009) only after all 100 squares have been formed and accumulated.
022	STOP	---	
023	+00300		
024	[+00100]		
025	[+00000]		
026	+00001		

With this particular arrangement one detail that should not be overlooked when the program is used a second time is that the address parts of the instructions in 010 and 011 and the number in 025 should be reset to zero. In this "specimen" machine the number in 025 would be reset to zero if the following three instructions were inserted at the beginning of the program.

```
RESET AND ADD     025
SUBTRACT          025
STORE             025
```

The address part of 010, for example, could be reset to zero by using the SHIFT instructions as follows.

```
RESET AND ADD     010
SHIFT RIGHT       003
SHIFT LEFT        003
STORE             010
```

The STORE ADDRESS instruction is of considerable value in programs of the above type. A substantial reduction in the number of program steps can be achieved, and the need for resetting to zero the address parts of the instructions in 010 and 011 in the illustration is eliminated. The required alterations in addresses can be achieved by substituting the following five instructions in place of the first six instructions in the illustration and by eliminating instructions 014 through 019.

000	RESET AND ADD	023	The address parts of the instructions in 010 and 011 are replaced by the number in 023, and that number is increased by 1.
001	STORE ADDRESS	010	
002	STORE ADDRESS	011	
003	ADD	026	
004	STORE	023	
023	--300		The first two digits are of no consequence.

The JUMP instruction would be altered to include this group of instructions in the loop, and, of course, a renumbering of all instructions would be necessary.

Sub-programs. When a sequence of instructions is to be used frequently, but not in the uniformly recurring manner of the previous examples, a technique known as "sub-programming" is advantageous. By preparing the sequence of instructions as a sub-program it is possible to arrange the "main" program so that the calculator will jump to the sub-program, perform the desired standard sequence of operations, and then return to the main program. Actually, the summing of the squares of a series of numbers when considered as an integrated operation might be a good example of a program to be made into a sub-program if this operation is required frequently. However, another example has been chosen to illustrate sub-programming because it is somewhat less complex, and also because some other interesting points of programming are introduced incidentally. The sub-program to be explained is one for calculating the square root of a number. The calculation of the square root is required frequently in many problems. If the sequence of instructions for extracting the square root were inserted in the main program each time it was needed, an unduly large number of storage locations in the calculator would be consumed. From the standpoint of storage space it is preferable to write the square root program once as a sub-program, and use the relatively few instructions which are required to refer to it each time a square root is needed.

Before proceeding to the example itself, a useful notation relating to addresses will be explained. Often when preparing programs it is inconvenient to specify the exact address to be used in each and every instruction. Instead, the notation, L(x), which signifies the address (location) where x is stored whatever that address might be, may be used. With this definition, an instruction such as ADD L(+00003) means that 3 should be added to the number in the accumulator. Of course, before the program is executed by the calculator the actual number representing the address, $a = L(x)$, must be inserted in the address part of the instruction by the programmer either directly or by means of other instructions in the program. Also, it is frequently convenient to refer to the number stored at a given address by the notation, C(a), which means the "content" of the address (location). If x is stored in a, $C(a) = x$. To appreciate more fully the meaning of the notation, observe that the equations,

$$L\left[C(a)\right] = a$$

$$C\left[L(x)\right] = x,$$

follow directly from the definitions.

For calculating the square root, the iterative formula,

$$b_{k+1} = 1/2\,(b_k + \frac{x}{b_k}),$$

will be used with +00317 (the largest possible square root in a five-digit machine) as the first approximation, b_0. Successive approximations will decrease monotonically toward the desired value. The constants, +00317 and +00005, may be stored as appendages to the sub-program, or they may be at some other known storage locations. Since the square root is to be calculated by an iterative process, the sub-routine will contain a loop, and this loop will be traversed repeatedly until $b_{k+1} = b_k$. Assume for purposes of illustration that the first instruction of the sub-program is at address 350. The number, x, for which the square root is desired, is at address 363 and the b_k are stored at 364.

Address	Contents of Address		Remarks
350	RESET AND ADD	L(+00317)	The number for b_0 is placed in the accumulator.
351	STORE	364	The approximation, b_k (b_0 the first time through the loop) is placed in 364.
352	RESET AND ADD	363	The quantity, $b_k + x/b_k$, is formed in the accumulator.
353	DIVIDE	364	
354	ADD	364	
355	MULTIPLY	L(+00005)	A division by 2 with a rounded quotient is effected by multiplying by 5, adding 5 in the units position, and dividing by 10 (shifting right). The accumulator now contains b_{k+1}.
356	ADD	L(+00005)	
357	SHIFT RIGHT	001	
358	SUBTRACT	364	If $b_{k+1} < b_k$, the loop, starting with instruction at 351, is to be repeated. Instruction at 361 is needed to restore the contents of the accumulator to b_{k+1}.
359	JUMP IF MINUS	361	
360	JUMP		Sub-program returns to main program.

Address	Contents of Address		Remarks
361	ADD	364	See remarks above.
362	JUMP	351	
363	x		
364	b_k		

In order for the main program to be able to use a sub-program, two important conditions must be satisfied. Any parameters of the sub-program, such as x in this example, must be placed where the sub-program can find them, and the proper address must be placed in the JUMP instruction which causes the return to the main program. The instructions which provide for the entry into a sub-program and the return from it are commonly referred to as the "linkage." In particular, the JUMP instruction which causes the return from the sub-program to the main program (the one at address 360 in this example) is called the "link" instruction. As an example of the use of the sub-program, assume that the main program has arrived at a step where the next instruction is to be taken from address 100, and it is desired to calculate the square root of number in address 750. The square root sub-program may be utilized by the following instructions in the main program.

Address	Contents of Address		Remarks
100	RESET AND ADD	750	C(750) is placed at 363, where
101	STORE	363	the sub-program can find it.
102	RESET AND ADD	105	The number representing the link
103	STORE	360	instruction is obtained from 105 and is placed at the appropriate address in the sub-program.
104	JUMP	350	Calculator jumps to the sub-program.
105	JUMP	106	See above remarks.
106 . .	Continuation of main program		The sub-program returns the calculator to this step. The square root of C(750) is now at 364.

With above method of linkage, all steps in the placing of the link are accomplished by the main program, and new instructions for this purpose are required each time the sub-program is used. By making use of the facility that the operation part as well as the address part of an instruction can be modified by the program it is possible to employ two less instructions in the main program for each entry into the sub-program at the expense of two instructions at the beginning of the sub-program and one special constant. When a sub-program is used many times in one program, a considerable saving in storage space can be achieved in this way.

Address	Contents of Address		Remarks
100	RESET AND ADD	750	These two instructions are the
101	STORE	363	same as before.
102	RESET AND ADD	102	The number, 01102, is placed in
103	JUMP	348	the accumulator. The next instruction is taken from 348, which is the new beginning of the sub-program.
104	Continuation of main program		The sub-program returns the calculator to this step.
- - - - -	- - - - - - - -	- - -	- - - - - - - - - - - -
348	ADD	L(09002)	The sum, 01102 + 09002 = 10104,
349	STORE	360	which equivalent to JUMP 104, is formed in the accumulator and then placed in 360 as the link instruction.
350	The main body of the square root sub-program is the same as before.		
.			
.			
.			
364			

By using the STORE ADDRESS instruction the linkage can be made equally conservative of storage space and possibly somewhat more straightforward. In this example different addresses will be used for the linkage instructions in the main program to emphasize that the link instruction in a sub-program must provide for return to the "next consecutive" step in the main program regardless of the location of the instruction that caused the jump to the sub-program. Assume that the calculator has arrived at step 620 and that the square root of the number at address 506 is desired.

Address	Contents of Address		Remarks
620	RESET AND ADD	506	These two instructions are the same in principle as before.
621	STORE	363	
622	RESET AND ADD	622	The number, xx622, (the first two digits are irrelevant in this case) is placed in the accumulator, and next instruction is taken from 348.
623	JUMP	348	
624	Continuation of main program.		The sub-program returns the calculator to this step.

- -

348	ADD	L(00002)	The sum, xx622+00002 = xx624 is formed in the accumulator. The address part of this number is placed in 360, which already contains the code for JUMP in the operation part.
349	STORE ADDRESS	360	

350
.
.
.
364 } The main body of the square root sub-program is the same as before.

In all of the linkage examples which have been cited the same method was used for placing the parameter where the sub-program could find it. Another frequently used method for locating the sub-program parameters is to place them immediately after the instructions that cause the jump to the sub-program. Then, if x is the number for which the square root is desired, a linkage arrangement employing the STORE ADDRESS instruction as illustrated in the next example could be used. Prior to the arrival of the calculator at step 622, x must be placed at 624 by instructions not shown. Note that in this case, the link must provide for a return to the main program at a step that is 2, instead of just 1, steps beyond the JUMP instruction which caused entry into the sub-program.

Address	Contents of Address		Remarks
622	RESET AND ADD	622	Same as corresponding instructions in previous examples except that the first instruction of the sub-program is now at 346.
623	JUMP	346	

Address	Contents of Address		Remarks
624	x		
625	Continuation of main program.		

- -

346 347	ADD STORE ADDRESS	L(00002) 352	The sum, --622 + 00002 = --624, is formed in the accumulator. The address part is placed in 352, which contains the instruction involving the location of x.
348 349	ADD STORE ADDRESS	L(00001) 360	The sum, --624+00001 = --625, is formed, and the address part is placed in the link instruction.
350 . . . 364	The sub-program is the same in principle as before. Changes in details will result from the fact that address part of the instruction in 352 will be altered by the program and from the fact that storage location 363 will no longer be needed for the storage of x.		

In some of the above examples, the signs of certain numbers which relate to instructions have been disregarded in cases where only the magnitudes were of primary consequence. Actually, for the program to function properly, the signs of these numbers must be chosen correctly. A more detailed discussion of signs will be omitted because it would most likely add confusion without aiding in the explanation of programming principles. In many calculators special "magnitude" instructions are provided which facilitate arithmetic operations involving only magnitudes and not signs.

Library of Sub-programs. Many sub-programs are likely to be useful in more than one problem. The need for extracting the square root, for example, is encountered in a wide variety of problems. Much programming effort can be saved if sub-programs can be retained in a "library" of some sort from which they can be "withdrawn" and inserted in other programs as needed. The outstanding factor to be considered in the formation of a library arises from the need for being able to place a sub-program at any set of consecutive addresses in calculator storage instead of one fixed set such as 348 to 364.

Assume, for example, that in the preparation of the program for the solution of some problem the square root is needed, and addresses 348 to 364 happen to have been employed for the storage of other instructions or data. The square root sub-program can be moved to some other part of the storage, say addresses 448 to 464, by adding 100 to the address where each instruction of the sub-program is stored and by adding 100 to the address part of each sub-program instruction which refers to another part of the sub-program. Note that in the square root sub-program the address parts of some instructions, particularly the SHIFT instruction should be the same regardless of where the sub-program is stored. The address parts of the instructions involving constants should also remain unchanged if special addresses are assigned for the storage of constants. On the other hand, if the constants are stored as part of the sub-program, the instructions which refer to the constants must be altered by adding 100 to the address parts.

Since the altering of a sub-program to fit it into any set of consecutive addresses in storage is a routine job, it is possible to prepare a "positioning sub-program" which will make it possible for the calculator to relieve the programmer of this work. However, one difficulty of consequence is encountered. The positioning sub-program must, by some means or other, be able to distinguish which of the instructions are to have their address parts modified and which are not. One reasonably simple means of identification which can be used in some calculators (including the example calculator) is to employ the sign of the number which represents an instruction. Previously, the signs of these numbers have been ignored because they were not used for anything. The convention might be used that all instructions represented by a positive number should not be altered, but all instructions represented by a negative number should have the appropriate constant added to their address parts. Of course, any constants in the sub-program must then be stored as positive numbers; otherwise, the positioning sub-program will incorrectly interpret them as instructions to be modified. Signs used in this manner are sometimes called "tags." Other means for making it possible for the positioning sub-program to identify the instructions to be modified would be to use extra storage locations to list the addresses of the instructions to be modified. After the sub-program to be positioned has been properly modified, the extra storage locations are no longer needed and may be employed for other purposes. Since the details of positioning sub-programs vary tremendously from calculator to calculator and since no new programming principles are introduced (once the possiblity of a positioning program is recognized), the subject will not be carried further here.

When assembling a library of sub-programs it is important to record not only the sub-programs themselves but also as much pertinent data as is available about each. In the example of the square root sub-program a graph of the required time as a function of the parameter, x, would be useful. In other cases where approximations or cumulative round-off errors are involved, the accuracy of the result as a function of the various parameters would be important. Also, sub-programs are likely to respond in peculiar and unexpected ways when certain combinations of parameters are used or when certain errors are made. The person preparing the sub-program might be well aware of its limitations, but unless these limitations are recorded, the full value of the sub-program as a library item will not be realized.

Interpretive Sub-programs, First Type. An important extension of the sub-program concept is the "interpretive" sub-program. When an interpretive sub-program is used, each step in the main program is "interpreted" and executed in a manner specified by one of a set of auxiliary sub-programs. Interpretive sub-programs of many different forms and variations have been developed; two have been selected for presentation here, and for lack of a better designation, they are referred to as first and second types, respectively.

Consider the situation where all items of data are represented by complex numbers of the form, x + jy, where j is the square root of minus one. With complex numbers, the arithmetic operations are not as simple as with real numbers. For examples, addition and multiplication in the complex number system are represented by the following equations.

$$(x_1 + jy_1) + (x_2 + jy_2) = (x_1 + x_2) + j(y_1 + y_2)$$

$$(x_1 + jy_1)(x_2 + jy_2) = (x_1 x_2 - y_1 y_2) + j(x_1 y_2 + x_2 y_1)$$

While it is possible to program the calculator so that it will perform all the necessary steps for each operation in the complex system, it would be desirable to relieve the programmer of this burden. The sub-programming procedure as described previously allows some reduction in programming effort, but it is possible to simplify program preparation much further. Through the use of the interpretive sub-program, each instruction such as ADD can be interpreted to mean the addition of two complex numbers, and the jump to the appropriate sub-program is made automatically and without any special link or other instructions being provided by the programmer (once the interpretive sub-program has been prepared).

For some purposes, particularly in forming loops and sub-programs which are not related to the fact that the data are in complex number form, it will be necessary for the ADD and other instructions to be interpreted in their normal way. To distinguish which instructions pertain to complex numbers, the signs of the numbers representing instructions may be used as tags. In the example to be presented, a minus sign will be used to signify that a complex-number operation is to be performed, and a plus sign will indicate that the instruction is to be interpreted in the usual way. Note that the programmer must be careful if the signs are used as tags for some other purpose such as in the positioning sub-program described in an earlier section.

The convention will be used that the two parts, x and y, of a complex number will be stored in two consecutive address positions. To specify the location of a number it is sufficient to give the address of the real part, and it is understood that the imaginary part is located at the next higher numbered address. Then, to illustrate by a simple example, all the programmer needs to prepare to add two complex numbers and print the sum are these instructions.

075	(-) RESET AND ADD	124
076	(-) ADD	151
077	(-) STORE	120
078	(-) PRINT	120
079	(+) STOP	---

The real parts of the two numbers are taken from addresses 124 and 151, and 120 is used for temporary storage of the real part of the sum. Addresses 125, 152, and 121 play analogous roles for the corresponding imaginary parts. The main program does not start at address 000; instead, in this example, the first instruction of the main program is found at address 075. The calculator follows the interpretive program and not the main program for its detailed instructions. Therefore, it is the interpretive program which should start at address 000.

Before describing the interpretive sub-program itself, one of the auxiliary sub-programs will be explained. Assume that the sub-program for adding two complex numbers is stored with its first instruction at address 500. In principle, this sub-program does the same thing as the ADD instruction for ordinary numbers; that is, the number at the specified storage location is added to the number already in the accumulator, and the sum is left in the accumulator. However, the accumulator built into the calculator is not capable of holding both the real and imaginary parts of a complex number at the same time. For this reason, two addresses, numbers 998 and 999, in the main storage are reserved for use as a sort of "acting" accumulator. The effect of the complex ADD sub-program, then, is to cause the addition of the complex number at the specified location to the complex number in 998 (and 999).

Address	Contents of Address		Remarks
500	RESET AND ADD	003	The address of complex number to be added is placed in 503. (The interpretive sub-program will have placed this address in 003.)
501	STORE ADDRESS	503	
502	RESET AND ADD	998	The real parts are added, and the sum is placed in 998.
503	ADD	[ ]	
504	STORE	998	
505	RESET AND ADD	503	The address of the imaginary part of the complex number to be added is formed by adding 1 to the address of the real part. The result is stored in 509.
506	ADD	L(+00001)	
507	STORE ADDRESS	509	

Address	Contents of Address		Remarks
508	RESET AND ADD	999	The sum of the imaginary parts is formed and placed in 999.
509	ADD	[ ]	
510	STORE	999	
511	JUMP	009	The calculator takes its next instruction from 009, which is in the interpretive sub-program.

An analogous sub-program will be needed for each type of instruction which can pertain to complex numbers. Besides the arithmetic operations, it is desirable to be able to interpret instructions such as STORE, PRINT, and SHIFT, as operations to be performed on complex numbers. The instruction, STOP, presumably has the same effect regardless of the type of number under consideration and may therefore be treated as an ordinary instruction always.

The interpretive sub-program serves the function of examining each instruction in the main program and causing a jump to the proper auxiliary sub-program whenever a complex number operation is involved. When the instruction is to be executed in the ordinary way, as indicated by a plus sign, the interpretive sub-program must provide for the appropriate calculator action. For interpretive sub-program to start properly, the address of the first instruction in the main program is placed in the address part of instruction at 000. The interpretive sub-program is as follows.

Address	Contents of Address		Remarks
000	RESET AND ADD	[075]	The number representing the instruction to be interpreted (the first one of which is at 075 in this example) is placed in 003.
001	STORE	003	
002	JUMP IF MINUS	005	If the instruction is accompanied by a plus sign, the calculator arrives at 003 and executes it in normal fashion and then jumps to 009. Otherwise, it jumps to 005.
003	[	]	
004	JUMP	009	
005	SHIFT RIGHT	003	The two digits representing the operation code are shifted to the right-hand end of the accumulator, and 12 is added to the result.
006	ADD	L(+00012)	
007	STORE ADDRESS	008	The number representing the operation code, increased by 12, is used as the address to which the calculator jumps from step 008.
008	JUMP	[ ]	

Address	Contents of Address		Remarks
009	RESET AND ADD	000	These four instructions have the effect of increasing the address part of the instruction at 000 by 1 and then causing a jump to 000 for the interpretation of the next instruction in the main program. Auxiliary sub-programs provide for a return to 009.
010	ADD	L(+00001)	
011	STORE	000	
012	JUMP	000	
013	JUMP	L (r and a)	The calculator arrives at one of these JUMP instructions from 008, and from here goes to the appropriate auxiliary sub-program. (Recall that the code for ADD is 02; the jump to 500 is made from 014 since 02 + 12 = 14.)
014	JUMP	500	
015	JUMP	L (sub.)	
016	JUMP	L (mult.)	
	.		
	.		
	.		

Observe that when the desired sequence of operations is accomplished through the medium of an interpretive sub-program, the calculator never does arrive at step 075 or any other step in the main program. Instead, the instructions of the main program are executed by transferring them to the accumulator and address 003 for interpretation. As a result of this situation, instructions of the JUMP type in the main program require further explanation. If a JUMP instruction in the main program is interpreted in the ordinary way (that is, if it is accompanied by a plus sign), the calculator will jump to the specified address, which could be at any point in storage. In particular, it could be at a location such that the calculator would leave the interpretive sub-program altogether. For some applications this effect would be highly useful. However, if it is desired that the JUMP instruction cause a jump to some other step in the main program while keeping the interpretive sub-program in effect, the JUMP must be accompanied by a minus sign. An auxiliary sub-program for JUMP is needed to make the required alterations in the interpretive sub-program. It is necessary that the address part of the instruction stored at 000 be changed to the address of the instruction which is to be interpreted after the jump; this alteration takes the place of increasing by 1 the address part of the instruction at 000. The following three instructions, arbitrarily stored at 580 to 582, may be used for auxiliary sub-program for JUMP.

Address	Contents of Address		Remarks
580	RESET AND ADD	003	The address part of the JUMP instruction from the main program (which was placed in 003 by the interpretive sub-program) is placed in 000. Return is made to 000 in this case, instead of 009, because it is not necessary to increase the address by 1.
581	STORE ADDRESS	000	
582	JUMP	000	

The auxiliary sub-program for JUMP IF MINUS is slightly more involved, but it follows the same principles. Here, the "IF MINUS" applies to the complex number stored in the "acting" accumulator. The "sign" of the complex number may be taken from either the real or the imaginary part by using the sign of the number stored in 998 or 999, respectively.

The same interpretive sub-program may be used for all problems where the data are represented by complex numbers. After it has been prepared for one problem, the detailed procedure by which it functions need no longer be of concern to the programmer. However, the programmer must keep in mind the addresses in storage which are consumed by the interpretive sub-program and all of its auxiliary sub-programs, because these addresses will not be available for use in the main program.

Other categories of problems to which interpretive sub-programs of this type can be applied are multiple-accuracy calculations (that is, where each number is so long that two or more locations are required for its storage), floating-point calculations, matrices, and multi-dimensional vectors.

Interpretive Sub-program, Second Type. The second type of interpretive sub-program which will be described is similar in basic principles to the first, but it is much more general in its applications. Instead of merely interpreting each existing kind of operation or instruction in some special way it becomes possible to create (in effect) any number of new operations which are not built into the machine.

To illustrate how new operations can be created by means of an interpretive sub-program, a simplified example has been worked out for the "specimen" machine. It should be understood that almost any practical interpretive sub-program for any real machine would probably differ vastly from the example, but the important concepts can be found in this simplified version. In the organization of the "specimen" machine, the first two digits of each word were used as a code to indicate the various instructions the machine is capable of executing. With two digits, one hundred different instructions can be represented, but only the codes from 00 to 13 were actually used. By using interpretive sub-programming techniques these same two code digits can be made to represent any one hundred instructions the programmer desires except that the required auxiliary sub-programs must not be so elaborate that the storage capacity of the machine is exceeded. Some of the selected instructions may be substantially the same as the "built-in" instructions, but in the general case there is no relationship between the list of instructions which are formed by programming and the list of "built-in" instructions which was given at the beginning of this chapter.

Suppose for example that the extraction of the square root is a frequently encountered operation and that it is desired to form a special instruction for it. The code, 30, might be selected for the square root instruction so that the number, 30721, will have the meaning: "extract the square root of the number found in address position 721 and leave the result in the accumulator."

As in the original examples, the sign of a number has no significance when the number is used to represent an instruction. Also, it will be assumed that ordinary numbers (not complex) are involved, although complex numbers or other special situations can be handled easily with appropriate auxilliary sub-programs and minor modifications in the interpretive sub-program. As in the interpretive sub-programs of the first type, the calculator never arrives at a program step which contains an instruction of the main program. Instead, each instruction in the main program is "interpreted" in accordance with the details of the interpretive sub-program together with all of its auxilliary sub-programs. Since the instruction counter which is built into the calculator is needed to control the progress of the machine through the interpretive and auxilliary sub-programs, an extra storage location (009 in the example below) is used as an "acting" instruction counter to cause the interpretive sub-program to interpret the main program instructions in the proper sequence. Further, the "accumulator" referred to in the square root instruction or in other programmed instructions is not the accumulator built into the calculator, but is a particular storage location (say 999) which is reserved for the purpose. In other words, an interpretive sub-program executes an instruction of the main program in a manner roughly analogous to the way the calculator itself executes an instruction of the interpretive sub-program. The procedure by which an interpretive sub-program functions is explained in the "remarks" column of the following example.

Address	Contents of address		Remarks
000	RESET AND ADD	009	The contents of the instruction counter (the address part of 009) is placed in 004 and then increased by 1.
001	STORE ADDRESS	004	
002	ADD	L(+00001)	
003	STORE	009	
004	RESET AND ADD	[ ]	The instruction to be interpreted is placed in the accumulator.
005	SHIFT RIGHT	003	The operation part of the instruction is caused to appear as the two lowest order digits.
006	ADD	L(+00009)	The code representing the operation or instruction to be performed is modified by the additon of 9, and the resulting number is used as an address to which a jump is made.
007	STORE ADDRESS	008	
008	JUMP	[ ]	
009	- -	[ ]	Instruction counter. Initially, the address of the first instruction to be interpreted is placed here.

Address	Contents of Address		Remarks
010	JUMP	xxx	Jump table. The jump to the appropriate auxiliary sub-program is made from here. The square root sub-program is assumed to start at 600.
011	JUMP	xxx	
	.		
	.		
039	JUMP	600	
	.		
.	.		
	.		

The auxiliary sub-programs can be substantially the same as described in earlier sections. However, certain alterations in the linkage are required. In the case of the square root sub-program for example, note that address 004 will contain the address of the instruction being interpreted, and the instruction being interpreted in turn contains the address of the number for which the square root is desired. Therefore, the following sequence of instructions can be placed at the beginning of the square root sub-program for the purpose of placing the number where the sub-program can find it.

Address	Contents of Address		Remarks
600	RESET AND ADD	004	Places the address of the instruction in 602.
601	STORE ADDRESS	602	
602	RESET AND ADD	[ ]	Places the address of the number for which the square root is desired in 604.
603	STORE ADDRESS	604	
604	RESET AND ADD	[ ]	Places the number where the sub-program can find it.
605	STORE	xxx	

With the system as described the square root sub-program must also contain an instruction which will cause the result to be placed in the "acting" accumulator. The return link is simply a JUMP 000 instruction in this case.

An outstanding extension of the interpretive sub-program principle arises from the fact that the instructions formed by the auxiliary sub-programs need not be limited to the single-address variety. A convention may be adopted whereby a group of two or more address positions may be used for each instruction, and an interpretive sub-program can be prepared which will interpret the instructions in any multi-address fashion. By this means a single-address calculator can be made to appear to a programmer as a multi-address calculator of any desired form (after the interpretive sub-program has been worked out). Conversely, a multi-address machine can be made to function in single address fashion from a programmer's viewpoint. The ability to make one machine appear like another one has proved to be of great value in calculator design. Before a new machine is actually built, programs for it can be tested and corrected on some existing machine. By this means, the need for certain improvements in the organization of the new machine can frequently be discovered at an early stage of design when it is not difficult to make changes.

Although it is probably obvious to most readers, it should be noted that the time required to execute a given number of steps in the main program is increased by a large factor in most cases where an interpretive sub-program is used.

Programming when Index Registers are Available. As was explained in the previous chapter, a calculator may be equipped with one or more index registers. When index registers are available, the calculator is usually so organized that certain digits in each instruction are reserved for specifying which index register is to be used and, in some machines, for specifying the amount (usually 1) by which the contents of the index register should be altered. In the examples which have been presented to illustrate various programming techniques, it may have been observed that the addition of 1 to the address part of an instruction was a frequently encountered requirement when a program loop of any sort was involved. In most cases the traversing of the loop was terminated when the address reached some predetermined number. The number of instructions involved was dependent upon the detailed requirements of the situation, but it was always at least three. With index registers, the number of instructions required for this function can be reduced to one because built-in circuits cause the number in the specified index register to be added to the address before execution of the instruction, and at the same time the number is increased by 1 (or some other specified amount) in an automatic fashion. Further, succeeding jumps in the program can be made automatically in accordance with whether or not the number in the index register has reached some preset value.

Besides vacilitating the preparation of program loops, index registers are found useful in numerous miscellaneous ways. No examples will be given because, although index registers have been incorporated in many calculator designs, the details and elaborations vary so greatly from machine to machine that the value of an example worked out for a "specimen" machine would be severely limited.

Assembly Programs. When a single program contains hundreds or thousands of instructions, its preparation contains problems which are not encountered to any extent in short programs where the programmer can remember the purpose and effect of each instruction. If, during the preparation or correction of a long program, it is necessary to alter the program by as little as inserting one instruction, it frequently happens that numerous changes throughout the program must be made. Not only are the storage locations of the instructions modified, but also certain constants and the address parts of many instructions are usually affected.. For this reason it is extremely difficult to prepare a long program without errors even after the logic of the program has been worked out perfectly. For the purposes of reducing the labor and chances for error involved in program preparation, techniques have been worked out whereby the calculator itself is given the job of assigning storage locations to the instructions and data and of determining the address parts of all instructions which refer to storage locations. With these techniques the programmer writes the program in sections with some sort of symbolic notation to describe the storage locations and addresses. An "assembly program" is then used to assemble the sections and compute the actual storage locations and addresses from the symbolic notation.

The basic idea behind at least one form of assembly program is to use a symbolism whereby one series of numbers designates the major sections of the program, a second series separated from the first by a dash or some other mark designates the instructions within the section, and third series designates instructions inserted after the initial writing of the program. A portion of the eighth section of a program might then appear as follows.

Address	Contents of Address	
8-12	JUMP	8-15
8-13	RESET AND ADD	10-2
8-14	SHIFT RIGHT	002
8-15	STORE ADDRESS	8-17
8-16	ADD	2-1
8-17	JUMP	15-3

The meaning of the instruction at 8-13, for example, is that the number representing the second instruction in the tenth section of the program is to be placed in the accumulator. The assembly program will assign an actual address in storage for the symbol 8-13; also, it will assign an address for 10-2 and place it in the address part of the instruction. The assembly program will not affect the address part of the shift instruction because no storage locations are involved and no alterations are to be made.

Suppose that through an oversight or for some other reason it is desired to insert a new instruction, say ADD 5-2, between instructions 8-13 and 8-14. A study of the example will reveal that it will be necessary to alter the storage assignments of all instructions after 8-13. Also, the address parts of some, but not all, of the instructions must be changed. However, if a symbol such as 8-13-1 is assigned as the address (location) of the inserted instruction, there will be no need for any reshuffling of symbols by the programmer. Since with this system of notation each address will be represented by different symbols and no two symbols will represent the same address, it is possible to make all changes in address assignments by means of the assembly program.

Numerous complexities in assembly programs are encountered in certain cases. One important case is in the assembling of a program with help of a library of sub-programs. Some means must be provided for distinguishing one sub-program from another; otherwise address 1-1 would mean one thing in one sub-program and something else in another (it is assumed that the same symbolic system is used for the sub-programs). Also, situations arise where it is desirable to cause particular instructions in two different sub-programs to refer to the same address where the symbolic addresses as stored in the library are not the same.

Another complexity arises when the assembly program and the program being assembled are too extensive to fit into the storage unit of the calculator. The procedure in this case is to divide the program into parts and assemble one part at a time. Since the instructions in one part may refer to the addresses in another part, it is not always possible to convert the address part of each instruction from symbolic form to an actual address on the first pass through the machine. By using a part of the storage unit for maintaining a file of unassigned addresses the assembly of the program can be completed the second time it is entered into the calculator. A further requirement of the assembly program is that it record the assembled program in forms suitable for subsequent use by the calculator and suitable for visual reference by the programmer.

Most practical assembly programs for real machines are themselves long and complicated and may require literally months to prepare. The usefulness of an assembly program comes, of course, from the fact that, once prepared, it can be used without alteration for the assembly of innumerable other programs.

The "Speed-coding" System. A different approach to the problem of simplifying the preparation of long programs is to provide by interpretive sub-program techniques a set of instructions which are much more comprehensive than those built into the machine. By this means it is possible in many cases to eliminate all sub-programs and certain other complicating factors in the program as prepared by the programmer; in fact, it is not even necessary that the programmer understand sub-programming techniques at all. Another advantage of the system is that the number of instructions which must be written to solve a given problem can be reduced by a large factor. The name, "speed-coding", was first applied to a system of this type which was prepared at IBM for the 701 calculator. Other groups have worked out analogous systems for different calculators, and other names have been chosen to describe them.

As in the case of an assembly program, the speed-coding interpretive program is long and involved and requires a matter of thousands of instructions with its usefulness being derived from the fact that, once prepared, it may be used by programmers who have no knowledge of how it works. The programmer need be familiar only with the list of "programmed instructions" which the speed-coding system provides. Included in the list are instructions for square root, sine, arc tangent, exponentials, and logarithms, all of which would normally require sub-programs when using the "built-in" instructions of the calculator. Also, several input and output instructions are included which allow the programmer to move an entire block of information from one place to another with only one instruction. To handle situations where it is desirable to modify an instruction, a set of index registers (referred to as R-quantities in this case) and an address counter are provided by the speed-coding system. The index registers function in principle by the procedure described previously, and by means of the address counter it is possible to replace selected addresses with new ones or to add address increments in a variety of manners.

A further convenience included in the speed-coding system are special instructions for the automatic checking of calculations. By placing a START CHECK instruction at the beginning and an END CHECK instruction at the end of the series of instructions to be checked, the calculator will be caused to proceed through the sequence twice and to generate a check sum each time. The calculator will then compare the two check sums and, if they are equal, it will skip the instruction immediately following the END CHECK, but if there is a discrepancy, that instruction will be executed. In the event an error is detected, the programmer may control the course of action the calculator is to follow by having placed an appropriate instruction in the position immediately following the END CHECK instruction.

The basic ideas employed in writing a program in the speed-coding system are substantially the same as when writing a program using "built-in" instructions. There are a great many differences in detail, however, because of the considerable differences in the nature and quantity of the individual instructions. An instruction in the speed-coding system consists of two operations, four addresses, and a digit relating to the index registers. One operation and three of the addresses are used for arithmetic and input-output functions in a manner similar to that of a three-address calculator. The other operation together with the fourth address are used for jump, address modification, and error checking functions, although there are many interrelationships between the two operations. All items of data are handled through the speed-coding system as floating point numbers even though the calculator is a fixed-point machine.

In spite of the relative complexity of the individual instructions in the speed-coding system, the over-all task of preparing a program for a given mathematical problem is made easier because many of the confusing factors arising from sub-programming are avoided through the interpretive process and because many less instructions in the "main" program are required. An outstanding disadvantage of the speed-coding system is that the great number of instructions ~~are~~ consumed in the interpretive process cause more time to be required for calculations in the speed-coding system than when the program is prepared entirely in terms of "built-in" instructions.

An interesting combination of assembly program and speed-coding concepts is the use of an assembly program to prepare address assignments for a program written in terms of speed-coding instructions.

<u>Tracing Programs</u>. It seems to be almost axiomatic that a program when first prepared will not work. Occasionally programmers are able to write short programs that are totally free from errors, but for a long program the places where typographical errors, unforeseen circumstances, and mistakes in logic can arise are so numerous that the chances of getting it right the first time are extremely small. In spite of the difficulty a human being has in discovering errors by a mental checking process, the calculator usually will show up errors in programming very quickly by a failure to arrive at the right answer. The calculator, then, is the best tool the programmer has for perfecting his programs. However, the calculator will show

only the existence of an error; it will not indicate precisely what or where the error is. The programmer still has the job of deducing the source of the error from the calculator's behavior, which may be peculiar to say the least.

To assist the programmer in locating errors in a program, "tracing programs" may be used. A tracing program is basically an interpretive program which functions in the same general manner as other interpretive programs which were described by means of examples in earlier sections. When a tracing program is used, the calculator does not execute the instructions of the main program directly; instead, each instruction is "interpreted". The interpreting in this case accomplishes the same end result as direct execution except that certain additional features are added. For example, each instruction may be caused to be printed as it is executed so that the programmer can have a record of what the calculator actually does. By studying this record the path of the calculator through the main program can be determined to see whether or not it was following the various jump instructions as expected. The address parts of the instructions can be observed to indicate whether or not the desired items of data entered into the calculations as each point. Tracing programs can also be used to obtain a printed record of the contents of the accumulator or of other registers at the end of each program step. With this information the progress of the calculations for a sample set of parameters can be compared with the results obtained from a desk machine in an effort to locate the mistake.

Innumerable variations can be worked out for special cases. For example, if the programmer is interested only in knowing the sequence of program steps that the calculator followed, the tracing program can be altered so that it will be limited to printing information relative to jump instructions. In other applications, calculated values will be needed only at certain intermediate points in the program instead of at each program step. Since the progress of the calculator through the main program is much slower when a tracing program is used, it may be important from a purely time consideration to limit the scope of the tracing functions.

Diagnostic Programs. If errors are being encountered which place the calculator and not the program under suspicion, special programs known as "diagnostic" programs can frequently be used to locate the source of the errors. A diagnostic program is to be distinguished from a test program although the dividing line between the two is not at all well defined. A test program is generally employed to determine whether or not the calculator, or some particular portion of it, is working properly. This purpose is accomplished by exercising each element in the calculator as thoroughly as practical by means of appropriate instructions in the program and then observing to see that the proper response is obtained. Since the proper response may consist of a check sum or some other number obtained after a long sequence of operations, there may be no indication of the source of an error in the event that the existence of an error is sensed. On the other hand, diagnostic programs are generally employed after a defect is known to exist, and the purpose is to find the defect.

The details of any specific diagnostic program would be dependent upon the engineering details of the calculator under consideration. However, there are a few considerations which appear to apply to nearly all calculators. For one thing, for a diagnostic program to work, the calculator must be functioning well enough "to get a program off the ground"; that is, the instructions of the program must be executed properly, and therefore certain portions of the calculator must be in good condition. For this reason it is difficult to prepare a program which will properly diagnose an error in some of the important control circuits of the machine. The power supply is another source of errors which are difficult to locate by means of diagnostic programs. It is usually necessary to check these parts of the calculator through the use of oscilloscopes, voltmeters, and other test instruments.

While it would be desirable to have one all-inclusive diagnostic program which would indicate the cause of an error regardless of its source, it is generally more feasible to employ a set of specialized programs each of which pertains to a relatively small portion of the machine. Frequently when errors are encountered, the approximate location of the source is known, and in this case the appropriate diagnostic programs may be selected to aid in finding the trouble quickly. In instances where incorrect results are being obtained with no indication whatsoever of the cause, the specialized diagnostic programs are still of great value although it is then necessary to examine the entire calculator in some systematic manner such as would be used when testing a new machine.

For IBM's 701, for example, a library of at least seventy different diagnostic programs has been assembled. One of the simplest (and surprisingly useful) programs is merely a blank card to be fed through the card reader; many clues to error causes can be obtained if the card fails to feed properly or if extraneous digits are entered into the machine. Most of the diagnostic programs are highly specialized and pertain to such things as controls for a particular instruction, the instruction counter, a specific property of the electrostatic storage unit, drum addressing circuits, or the printer circuits. The medium through which the programs present the results of their diagnosis to the operator is usually a stopping of the calculator at a point where the pertinent data is readily observable in the various registers. Instructions for interpreting the data are recorded in the library along with the programs. Some of the diagnostic programs are capable of generating relatively elaborate reports on the output printer.