Memorandum M-2539-1

December 1, 1953

Revised April, 1954

COMPREHENSIVE SYSTEM MANUAL

I. Introduction to Programming

Digital Computer Laboratory
Massachusetts Institute of Technology
Cambridge 39, Massachusetts

by

H. H. Denman

E. S. Kopley

J. D. Porter

I. INTRODUCTION

In barter, man first learned the need for assessing the relative sizes of different commodities. He found it possible to do this in two quite different ways - by measuring to determine how much, or by counting to determine how many. Just as he learned to measure length by comparing with the length of a hand or foot, he learned to count by placing the apples or skins or stones to be counted in one-to-one correspondence with his fingers - the digits on the end of his arms. Little wonder that he learned to count by fives and tens, or that the symbol V was used to represent one hand full while X represented two hands full.

Computation developed out of measuring and counting, the two sciences being called geometry and arithmetic, respectively. The introduction of a very important concept - the digit zero - and thence the development of the Arabic or positional system of numbers, permitted arithmetical computation to be performed with much greater facility than before. As always, greater speed led not to less time spent, but to more computation being undertaken.

As the complexity and frequency of each problem grew, the need to mechanize the arithmetical processes became more urgent. A thousand years ago, the development of the abacus overcame the limitation set by the inadequacy of the number of fingers and toes a man could produce. Finally, hundreds of years after the development of the abacus, and thousands of years after the beginnings of counting and arithmetic, great minds produced the little cogs which grew into the modern adding machines, desk calculators and accounting machines.

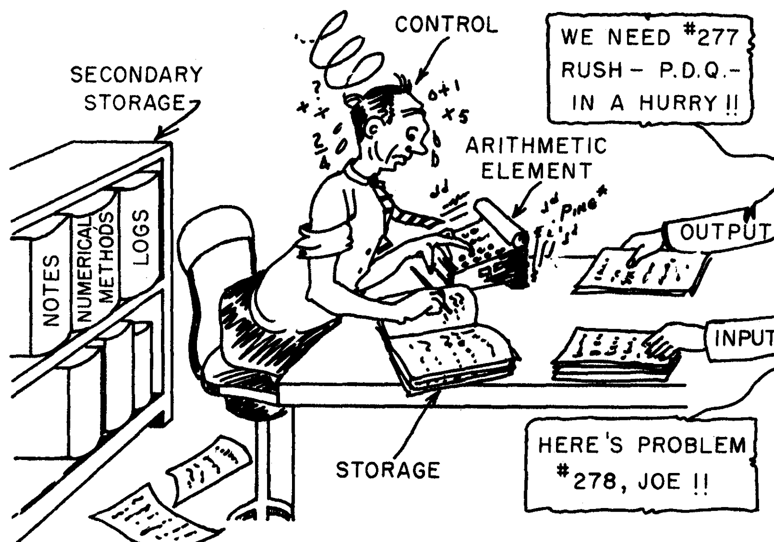
As time went on, the cogs grew better and went around faster. Special sets of cogs made the machines perform the sequences of addition needed to form a product, and later the sequences of addition and subtraction needed to find a quotient. Motors replaced hand cranks. Today a good mechanical calculator multiplies two 10 digit numbers in ten seconds or less.

Modern electronics could speed this up almost a million fold. But to what avail? Practical experience indicates that a competent person operating a modern calculator performs about 500 operations a day - and many of these operations require but a fraction of the 10-second maximum in the calculator. Speed up the calculator a million fold and you speed

up the overall computation by at most 10%, assuming that the operator can stand the increased strain. What is needed is to replace the human being in the system, not merely to speed up the arithmetical processes themselves.

PORTRAIT OF CHARLES BABBAGE.

More than 140 years ago, one man, Charles Babbage, dreamed of machines which would far surpass the wondrous contrivances of Pascal and the others. The more advanced of his two machines, for he dreamed of two different types, would perform the basic operations of addition, subtraction, multiplication and division. And it would do much more: It would perform, automatically and without human intervention, a predescribed sequence of arithmetic operations and make predescribed logical decisions based on the results as it went along. In outlining his Analytical Engine in 1834, Babbage described all the important principles of today's ultramodern automatic digital computers. It took over a century, however, for mechanical and electrical sciences to reach a state at which his dream could be fulfilled.

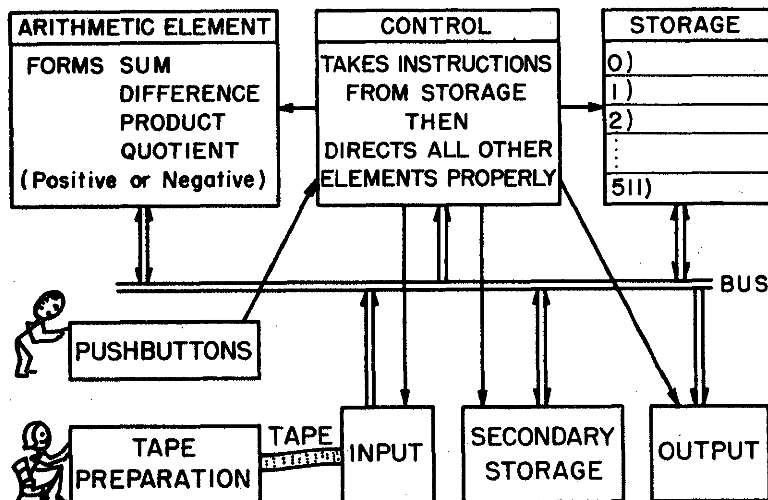


SEMI-AUTOMATIC DIGITAL COMPUTATION

The automatic digital computer is simply a mechanization not only of the arithmetical operations, but of the operator which determines the sequence in which the operations are performed. The arithmetic element of the digital computer, corresponding to the desk calculator, can advantageously be made to work very fast, performing arithmetical operations in a few millionths of a second, for the rest of the system can now keep up with it. The control element, the counterpart of the human operator, can readily be made far

faster, more reliable, and somewhat less demanding of wages and fringe benefits than the man.

Unfortunately, however, there is need for automatic memory or storage of various degrees of accessibility, corresponding to the memory of the operator, the notebook, and the reference library. There is also need for input and output - the means of communication with the outside world. Primarily, it is memory or input-output, depending upon the particular problem, that causes the greatest difficulty in the physical realization, and places the greatest limitations on the speed and reliability of existing computers.



AUTOMATIC DIGITAL COMPUTATION

When a human operator is to solve a problem using a calculator or to process a payroll on an accounting machine, he must be supplied with instructions which specify just how the solution is to be obtained. In like manner, the digital computer must be provided with a list of instructions, or program, in properly coded form, to describe how the solution is to be obtained. The process of preparing such a coded program is called programming. Programming really consists of two parts:

1) planning the program, or sequence of elementary steps, by which the problem may be solved
2) coding the sequence of steps into a coded program - a sequence of computer instructions

The coding of a problem requires detailed knowledge of the specific computer on which the problem is to be solved. A coded program has meaning only to the computer for which it was written. The planning of a solution, on the other hand, does not necessarily involve the details of any given computer, although a given problem may frequently be solved most efficiently if formulated one way for one computer and another way for another.

The particular computer for which this manual is prepared does not in fact exist, but its characteristics have been simulated on the Whirlwind I computer by means of a system of programs. This system has been termed the Comprehensive System of Service Routines and abbreviated as CS. The simulated computer will be called the "CS computer".

The CS computer has, of course, the basic computer elements: arithmetic element, control, primary ("high-speed") storage, secondary storage, input and output. Naturally, a number of important concepts and innumerable details

make up a complete description of the computer. Rather than attempt to describe the computer completely at the outset, we will first describe a simplified form of the computer, embellishing it with more details and more new concepts as we progress. In simplifying anything, one must sometimes tell half-truths, and this we will do; but we shall not tell any forthright lies.

The CS Computer - Simplified Version

The primary storage element of the computer consists of 1387 registers. Storage registers are locations, pigeon holes into which computer words may be stored by the computer control and recovered by it when needed. A word is a sequence of digits representing a number or an instruction. The location of each register is identified by an address, just as the houses on a street are identified by addresses. The addresses of the 1387 different registers are 32, 33, 34, 35, ...,1417 and 1418.

In the CS computer a word that represents a number occupies two successive registers of storage. The location of the number is always specified by the address of the first of the two registers. The maximum magnitude of a number that can be stored in the memory of the CS computer is about $9. \times 10^{18}$, and the smallest non-zero magnitude is about 5.5×10^{-20}. A programmer will write his numbers in the usual decimal form:

e.g., ± 129.7863

where he may indicate as many as 8 significant digits provided the magnitude of the number satisfies the range requirements indicated above.

A more detailed discussion of the representation of numbers will be given below.

An instruction is the second kind of word and occupies a single register of storage. It specifies both an operation such as add or subtract, and an address. This address designates where the **word** to be operated upon is to be found. For example, the word iad 237 is an instruction which specifies that the number contained in registers 237 and 238 is to be added to the number already in the multiple register accumulator (MRA).

The multiple register accumulator (MRA) forms the heart of the arithmetic element. In it, the sum or difference, product or quotient of two numbers is formed. The maximum magnitude of a number that can be handled in the MRA is about 7.0×10^{9863} whereas the minimum magnitude for a non-zero number in the MRA is about 7.1×10^{-9864}.

The first few basic instructions to be considered are described below by their abbreviations, names, and effects. They can be described more concisely and compactly if a few standard symbols and terms are first defined.

symbol	meaning
MRA	multiple register accumulator
al	address of any chosen storage register
N(MRA)	the number in the MRA before the instruction is obeyed
N(al)	the number stored in registers al and al+1 before the instruction is obeyed
$\longrightarrow$	replaces
clear ...	set the contents of ... to zero

Thus "N(MRA)+N(al) $\longrightarrow$ MRA" is to be read as "the initial number in the MRA plus the number stored in registers al and al+1 replaces the initial number in the MRA"; i.e., the sum of the numbers contained in MRA and al appears in MRA.

Abbreviations for the instructions of the CS computer have been selected for mnemonic reasons. An initial letter i is used to denote that the instructions are interpreted by special programs stored in the Whirlwind I computer (see Chapter)

ica al	clear MRA and add to it N(al)	N(al) $\longrightarrow$ N(MRA)
ics al	clear MRA and subtract from it N(al)	-N(al) $\longrightarrow$ N(MRA)
its al	transfer N(MRA) into (al,al+1)	N(MRA) $\longrightarrow$ N(al)
iad al	add	N(MRA)+N(al) $\longrightarrow$ N(MRA)
isu al	subtract	N(MRA)-N(al) $\longrightarrow$ N(MRA)
imr al	multiply and roundoff	N(MRA)·N(al) $\longrightarrow$ N(MRA)
idv al	divide	N(MRA)÷N(al) $\longrightarrow$ N(MRA)
iex al	exchange N(MRA) with N(al)	N(MRA) $\longrightarrow$ N(al);N(al) $\longrightarrow$ N(MRA)
iTOA+nl.2345c	type out N(MRA) in normalized*form followed by a carriage return	
sp 0	STOP	

IF IT ISN'T MENTIONED, IT DOESN'T HAPPEN!

The definition of ica al does not specify that anything new goes into register al. This means that N(al) does not change. Likewise, N(al) is unchanged by ics, iad, isu, imr, and idv. Similarly, N(MRA) is unchanged by its, iTOA and sp 0.

---

* The normalized form referred to is one that represents a number in the form

$$\pm\, d_1 . d_2 d_3 d_4 d_5 \times 10^{\alpha}$$

where α is adjusted so that $d_1 \neq 0$.

WHEN A NEW QUANTITY REPLACES AN OLD ONE, THE OLD ONE DISAPPEARS!

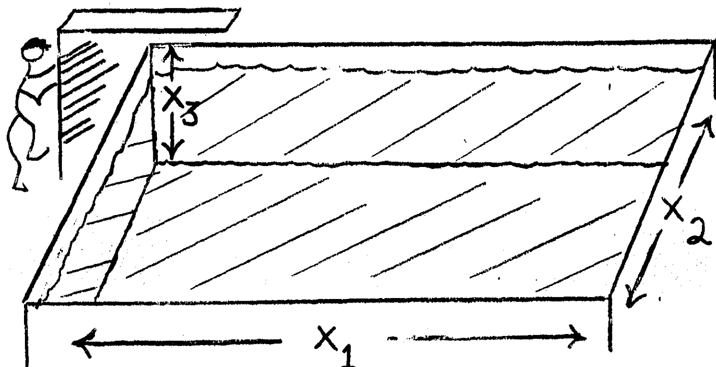
The definition of ica says that N(al) becomes the new contents of MRA. The MRA is first cleared in this process so that its former content is lost.

TOO LARGE A RESULT LEADS TO TROUBLE.

Obviously, if the result of an arithmetic operation lies outside the range that can be stored, the result cannot be copied into storage. It may, however, be further operated on in the MRA. If a programmer, through oversight, instructs the computer to copy into storage a result which will not fit, the computer will stop and indicate an alarm. Obviously, also, it is possible for the result of an arithmetic operation to become too large to fit even in the extended capacity of the MRA. This, too, is an overflow and produces an alarm.

Straightforward Computation - Example 1.

Suppose a rectangular swimming pool of any given dimensions is to be filled with water to a level 1 foot below the top of the pool. Before filling, the pool is to be painted green on the bottom and on all four sides up to the water level. One gallon of paint covers 500 square feet, and one cubic foot of pool water weighs about 63 pounds. We wish the computer, given the length, width and height in feet, to print out:



a) The weight of the water the pool will hold

b) The number of gallons of green paint needed

Representing feet of length by X_1, width by X_2, and height by X_3, we readily find that the pool surface area to be painted equals

bottom + 2 sides + 2 ends

$$= X_1 \cdot X_2 + 2 \cdot X_1(X_3-1) + 2 \cdot X_2(X_3-1)$$

and the amount of paint in gallons is obtained by dividing the above number of square feet by 500, the number of square feet per gallon.

The volume is $X_1 \cdot X_2(X_3-1)$, and the weight of water is obtained by multiplying this number of cubic feet by 63 pounds per cubic foot.

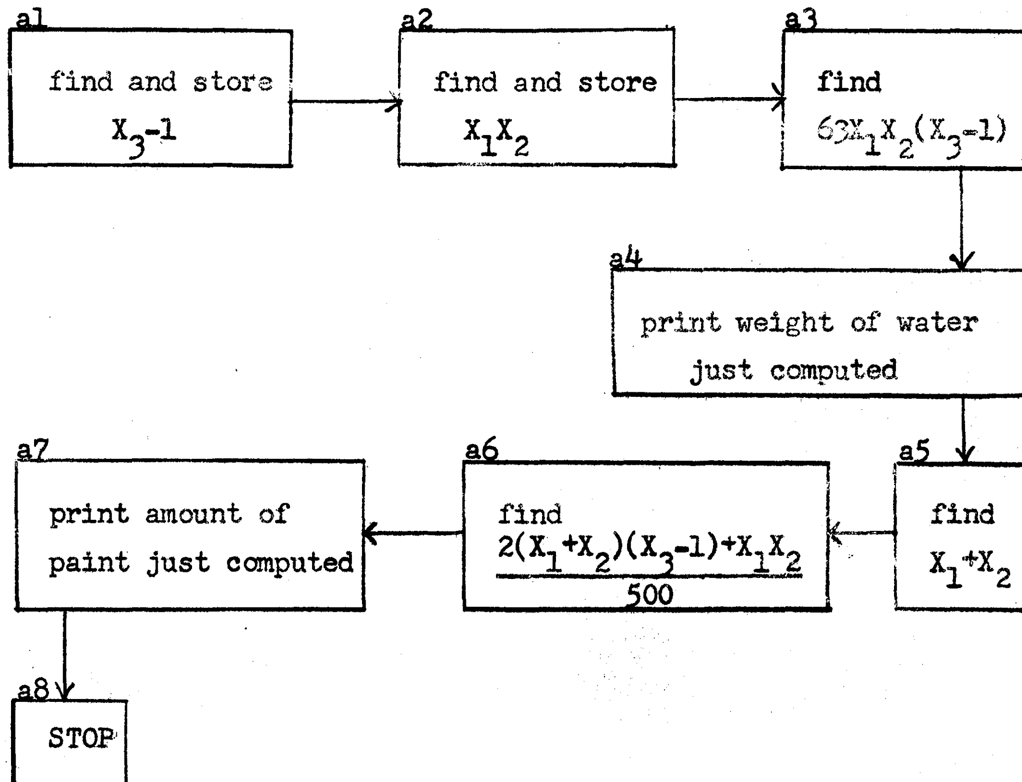
By collecting terms with an eye to reducing the number of multiplications which will be required in the computation, the two quantities desired can be

rewritten as follows

$$\text{Paint} = \frac{2(X_1+X_2)(X_3-1) + X_1 X_2}{500}$$

$$\text{Water} = 63 X_1 X_2(X_3-1)$$

A procedure for finding these quantities might be



Certain numerical constants are obviously needed. These must be stored in storage registers. The registers may at this stage be chosen anywhere in storage that the programmer desires. Suppose we place them in the first available registers (viz., 32, 33, etc.) - although other registers might have been used. The only rule that needs to be noted at this time is that instructions must be stored in a sequence of registers that corresponds directly with the sequence in which the instructions are to be executed. Thus numbers may be stored anywhere provided they do not interrupt the sequence of instructions.

32	+1.
34	+2.
36	+63.
38	+500.

Recall that +1. occupies registers 32 and 33, hence +2. occupies registers 34 and 35, etc. The dimensions of the pool must also be provided initially.

(It is worthwhile for the student to note that by beginning the problem with a different set of dimensions this same program can be used unchanged.) Suppose we store this data as follows:

40| (length) = X_1
42| (width) = X_2
44| (height) = X_3

During the calculation various intermediate quantities (viz., X_3-1 and X_1X_2) will be calculated and need to be stored. We can provide for them by initially storing +0. in two pairs of registers which will later contain the desired quantities; thus

46| +0. } Initially, later { X_3-1
48| +0. } { X_1X_2

We are now ready to write down the required instructions. We can begin storing them in any register we desire but once we have chosen the first register, succeeding instructions must occupy successive registers. Exceptions to this rule will be the subject of Chapter II of these notes. Since the next available register in storage is 50 (not 49, since +0. occupies both 48 and 49) we could begin with that register. Note that if we were to need some more registers for constants or intermediate quantities we could wait until after we had written down all the necessary instructions and then choose these additional registers so as not to interrupt the sequence of instructions. In the present case, our preliminary analysis has made this unnecessary. We can, then, write the instructions as:

a1	50|	ica 44	place N(44) = height in MRA
	51|	isu 32	subtract N(32) = +1.; leaving X_3-1 in MRA
	52|	its 46	store the result in register 46(for later use); the result also remains in MRA
a2	53|	ica 40	clear the MRA and add to it (i.e., place in it) the N(40)=X_1
	54|	imr 42	multiply by N(42) = X_2, forming X_1X_2 in MRA
	55|	its 48	store the result in register 48(for later use); the result also remains in MRA
a3	56|	imr 46	multiply by N(46) = X_3-1, forming $X_1X_2(X_3-1)$ in MRA
	57|	imr 36	multiply by N(36) = +63., forming $63X_1X_2(X_3-1)$ in MRA, equal to the weight of water.
a4	58|	iTOA+nl.2345c	type out the numerical value of N(MRA) which is the desired result
a5	59|	ica 40	place X_1 in MRA
	60|	iad 42	form X_1+X_2 in MRA

(continued on next page)

```
a6 { 61  imr 34   form 2(X1+X2) in MRA
     62  imr 46   form 2(X1+X2)(X3-1) in MRA
     63  iad 48   form    "    +X1X2 in MRA
     64  idv 38   form 2(X1+X2)(X3-1) +X1X2  in MRA equal to the amount of paint
                       ------------------
                              500

a7   65  iTOA+nl.2345c    type out the amount of paint

a8   66  sp 0     stop
```

The brackets indicated to the left of the above program show how the sequence of instructions effects the operations included in the boxes of the diagram on page I-7. Such a diagram is often called a "flow diagram". This is an example of how more complicated programs can be decomposed into the programming of simpler logical blocks. Of course the present program is rather trivial but the usefulness of this procedure will become apparent later.

II Transfer of Control - Counting

Thus far, the computer has been instructed to solve simple arithmetical problems, but it can only be made to solve the same sort of problem more than once by starting it over again manually with new data. Since whole programs such as the swimming pool paint and water calculation just given can be performed (exclusive of output) by the computer in less than 1/40 of a second (faster than the eye can see), there would be a tremendous proportion of time spent by the computer in waiting to be told what to do next. The key to the situation is to give the computer ability: (1) to repeat calculations with new data which it has itself generated, (2) to make predescribed logical decisions based on results it has obtained, and (3) to modify not only its own data but its own instructions. These abilities will be discussed and illustrated in this and the following sections.

Special instructions are needed to make the computer repeat, or make logical decisions. These are called "jumps" or "transfers of control", and they tell the computer, under certain conditions, to take the next instruction not from the next consecutive register, but from the register specified by the address section of the jump instruction. One of the jump instructions is unconditional; the other is conditional:and makes the computer transfer control if and only if a given condition is fulfilled; otherwise the next instruction is taken in sequence.

The following are the available transfer of control instructions:

isp al	transfer control	take the next instruction from register al and continue from there
icp al	conditionally transfer control (conditional program)	ditto, if N(MRA)<0*; if N(MRA)>0, take the next instruction in sequence

* We define N(MRA)>0 if the sign of the number stored in the MRA is +;and N(MRA)<0 if this sign is -. Arithmetic rules apply in the normal way except for the difference of two equal numbers. In this case the MRA will contain zero but the sign is -; hence the icp would assume N(MRA)<0.

Unconditional Transfer of Control (Example)

Suppose a swimming pool contains 40,000 gallons of fresh water. Once each minute a bucket containing 20 gallons of salt water, containing .1 pound of salt per gallon, is lowered gently into the pool. A corresponding amount of pool water, unmixed with the newly-added salt water, but thoroughly mixed otherwise, escapes through an overflow pipe at the other end. We wish the computer to print out the amount of salt which will be in the pool after 1 minute, 2 minutes, 3 minutes, etc.

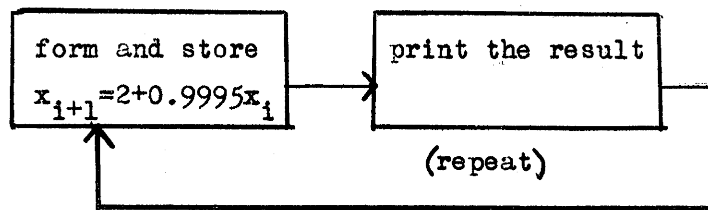
During each minute, 2 pounds of salt come in with the salt water. After 1 minute then, there will be 2 pounds of salt in the water. But during the second minute, not only do 2 more pounds come in, but a small quantity goes down the drain. The amount down the drain is 1/2000 times the amount present, since 20 gallons out of 40000 contains one two-thousandth of the amount of salt present. Hence, at the end of the second minute, the total salt equals the 2 pounds from the first minute plus the influx of 2 pounds minus the spillover of $2 \cdot 1/2000 = 0.001$ pounds, for a total of 3.999. During the third minute, 2 more pounds come in, and 1/2000 of the 3.999 already there escapes, leaving 5.9970005 pounds. To formulate this more generally, let x_i = pounds of salt at the end of the ith minute. Then,

$x_0 = 0$	at start
$x_1 = 2$	during first minute
$x_2 = 2 + 2 - 2/2000 = 3.999$	during second minute
$x_3 = 2 + 3.999 - 3.999/2000 = 5.9970005$	during third minute

and in general

$$x_{i+1} = 2 + x_i - x_i/2000 = 2 + 0.9995x_i \qquad \text{during the } i+1^{th} \text{ min.}$$

A possible procedure for programming the problem would be:



The program can be written:*

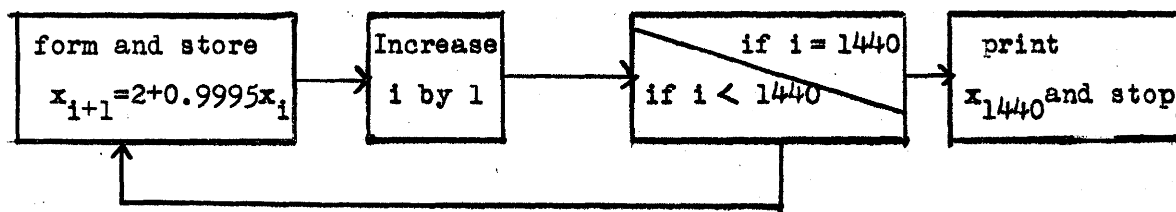
100	+2.	pounds of salt added
102	+0.9995	
104	+0.	amount of salt in pool (initially 0)
106	ica 104	place $+x_i$ in MRA
107	imr 102	form $0.9995x_i$
108	iad 100	form x_{i+1} in MRA
109	its 104	replace x_i by x_{i+1} in register 104
110	iTOA+nl.2345c	print x_{i+1}
111	isp 106	repeat; i.e., take next instruction from 106

Counting Using Conditional Transfers of Control (example)

The program just written will be performed over and over again until stopped by human intervention or machine breakdown.

Suppose what is really desired is knowledge as to how much salt will have accumulated in one day = 1440 minutes. One could, of course, simply wait until 1440 lines of results had been printed, then stop the computer and copy the result. Much more efficient, however, would be a revised program which would compute 1440 steps without printing, print the result, and stop. For this, we have the computer decide, by means of a conditional transfer of control just when 1440 steps have been completed. The importance of such an ability can hardly be overemphasized.

The necessary program might be:



---

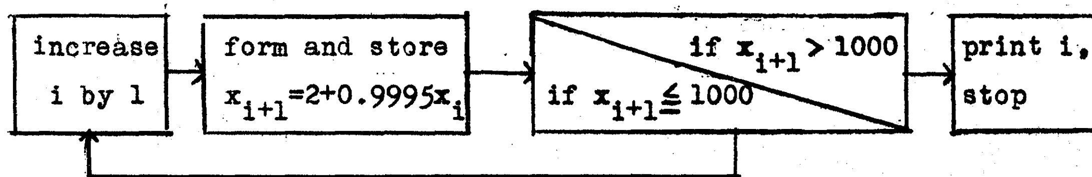
* As an exercise the student may attempt to shorten this program.

```
100|    +1.          }
102|    +2.          }
104|    +0.9995      }  constants
106|    +1439.       }
108|    +0.             pounds of salt
110|    +0.             minutes
112|    ica 108
113|    imr 104
114|    iad 102
115|    its 108
116|    ica 110
117|    iad 100
118|    its 110
119|    isu 106         form i - 1439. in MRA
120|    icp 112         if N(MRA) < 0, which occurs (since i changes in unit steps)
                            if i < 1440, take the next instruction from 112
                        if N(MRA) > 0, which occurs when i = 1440, ignore this
                            instruction
121|    ica 108          }
122|    iTOA+nl.2345c    }.  print x_1440
123|    sp0
```

Note that if +1440. had been stored in 106, then when i became = + 1440, the N(MRA) would be < 0 (cf. page II-1). Hence control would be transferred back to 112.

Calculating Until Desired Value is Reached (example)

As a third possibility, suppose what is really wanted is the time at which the amount of salt in the pool exceeds 1000 pounds. Again the computer must be programmed to make a simple decision. A possible program would be:



```
501|   +1.        }
503|   +2.        }  constants
505|   +0.9995    }
507|   +1000.     }

509|   +0.           pounds of salt
511|   +0.           minutes

513|   ica 511    }
514|   iad 501    }  increase i by 1
515|   its 511    }

516|   ica 509    }
517|   imr 505    }  calculate new x_{i+1}
518|   iad 503    }
519|   its 509    }

520|   isu 507    }  If x_{i+1} ≤ 1000, take next instruction from 513
521|   icp 513    }  If x_{i+1} > 1000, ignore this instruction and go on to 522

522|   ica 511
523|   iTOA+n1.2345c
524|   sp 0
```

Programming Exercises

Construct sequences of instructions to carry out the following processes on the CS computer.

It will be assumed that x and y are contained in registers 32 and 34 respectively. All results will be assumed to have values that will not exceed the capacity of any register. Stop the computer after each problem.

1. Place $x+y$ in 41.
2. Place x^3 in 53.
3. a, b, c, d are contained in 100, 102, 104, and 106, respectively. Place ax^3+bx^2+cx+d in register 500.
4. Place the larger of the positive numbers x and y in register 75.
5. Place x^4 in 115.
6. Place x^9 in 115 by a program of no more than 8 instructions.
7. Place x^n in 115; where $x \neq 0$ and n is an integer $\geq +0$, unknown to you, which has been placed in 72 by a preceding program.
8. Place x^{41} in 77.
   (a) using the fewest possible number of registers, storagewise, in your program.
   (b) using the fewest possible number of instructions, timewise, in your program. (Assume that the CS computer consumes about the same amount of time for each instruction.)

<u>Solutions to examples on p.II-6</u>:

```
1)      ica 32
        iad 34
        its 41
        sp 0
2)      ica 32
        imr 32
        imr 32
        its 53
        sp 0
3)      ica 100
        imr 32
        iad 102
        imr 32
        iad 104
        imr 32
        iad 106
        its 500
        sp 0
4) 100| ica 32
        isu 34
        icp 106
        ica 32
        its 75
        sp 0
        ica 34
        isp 104
5)      ica 32       ica 32
        imr 32       imr 32
        imr 32  OR   its 115
        imr 32       imr 115
        its 115      its 115
        sp 0         sp 0
6)      ica 32
        imr 32
        imr 32
        its 115
        imr 115
        imr 115
        its 115
        sp 0
7) 102| ics 72
        its 72
        ica 72
        iad 113
        its 72
        icp 109
        sp 0
        ica 115
        imr 32
        its 115
        isp 104
        +1.
        +1.
8(a) 65| ica 77               66| ica 73
         imr 32                   iex 77
         its 77                   imr 32
         ica 73        OR         iex 77
         iad 75                   iad 75
         its 73                   icp 67
         icp 65                   sp 0
         sp 0                     -40.
         -40.                     +1.
         +1.                      +1.
         +1.
```

```
8(b) 100|ica 32           100|ica 32
         imr 32     OR        imr 32
         its 77               imr 32
         imr 77               imr 32
         its 77               imr 32
         imr 77               its 77
         its 77               imr 77
         imr 77               its 77
         imr 77               imr 77
         imr 77               imr 77
         imr 77               imr 77
         imr 32               imr 32
         its 77               its 77
         sp 0                 sp 0
```

III Cycle Counter - Modification of Addresses

In the cyclic examples described in chapter II the addresses of instructions within each cycle did not change. However, one of the more important jobs that computers are called upon to do involves dealing with data which are stored consecutively and are to be operated upon cyclically in a set fashion. Cycle counters have been devised to facilitate the counting of the repetitions of a cycle of instructions and the modifying of instructions therein. Cycle counters are often called B-boxes, a term that has received wide-spread adoption since it was first used with the Ferranti computer at Manchester University in 1948.

A. Cycle Counting

As was indicated in chapter II, the counting of cycles of operations can be carried out by programming, utilizing the MRA. However, if an intermediate numerical result happens to be in the MRA it must be copied into storage while the counting is done. For example, suppose we wish to calculate x^{10} by forming in succession x^2, x^3, x^4,... (involving 9 multiplications) using a cycle of instructions. The program might be as follows:

```
43| +1.
45| +9.      number of times cycle }  criterion
            is to be carried out  }
47| x
49| +0.      receives result
51| +0.      used for counting } index
53| ica 47
54| its 49
55| ica 43
56| its 51
57| ica 49 }                    }
58| imr 47 } multiply by x      }
59| its 49 }                    }
60| ica 51 } increase           }
61| iad 43 } counter            }  cycle
62| its 51 } by 1               }
63| isu 45                      }
64| icp 57                      }
65| sp 0
```

Note that the power of x obtained in the MRA after the instruction in register 58 has been executed must be stored preparatory to the counting effected in registers 60-64.

The number of registers used in this program can easily be reduced. However, the form above was chosen to illustrate that cycle counting consists basically of two elements: an index element that actually counts the number of times the cycle has been carried out; secondly, a criterion element for determining when the cycle has been carried out the desired number of times.

In the CS computer special facilities have been included for counting cyclic operations independently of the MRA. The heart of this cycle counter is the cycle control register pair. This is actually two storage registers, one of which is called the index register and the other, the criterion register. Provision is made for clearing the index register, setting the criterion register to any desired integral value (up to 2047), increasing the index register by any desired integral amount (up to 2047), and testing when the magnitude of the integer in the index register becomes equal to or greater than the magnitude of the integer in the criterion register.

Care should be taken not to confuse the integers stored in the single index register and single criterion register with the ordinary numbers that are stored in two consecutive registers. The arithmetic instructions described in Chapter I deal automatically with two-register numbers. However, the following instructions affect the cycle counter and hence, as indicated, deal with the integers stored in the single special registers. (See Chapter).

We define C(...) = contents of ...

i = C(index register)

n = C(criterion register)

icr m	cycle reset	Set i = +0, n = m ($+0 \leq m$ is an integer < 2048)
ict al	cycle count	Increase i by 1 and if this new value of $\lvert i \rvert \geq \lvert n \rvert$, then reset i = +0 and take the next instruction in sequence; if the new $\lvert i \rvert < \lvert n \rvert$, take the next instruction from register al.

The calculation of x^{10} may now be done by the following program.

```
32| x
34| +0.
36| icr 9     set up for 9 cycles
37| ica 32
38| imr 32 \
           } cycle
39| ict 38 /
40| its 34
41| sp 0
```

The following table presents a history of the contents of the index and criterion registers and the MRA after the execution of the ict instruction in register 39 of the program above.

		i	n	MRA
End of cycle	1	1	9	x^2
	2	2	9	x^3
	3	3	9	x^4
	4	4	9	x^5
	5	5	9	x^6
	6	6	9	x^7
	7	7	9	x^8
	8	8	9	x^9
	9	9 reset to 0	9	x^{10}

B. Modification of Addresses

The machinery for adjusting an address by means of the cycle counter is quite simple. The programmer simply appends "+c" to an address. When this instruction* is to be executed the address is first modified. If we let "i" denote the integer that is contained in the index register, then the address is increased by 2i before it is executed (except in the case of the instruction whose operation is isp where the address is increased merely by i).

* The +c cannot be used in the icp and ict instructions.

For example if the programmer writes

ica 100+c

then when this instruction is to be executed, the following instruction is actually formed

ica (100+2i)

and then executed. The increment 2i was selected since we are usually working with arithmetic operations on numbers and these numbers occupy two registers of storage. In the case of isp 100+c we get isp (100+i) since this is used with instructions (recall that instructions occupy one register of storage). It should be noted that if at any time one were to examine the contents of the register containing the instruction ica 100+c the address part would be 100 (not 100+2i). The increment 2i (or i in the case of isp) is added on only during the execution of the instruction.

A simple example illustrating the use of the cycle counter for address modifications as well as for counting is the following. Suppose we wish to transfer the numbers in registers 100, 102, 104 and 106 to registers 200, 202, 204, 206. We would then write:

32	icr 4	set up for four cycles
33	ica 100+c	clear MRA and add to it N(100+2i); i=0,1,2,3
34	its 200+c	store N(MRA) in 200+2i; i=0,1,2,3
35	ict 33	add one to index register; if the new $|i| \geq 4$ then reset i=+0 and take the next instruction from register 36; if $|i| < 4$, then take the next instruction from register 33. Note that n does not change (=+4).
36	sp 0	

Since there are many cases when we desire to operate on numbers that are not stored in consecutive locations, but are spaced a constant number of registers apart, we have the following instructions:

ici m	cycle increase	Increase the contents of the index register by m.
icd m	cycle decrease	Decrease the contents of the index register by m.

Here $+0 \leq m$ is an integer < 2048.

As an example of the use of the ici instruction, let us write a

program which transfers numbers in register 100, 104, 108 and 112 into registers 200, 204, 208 and 212.

We have:

301	icr 8	set up for 4 cycles	{ i=+0, n=+8
302	ica 100+c	pick up N(100+2i)	i=0,2,4,6
303	its 200+c	store in 200+2i	i=0,2,4,6
304	ici 1	increase i by 1	
305	ict 302	go through 4 cycles	

The following table presents a history of the contents of the index and criterion registers after the execution of the ict instruction in register 305 of the program above:

		i	n
End of cycle	1	2	8
	2	4	8
	3	6	8
	4	{ 8, reset to 0	8

C. Multiple Counters

Since programs usually contain cycles within cycles, provisions have been made for selecting any number of counters the programmer requires (the upper limit on the number of counters available to each program is a function of the amount of storage the programmer is willing to spare for counting). Multiple counters are often referred to as counter lines. The following instruction permits the use of more than one cycle control register pair so that more complicated programs may be treated effectively:

isc j select counter (+0 $\leq$ j is an integer $<$ 2048)	selects counter j as reference for all subsequent interpreted instructions using a counter until execution of the next ics instruction; each counter has its own index and criterion registers

If a programmer wishes to use only one cycle counter, there is no need for him to select this counter with the isc j instruction. He will

automatically get one cycle counter if there appears in his program any cycle counter instruction (other than ici, icd, or icx*). In any program, counter zero is initially considered selected until an isc j, with j > 0 is executed. Except for this property, counter zero is no different from any other counter such as 1, 2,....

Two separate cycle counter registers (index, criterion) are set aside automatically for counter n (where n is the maximum value for j in any program) and for counters n-1, n-2,..., 2, 1, 0. Consequently, it is advisable to select an uninterrupted sequence of counters so as not to waste storage for counters that are never used. Thus if the only isc j instructions in a particular program were isc 0, isc 3, isc 6, the programmer would be wasting 8 registers of storage.

To illustrate the use of this instruction, suppose we have a program that calculates the values of two quantities, F and G, as functions of the time t = jΔt for j = 1,2,3,...,1000 (Δt is a prescribed increment of time, say .01 seconds). Suppose further that it is desired to print out the value of F at the end of each 5 time steps (i.e., for i = 5, 10, 15,...), the value of G at the end of each 20 steps, and to stop the program at the end of 1000 time steps. If we store the value of F when calculated in 200 and of G, in 202, the following program would suffice:

```
32|   isc 0
33|   icr 50
34|   isc 1
35|   icr 4
36|   isc 2
37|   icr 5
38|   isc 2
39|   ... ..      calculate
      ... ..      and store
      ... ..      F, G
      ... ..
103|  ict 39
104|  ica 200
105|  iTOA+nl.2345c   Note that the letter c used here does not
                      refer to the cycle counter but, as will be
                      discussed in chapter 5, gives a carriage
                      return.
```

(continued on page 7)

*icx defined on page 7

```
106| isc 1
107| ict 38
108| ica 202
109| iTOA+nl.2345c
110| isc 0
111| ict 38
112| sp 0
200| +0.   will contain F
202| +0.   will contain G
```

(Would the program be as efficient without register 38? Explain.)

This problem could have been done differently by using an instruction we are about to define. <u>However, the first method is the preferred one since it is logically simpler</u>.

icx al	<u>c</u>ycle e<u>x</u>change	Exchange C (index register) with C(al) and exchange C (criterion register) with C(al+1).

<u>Second Method</u>

```
32| icr 50
33| icx 204
34| icr 4
35| icx 206
36| icr 5
37| ... ..  calculate
    ... ..  and store
    ... ..  F, G
    ... ..
103| ict 37
104| ica 200
105| iTOA+nl.2345c


106| icx 206
107| ict 35
108| ica 202
```

(continued on page 8)

```
109| iTOA+nl.2345c
110| icx 204
111| ict 33
112| sp 0
200| +0.  will contain F
202| +0.  will contain G
204| +0.  count for 1000Δt
206| +0.  count for G
```

<u>Advanced Section</u>

D. The following two cycle counter instructions appear in this chapter merely for completeness. They are considered to be part of the more advanced section of this manual for two reasons:

(1) There is an easier way to accomplish the same effect.

(2) They are used more rarely than other cycle counter instructions.

iat al	<u>a</u>dd and <u>t</u>ransfer	add C(index register) to the C(al) and store the result in the index register and in register al
iti al	<u>t</u>ransfer <u>i</u>ndex digits	transfer the right 11 digits of the index register into the right 11 digits of register al

These two instructions are principally used for altering the address section of an instruction.

The examples presented in the preceding sections have been simple ones chosen to illustrate specific points. It should be clear to the student that, in practice, programs are far more involved than the ones displayed. Nevertheless, even though these examples are simple, certain inconveniences may be observed in the writing of the programs. First of all, in writing a sequence of instructions it is rather tedious to have to write down all of the addresses especially since only a few of them are referred to in other instructions. But even worse, note that if by error we had left out an instruction in our sequence (e.g., if we had forgotten to multiply by (x_3-1) in the program on page I-8) then to insert this instruction would require our renumbering all the subsequent instructions and then searching all the address parts of instructions to correct those affected. This is not only annoying but very often leads to needless errors.

It should be pointed out that there is a remedy that can be used to avoid this inconvenience. To be specific, if we had:

```
...
...
55| its 48
56| iad 36
57| iTOA+n1.2345c
```

where we have omitted the instruction imr 46, between registers 56 and 57 we could replace the instruction in register 56 by an isp to some block of unused registers (e.g., 70, 71, 72); that is:

```
...
...
55| its 48
56| isp 70
57| iTOA+n1.2345c
```

and then we add to the program:

```
70| iad 36    carrying out the instruction
              that previously had been in 56
71| imr 46    carrying out the omitted
              instruction
72| isp 57
```

Such a procedure for correcting a program is frequently called a "patch".

Note that "patching" is not only unaesthetic, but it is wasteful of space and makes a program more difficult to follow (and therefore to correct) since it interrupts the basic logic of the program.

Finally we might note that before we write down each of the sample programs above we had to set aside certain registers for input data, intermediate results, and final results. Now it was emphasized that it mattered little where in storage we put these registers provided they did not interrupt a sequence of instructions. We now see that by using our jump instruction (isp) we have a good deal of flexibility in interrupting such sequences. However, it should be clear that it would be bad practice to make use of such jump instructions (since it is wasteful of computer storage*) simply to jump over a misplaced constant. On the other hand when one first begins to write down a program it may be very difficult to determine just how many registers will be occupied by data needed in the program and how many are needed for holding intermediate results. If one leaves too many registers for them then he may find he doesn't have enough registers left for his program. On the other hand if he doesn't leave enough - or if he actually starts out by writing instructions first (beginning in register 32) - then, since he has no way of knowing a priori precisely how many registers will be occupied by instructions, he is faced with the problem of what addresses to assign to the registers needed for the as yet unspecified data or results.

The obvious solution to this dilemma is to assign some sort of tentative addresses to these unspecified registers. Since we are already using numbers to specify addresses it is only natural to distinguish these unspecified registers by a literal nomenclature. Since there are only 26 letters, a simple system is to use letters followed by integers (e.g., a1, b12, g3, etc.). Such addresses will be called <u>floating addresses</u> (abbreviated as <u>flads</u>) since the actual value of the address (called the <u>absolute address</u> to distinguish it) can not be determined until the program is complete.

---

* Also in many cases when a set of instructions is repeated a great many times such extraneous instructions can represent a needless expenditure of computer time.

Thus for the example on page II-3 we could have written the program initially as:

```
100| ica i1
101| imr d3
102| iad b1
103| its i1
104| iTOA+n1.2345c
105| isp 100
{i1, +0.
{b1, +2.
{d3, +0.9995
```

It should be emphasized that at this point it does not matter what we label the bracketed quantities so long as each label is unique. We might even use names such as Joe, Tom, etc. However, the combination of a lower case letter followed by an integer is a neat and convenient one and has been adopted in the CS computer. (<u>The letters o and l are excluded.</u>)

Once we appreciate that these floating addresses (flads) can be chosen at the programmer's will, we recognize the possibility of mnemonic labelling. This makes it easier for others to follow the program - and also easier for the programmer himself to check his program. For example, we could use the letter c for registers assigned to contain constants, the letter x for a variable, etc. Thus the program above might have been written as:

```
100| ica x1
101| imr c2
102| iad c1
103| its x1
104| iTOA +n1.2345c
105| isp 100
c1, +2.
c2, +0.9995
x1, +0.
```

Having introduced the idea of a floating address we might examine the possibility of writing our sequence of <u>instructions</u> with such a procedure. Let us consider the example above. Note that the sequence of

instructions written there begins in register 100 and occupies each successive register through 105. However, the only instruction whose address needs to be identified is register 100 since that address is referred to by the instruction (isp 100) in register 105. Consequently, we could have written this same sequence of instructions as follows:

```
al,ica xl
   imr c2
   iad cl
   its xl
   iTOA +nl.2345c
   isp al
```

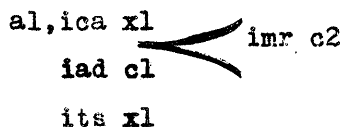
In this form the address al is floating - that is, the actual register in storage to be occupied by the instruction ica xl is unspecified. Once we specify that al should be equal to 100, these instructions take the form they had on page IV-3 since successive instructions will occupy successive registers.

However, note the tremendous flexibility we have gained by using the floating address form. First of all we do not need to write down a whole lot of addresses. We need only identify or tag those registers to which we wish to refer; e.g., we tag the register containing ica xl so that we can instruct the computer to transfer control to that register at a suitable point in our program (isp al). For this reason we shall refer to "al," as a floating address tag.

Secondly, if we discover that we have omitted an instruction we need only insert it at the proper place. For example, if we had erroneously written:

```
al,ica xl
   iad cl
   its xl
```

we need only indicate the correction by writing:




```
   its xl
```

The rewriting of the program into absolute address form is a simple clerical procedure. Each word is assigned an absolute address in a consecutive sequence (remembering that numbers occupy two successive registers), and then the address section of each instruction is replaced by

the corresponding one of the newly assigned absolute addresses. Since the procedure is straightforward, it is perfectly possible to make the computer perform the task automatically during input. The CS computer is so arranged that this substitution of absolute addresses for floating addresses is performed automatically when the tape containing the program is read into the computer. Consequently, although the programmer may do the job himself if he wishes, there is no need to rewrite a program in absolute address form. The tape is simply prepared using floating addresses as indicated; the rest of the job is performed by the computer. The actual procedure followed by the CS computer in transforming floating addresses to absolute addresses will be discussed in Chapter V.

The letter and number(s) forming a floating address may be chosen at will (except that the letters l and o should not be used because of ambiguity with the numbers 1 and 0). One other restriction is imposed by the procedure used by the CS computer for keeping track of the flads as the program is being read in. The sum over all letters of the maximum numbers used for each letter should not exceed 255 - e.g., if a program used only the floating addresses a1, a2, a3, a17, d9, x31, x100, and z5, this condition would be satisfied since 17+9+100+5=131 which is less than 256.

It is possible to refer to a register that has not been tagged by a floating address. This is done by referring its address to a floating address that has been used, e.g.,

```
       ...
       ...
    b1,ica c1
       its i1
       ica i1
    a1,imr c3
       ...
       isp b1+2
```

The instruction isp b1+2 will transfer control to that register whose address is <u>two</u> more than b1. Note we can obtain the same result by writing isp a1-1. This instruction would transfer control to that register whose address is one less than a1. Care should be taken in applying this procedure to numbers since they occupy two successive registers.

Thus in the example:

```
al,+17.6
   +3.984
   -0.78
bl,ica al+2
   its il
   ...
   ...
```

the instruction ica al+2 will place +3.984 in the MRA. The tendency, of course, would have been to use ica al+1 which would have been in error. This is one reason for avoiding the use of these address references. An even more significant reason for limiting the use of this procedure is the fact that it makes it as difficult to insert corrections as in the case of absolute addresses. For example, if we wanted to insert +7. in our program between +17.6 and +3.984, we would have to be careful to correct the address of the instruction in bl, etc. Consequently, rather than referring to the address of +3.984 by al+2, a different floating address is advisable.

It should be pointed out that it is permissible to use both floating addresses and absolute addresses within the same program. All of the sample problems given above can be used with the CS computer.

PROGRAMMING EXERCISES

Construct sequences of instructions to carry out the following processes on the CS computer.

It will be assumed that x and y are numbers contained in registers 32 and 34 respectively at the beginning of each problem. All results will be assumed to have values that will not exceed the capacity of any register. Stop the computer after each problem.

(1) Do the following examples of the first set of programming exercises (on page II-6) using the cycle counter instructions if this will shorten the program:

(a) ex. 6 (No more than 7 instructions).

(b) ex. 8 (a)

(2) Initially N(c1) = z and N(c2) = w. Make N(c2) = z and N(c1) = w.

(3) Find the sum of the 200 numbers in the consecutive registers d1, d1+2, d1+4,.....,d1+398. Place the sum in c4.

(4) Calculate x^n where $x \neq 0$ and where the cycle count pair (index = +0, criterion = $n \geq 0$) has been stored in registers 71 and 72 by a preceding program; the value of n is unknown to you. Place the answer in register 115.

Solutions to programming exercises on p.IV-7:

```
1. (a)     icr 8
           ica 32
        al,imr 32
           ict al
           its 115
           sp 0

   (b)     icr 40
           ica 32
        al,imr 32
           ict al
           its 77
           sp 0

2.     ica c1
       iex c2
       its c1
       sp 0
3.     icr 200
       ica c4
   a1, iad d1+c
       ict a1
       its c4
       sp 0
   c4, + .0
4.     ica 115
       icx 71
       icd 1
   a1, ict a2
       its 115
       sp 0
   a2, imr 32
       isp a1

 115| +1.
```

I. Input

Thus far it has been assumed that programs and data can somehow be gotten into the computer without worrying in detail how one goes about actually doing so. The process is simple and straightforward, but certain conventions must be observed. The conventions are described below.

Instructions, numbers, and certain control information must all be typed on a Flexowriter tape-perforating machine. This machine can simultaneously type a printed copy and punch a paper tape. In response to each key that is depressed, a unique combination of holes is punched in each of six of the seven positions across a 7/8 inch tape, the combination indicating which of the 50 different keys on the typewriter has been depressed. The code values corresponding to each of the keys are tabulated on a list called the Flexowriter code (see Table 1). The seventh hole is used for control purposes and must always be punched (which is accomplished automatically by leaving the button labeled "7th HOLE" depressed).

In point of fact, the programs are typed almost exactly in the form in which they have been written in these notes so far. The basic rules are:

1. Instructions: typed as 3 lower case letters followed either by a floating address made up of one letter (not o or l) and 1, 2, or 3 digits (any integer from 1 thru 255), or by an absolute address made up of 2, 3, or 4 digits (any integer from 32 thru 1418) followed by a carriage return or a tab shift. The only exceptions to this rule are the "sp 0" instruction for stopping the computer and the output instruction iTOA+nl.2345c. Other output instructions are available and will be discussed in detail later in this chapter.
2. Numbers: typed as a plus or minus <u>SIGN</u> followed by as many as 8 significant digits if desired with a <u>DECIMAL POINT</u>, <u>NO COMMAS</u>, followed by a carriage return or a tab shift. Exponentials with base 2 or 10 may be appended as factors each preceded by an x (e.g., $+1234.56x2^{-3}x10^{5}$). Numbers may be zero or have any magnitude between about $5.5x10^{-20}$ and $9x10^{18}$.
3. Absolute address assignments: typed as a 2, 3, or 4 digit integer (any integer from 32 thru 1418) followed by a VERTICAL BAR.

This causes the word that follows the vertical bar to be stored in the register identified by the absolute address that precedes the vertical bar(that is, the word is "assigned" to this register). It should be remembered that if the word is a number it will occupy two successive registers, the first of which is specified by the absolute address that precedes the vertical bar.

If the first word of a program is not preceded by an absolute address assignment it will automatically be stored in register 32. All succeeding words will be stored sequentially (with numbers occupying pairs of registers) until an absolute address assignment is encountered. The word that follows directly after the vertical bar will be assigned by the general rule. All successive words that are not given absolute address assignments will be stored sequentially following the last absolute address assignment. For example, if a programmer began his program with:

```
    ica 500
    imr 502
    its 36
    isp 70
    +.0
70| ica 712
    imr 36
    its 602
    .
    .
    .
```

he would find in storage in the CS computer:

Address of Register	Contents of Register
32	ica 500
33	imr 502
34	its 36
35	isp 70
36	+.0
37	

(continued on next page)

Address of Register	Contents of Register
⋮	⋮
70	ica 712
71	imr 36
72	its 602
⋮	⋮

Of course if the first word of a program is preceded by an absolute address assignment, the word will be assigned to the corresponding register (or register pair). Succeeding words will be stored sequentially in those registers following the given absolute address until another absolute address assignment is encountered, etc.

4. Floating address tags: typed as one lower case letter (not o or l) and 1, 2, or 3 digits (any number from 1 thru 255) followed by a COMMA. This is called a tag since it is used by the programmer to identify the word that follows it. For example

```
    c1,+.5000
```

tags the constant +.5000 so that it can be referred to elsewhere in the program (e.g., imr c1).

The general rules given above in section 3 for assigning words to storage registers are unaffected by the presence of floating address tags. The floating address itself is set equal to the absolute address of the register that contains the word tagged by the floating address. Thus if a program began with

```
a1, ica 500
    imr 502
    its i1
    isp 70
i1, +.0
70| ica 712
     imr i1
a2, its 602
    ⋮
```

the programmer would find in storage:

Address	Contents
32	ica 500
33	imr 502
34	its 36
35	isp 70
36	+.0
37	
⋮	⋮
70	ica 712
71	imr 36
72	its 602
⋮	⋮

The floating address al would be replaced by 32 wherever it appears in the program, il by 36, a2 by 72, etc. It should be pointed out that the only way a floating address gets set equal to an absolute address is when that floating address is used as a tag. Consequently, if a floating address is used in an instruction but is never used as a tag, the program will be in error and not perform properly (see VI-6).

5. Beginning of tape: in preparing the perforated paper tape to be read into the CS computer, the following two lines should be typed before typing the program itself:

a) The first line should contain suitable identifying information. The first word of this line should be "fc" followed by "TAPE" (with each letter capitalized, i.e., upper case). This word should be followed by the number that identifies the tape and at least one space. The programmer may then write his name followed by a space and then the date. Commas may be used where desired. However, the total number of characters (including spaces and commas) should not exceed 60.

Example of a typical title:

fc ↑ TAPE ↓ 123-45-6789 John Doe ↲

b) The second line should contain the special expression (24,6).

6. End of tape: each tape should conclude with a line containing i START AT xyz where xyz denotes the address (e.g., a1 or b12 or 719 or a1+5) at which the program starts. The words START AT are capitalized.

7. Typographical errors: if the typist makes a mistake while punching a tape on the Flexowriter and detects the error immediately (before any more characters are punched on the tape), then the tape can be corrected by backing the tape one line in the punch. This places the incorrect character under the punching heads. If the typist then presses the "Code Delete" button, all seven holes will be punched across that line of tape (this is called a "nullify" character). This character will be ignored when the tape is read into the computer. Similarly, if several characters have been punched after an erroneous one, all of these characters could be punched over with the "nullify" character, starting with the first incorrect one. The typing and punching can then be resumed with the correct characters. If an error is undetected for a large number of lines, it is usually necessary to duplicate the tape up to the error, then punch the correct character, skip the error on the original tape, continue to duplicate it, etc.

   Splices suitable for the available tape reader are difficult to produce. Occasionally, ingenious ways of correcting small mistakes can be found, but there are no standard and recommended ways available.

8. Corrections in the program after tape is typed: a tape may be remade by duplication and correction, and if floating addresses are used throughout, insertions and deletions may be made at will in the program. Simple changes can sometimes be made by adding words at the end of the tape, preceded by absolute address assignments. This causes the new words to be read in over the incorrect words, replacing them. To enable the programmer to make corrections in registers whose absolute addresses are not easily determined, he may make use of:

9. Floating address assignments: typed as a floating address that had been previously used as a tag, plus a small integer if desired, followed by a VERTICAL BAR. This causes the next word to go into a register already used, that is, the next word is "assigned" to this register.

Floating address tags should not be used in a program following the use of a floating address assignment unless an absolute address assignment intervenes. For example:

```
Correct              Incorrect
   .                    .
   .                    .
   .                    .
a3,iad b4            a3,iad b4
   its c6               its c6
   .                    .
   .                    .
   .                    .
a3|isu b4            a3|isu b4
550|ica a5              ica a5
   its i7               its i7
   isp d3               isp d3
c1,+.5               c1,+.5
```

Both programs will replace the contents of register a3 by isu b4. However, the program on the right will then store ica a5 in a3+1, etc. but it will NOT associate the correct absolute address with the floating address c1.

10. Ignored and synonymous characters: the space, back space, and nullify are completely ignored by the computer. Thus spaces may be used for typographical reasons wherever desired. They are recommended between operation and address sections of instructions. Carriage returns are interpreted in the same way as tabs; both have the logical function of terminating a word. Extra carriage returns or tabs may be used at will except within words or addresses. Commas, periods, signs, vertical bars, letters, and numbers all have certain meanings and must not be used indiscriminately. The digit zero and the letter o are interchangeable as are the digit one and the letter l. Shifts to upper and lower case have meaning and should not be used indiscriminately, but it is

actually only when punctuation, letters, or digits are typed in upper case that special things happen. If the shift key is accidentally pushed and no character typed before shifting down again, no harm is done. An important rule to which the computer adheres is: IF, WITHOUT MANUAL MOVING OF THE CARRIAGE, THE TAPE PRINTS AN ACCEPTABLE COPY, THE TAPE IS VALID; i.e., there are no mistakes possible on tape that do not show on the typewritten copy when printed from the tape.

11. Layout: ordinarily, several words are typed to a line, separated from one another by a single tab, the last word on a line being followed by a carriage return in place of a tab. The tab stops are set permanently and should not be changed.

A series of 10 consecutive vertical bars (called a FENCE) may be inserted where desired to subdivide the Flexowriter tape into convenient visible blocks.

It is good practice to tab twice before an address tag or assignment and once after it, making it easy to spot the address on the printed page. However, a tag or assignment <u>need</u> only be preceded by <u>one</u> tab and followed by none (i.e., followed immediately by an instruction or a number).

12. Sources of error: there are numerous errors that can appear on a tape. They manifest themselves in various ways. Some of these are looked for by the computer.

The computer detects the following mistakes: numbers that are too large, an excessive number of flads (more than 255, as explained on page IV-5) or of output requests (more than about 50), illegal Flexowriter characters (characters that do not appear in Table 1a on page V-13), referring to floating addresses that are not used as tags, illegal duplicate flad tags, starting addresses that are too large, programs that accidentally exceed the available storage, and the use of a flad tag after a flad assignment without an intervening absolute address assignment (see section 9 on page V-6).

Other errors, such as addresses that are too large and ambiguous words are not specifically detected by the

computer but usually cause improper operation of the program. Careful proofreading of the program typed during the preparation of the tape is suggested.

II. Output

The basic idea behind the procedure that has been set up for an output request is that the programmer should write a sample number in his output request. A program will then be automatically set up in the storage of the CS computer to present the output in the form desired.

The output **media** that are currently available for these automatic routines are: (1) a "direct" typewriter on which numbers may be recorded, and (2) a "delayed" typewriter, where the numbers are first recorded in Flexowriter-coded form at high speed on magnetic tape and later typed out while the computer is doing something else. The maximum speed of these media and the maximum number of characters obtainable on one line are as follows:

1) Typewriter: 8 characters/sec. 154 characters/line
2) Magnetic Tape(to be used later with Typewriter) 133 characters/sec. 154 characters/line

A programmer indicates his output request by writing the letter i followed by three upper case letters followed by a sample number. The first of the upper case letters will be either an M for magnetic tape or a T for direct typewriter. The second is O to indicate output. The third is A to indicate that he desires alphabetic or numerical (alphanumerical) output. For example, the request

iTOA+123.1234 (1)

will automatically set up in the storage of the CS computer a program that will print out the contents of the MRA as a decimal number, with proper sign, having three digits to the left of the decimal point and four digits to the right.

1. Initial Zeros

If the number actually contains more than three digits to the left, the routine automatically adjusts itself to print them all. On the other hand, any non-significant digits (i.e., initial zeros) will be printed as zeros. In many operations it is desired to skip initial zeros (except for the one just to the left of the decimal point) and print the first significant digit of the number at the extreme left of the column. This feature can be obtained by inserting the letter "i" in the request just before the sample number,

e.g.,

iTOA+i123.1234

On the other hand, it is often desired to line up the numbers so that the decimal points fall in a line. Yet it may be desirable to omit printing any initial zeros. By inserting the letter "p" instead of "i", e.g.,

iTOA+p123.1234

initial zeros will be printed as spaces.

2. Normalized Form

Finally, it may be desired to print all of the numbers in a normalized form, i.e., all numbers are multiplied by a power of 10 such that the first non-zero significant digit always falls in the same relative position with respect to the decimal point. In this case, the number printed is followed by a vertical bar followed by the signed power of 10 that the number is to be multiplied by. This kind of output is obtained by inserting an "n" instead of "p", e.g.,

iTOA+n123.1234

As an example, consider the number -7.953261. The above request will give the following printed numbers:

```
using form (1)..........-007.9532
           (2)..........-7.9532
           (3)..........-  7.9532
           (4)..........-795.3261/-02
```

3. Signs

If the programmer wishes to have the sign of all numbers printed, then he writes + after the iTOA as in the examples already considered.

In some applications the programmer may know that all his numbers are of one sign(e.g., positive) and therefore may not want to take the time or space to print the sign. In this case he simply omits the sign from his request;e.g.,

iTOA n123.1234

and the printed number will be unsigned.

On the other hand, he may want only the negative numbers to appear signed. For this he writes:

iTOA-n123.1234

Note that he cannot get both positive numbers with signs and negative numbers without signs from a single output instruction.

4. Terminal Characters

In any of the cases above, the carriage of the typewriter will remain exactly where it was after the last number was typed. It is possible for the programmer to terminate his number with one, two, three, or four spaces, or with a carriage

return, or with a tab. To get the spaces he simply writes the proper number of s's after his number - e.g., to get two spaces:

iTOA+i123.1234ss

To get a carriage return, use a "c" instead of the s's:

iTOA+i123.1234c

and for a tab:

iTOA+i123.1234t

5. Decimal Point

If the programmer wants only the digits to the left of the decimal point printed and does not want the decimal point itself printed (e.g., for integers) he need only omit the point in his request, thus:

iTOA+i123ss

On the other hand, if his numbers are all less than one and he desires to omit the decimal point in his print-out, he simply replaces the decimal point by the letter "r" (denoting radix point), thus:

iTOA+nr123ss

to print three digits to the right with no decimal point printed. Note that if the number in the MRA should unexpectedly exceed unity, then the resulting digits to the left would be printed along with the desired three to the right with no indicated decimal point.

6. Repetition of Output Requests

It is often desirable to insert output requests at different points within the same program. Provision has been made in the CS computer so that the sample number does not have to be repeated if that sample number and the desired output medium are the same as the one preceding it in the written program. Thus if a programmer has written:

iTOA+i123.1234ss (5)

and writes the next output request as

iTOA

the form of the output will be the same as for (5).

7. Scale Factors

It is possible to have the number in the MRA multiplied by a scale factor before that number is printed out. The permissible scale factors consist of exponentials with base 2 or base 10. As many such factors may be used as desired. The factors should be written after the sample number, and each factor should be preceded by an x. (N.B. -The "+" sign should not be written for positive exponents.)

Thus if the programmer desired to have the number in the MRA multiplied by $2^5 \times 10^{-3}$ before printing it out, he would write his request in the following form:

$$\text{iTOA} + \text{i}123.1234 \times 2^5 \times 10^{-3} \text{ ss}$$

8. Special Characters

Provisions are available for printing out special characters (such as a decimal point, space, tab, sign, or carriage return) by themselves (i.e., without printing some number with it). A request such as

iTOA c

will cause a carriage return to be typed on the "direct" typewriter. The significant characteristic of this request is that no sample number is indicated.

The symbols for the special characters are the same as those introduced above. Only one symbol should be used in any given request. Thus the request iMOA+. will not record a plus sign and a decimal point on the delayed printer. To obtain such a sequence of characters the programmer should request:

iMOA+

iMOA.

Provision has not been made in the CS computer for repeating requests for special characters as discussed in section 6 of this chapter since the saving of programmer's time would be trivial. Consequently a request for just a special character (no sample number) should always include the necessary symbol for the desired character.

9. Magnetic Tape Stop Character

In making use of magnetic tape for delayed printing it is desirable that each programmer terminate the Flexowriter printing from magnetic tape to avoid printing information recorded subsequent to his own. A special "STOP CODE" character, which can be recorded on magnetic tape, will automatically stop the delayed printout equipment. It is possible for a programmer to provide this stop code character automatically as follows.

The output request

iMOA end

can be used by a programmer when he has completed his recording on magnetic

tape to mark the end of his information. The "iMOA end" request records successively on magnetic tape a shift to lower case, stop code character, two carriage returns, and another stop code character.

10. Page Format

It is often desirable to arrange a set of numbers that are to be printed out according to a predetermined page layout. The pertinent information for such an arrangement specifies how many numbers are to be printed per line, how many spaces are desired between numbers, and how many numbers are included in the set. Thus three counters are required for keeping account of these numbers.

This counting can be set up automatically in the CS computer by means of the instruction

i FORMAT   or, more briefly,   i FOR

followed by a tab or carriage return, followed by the three pertinent counts separated by tabs or carriage returns. Thus the request:

i FOR
α
β
γ

will set up counters to provide α numbers per line, β spaces between numbers, and γ numbers per block. α, β and γ are positive integers and should be written without a decimal point. α and β are restricted by the requirement that the number of characters per line on the Flexowriter should not exceed 154. If a programmer sets $\beta = 0$, he will obtain a tab between his numbers. γ can be any positive integer not exceeding 32,767. A typical request would be:

i FOR
+ 10
+ 2
+ 95

giving a block of 95 numbers with 10 numbers per line for 9 lines, 5 numbers in the last line, and two spaces between each number.

After γ numbers have been printed out, two carriage returns are typed and the counters are reset ready to lay out a new block. If the programmer prints out fewer than γ numbers, the carriage of the Flexowriter will be left in a position determined by the last number printed out.

To make use of the counting facility described in the preceding

paragraphs, the programmer need only use the letter f as his terminal character (instead of the characters suggested in section 4 above). Thus the request

i TOA + nl.2345f

will print out a number in a form already described. After the number has been printed, the counter γ will be increased by 1 to see if a block has been completed. If it has, two carriage returns will be typed and the counters will be reset. If not, the counter α will be increased by 1 and a test will be made to see if a line has been completed. If it has, a carriage return will be typed and α will be reset. If not, β spaces (or a tab if $\beta = 0$) will be typed and the carriage of the Flexowriter will be left alone awaiting further output instructions.

Thus the request i FOR, when executed, sets the counters and calls in the routines needed to effect the counting. Any subsequent i FOR requests will simply reset the counters. It should be emphasized that the i FOR request does not return the carriage of the Flexowriter to its left-hand margin (since at this time the routine does not know what medium the programmer has selected). For this reason the programmer should be sure to return the carriage accordingly by a special request such as i MOA c (as a rule a programmer may assume that before his program is run on the machine the carriage has been returned by the computer operator to its left-hand margin).

The actual page layout counting is done in response to the suffix f used as the terminating symbol. Obviously any request using any other terminating symbol will not affect the counters and hence may spoil the layout unless planned by the programmer.

```
fc TAPE NO. 123-45-6789 SMITH
(24,6)                         s2,itas2+3             icr7       icap1+c    isps1      icts2+3    isps1+2
icr6           icap1+14+c                 isps1       icts2+4    isps1+2    ispw1      isp0
s3,itas3+3     icap1+3         itst1      itst1+2     icap1      itss4+3    icap1+4    iadt1      imrn2+6
isps4          icap1+5         idvs4+3    imrp1+6     imrp1+6
   .              .               .          .           .          .          .          .          .
   .              .               .          .           .          .          .          .          .
   .              .               .          .           .          .          .          .          .
||||||||||        .               .          .           .          .          .          .          .
   .              .          (MAIN BODY OF PROGRAM OMITTED FOR BREVITY)                   .          .
   .              .               .          .           .          .          .          .          .
   .              .               .          .           .          .          .          .          .
p1,+.          -5.74x10^-5                +.034274 4               +1716.226                        n2,+1.0
+.             +1.0x10^-6      +.         +.                       b1,ican2
iTOA+i123.1234ss
icap1
iTOA
icai1
iTOA-i12345.67c
isps2
i START AT a1
```

AN EXAMPLE OF A PROGRAM TYPED FOR THE CS COMPUTER

ERRORS AND POST-MORTEMS

I. Machine Errors

In spite of all precautions, a program may not perform as intended. It is not always easy to decide immediately whether or not the program is at fault, but there are certain possible indications which will help in the decision. For example: the machine will probably be at fault if the same program is run twice under the same initial conditions with different results each time. The mere fact that a program has worked correctly on several occasions with different data does not absolve the programmer from blame if that program should suddenly perform poorly with a new data tape. This may have been caused by the new data tape which might contain unsuitable data or which might have covered over significant storage information, etc. Whirlwind contains checking devices known as parity alarms, check alarms, etc., which are not supposed to reveal programming errors but only machine malfunctions. However, if certain test programs which make use of every part of the machine perform satisfactorily, then the chances are very good that the computer is working properly.

In charging programmers with machine time used, that time which the programmer has lost due to parity alarms, auxiliary equipment failure, etc. is credited to his account, so to speak, with certain reservations (e.g., if a program runs for 20 minutes and then has a parity alarm, the programmer should not expect to be credited with 20 minutes time, since his program should have had a rollback feature so that no more than the last few minutes of the 20 minute period is lost). Rollback means a periodic recording of the contents of storage on a form of secondary storage so that in case of machine error, such as a parity alarm, the contents of storage might be "rolled back" to what it was after the last recording on secondary storage preceding the machine malfunction. The program can then be restarted. (See Advanced Chapter.)

II. Definition of Post-Mortem (PM)

When the performance of a program leads to an undesirable situation which has been attributed to a programming mistake (or tape preparation mistake, etc.), the contents of particular storage registers, the MRA, etc., might be valuable in helping the programmer diagnose the mistake. A post-mortem is a special routine which records this data in some readable

form after the program has stopped or has been stopped by the operator.

III. Programming Errors

A. Loop - One type of programming error that can occur which may not stop the computer is the repetition of the same set of instructions in excess of the maximum number of repetitions the programmer would normally expect. When the operator realizes that the program has gone into a "loop" (indefinite repetition of a cycle of instructions) he stops the computer by pressing a special button and obtains a PA Post-Mortem (to be discussed in this chapter). Pressing this special button causes the program to stop on the next isp, icp (-), or ict (-) encountered. Consequently, if a PA Post-Mortem shows that a program has stopped on any one of the three operations mentioned above, then it is very likely that the program was in a loop. This PA PM will give the programmer some information about the loop. However, the programmer might wish to get a post-mortem of other registers in storage whose contents might aid him in diagnosing the loop.

B. Unsatisfactory Results - If a program stops as predicted but gives either no results or results which are not what the programmer expected, the programmer might ask for a post-mortem to aid him in diagnosing the program's ills. The programmer might wish to change and then re-run his program so that a post-mortem may be obtained at a critical point in its performance.

C. Unexpected Stop - A program may stop unexpectedly in the wrong place in the program or it may stop because of any one of the following errors listed under its respective alarm and, in most cases, followed by the instructions which could cause this error:

1. Check Order Alarms
    a. Counter not provided for by the PA* is selected (this can occur only if the "j" in isc j has been modified by the program).
    b. Exponent of N(MRA) $\geq 2^j$ where j refers to the (30-j,j) notation (provided al is not a buffer): its al, iex al
    c. $0 < |C(al)| < 1/2$: iad al, isu al, imr al, idv al

---

* PA (programmer arithmetic) will be discussed in a subsequent chapter.

d. When control is transferred to an undefined (illegal) instruction, an alarm occurs on the undefined instruction. (There are three undefined instructions and the decimal values of their operation sections are 0, 30, and 31 respectively.)

2. Divide Error Alarm

   C(al) = 0: idv al

3. Arithmetic Overflow Alarms

   a. C(index register)+m > 32,767 : ici m
   b. C(index register)-m < -32,767 : icd m
   c. |C(index register)+C(al)| > 32,767 : iat al (See Advanced Section of Chapter III)
   d. |i| = 32,767 before the ict is executed : ict al
   e. |Result| > 7.0×10^{9863} or |Result| < 7.1×10^{-9864} : imr al, idv al
   f. If al is a buffer, then alarm j could occur: iad al, isu al
   g. The contents of the index register could be large enough to cause an alarm, i.e., when al+c > 32,767 : its al+c, iex al+c, ica al+c, ics al+c, iad al+c, isu al+c, imr al+c, idv al+c, isp al+c

IV. PA Post-Mortem

A PA Post-Mortem furnishes the programmer with extremely useful information if his program has not performed satisfactorily. This information is based on the contents of storage after the program has performed and stopped. If a programmer desires no other type of post-mortem than a PA PM, he need only request it on his Performance Request form. If the programmer desires post-mortems other than the PA PM, then he must have submitted the proper PM Request for these post-mortems (see section V). In response to such a request, the programmer usually obtains, automatically, a PA PM in addition to the post-mortem requested. When a PM tape is used, the machine checks to see whether there is a PA in storage and whether it has been used:

A. If there is no PA in Core Memory (CM)* , a check is made to see

---

* CM will be discussed in a subsequent chapter.

whether there are any interpreted instructions (ii)* or Generalized Decimal numbers (G.D. numbers)* requested on the PM tape. If there are, then the words "no PA" are printed out instead of a PA PM printout. If there are nether ii nor G.D. numbers, then nothing at all will be printed out.

B. If there <u>is</u> a PA in storage but it has not been <u>used</u> then the words "PA unused" will be printed out instead of a PA PM printout. If the PA has been used, then a PA PM will be printed out. The PA PM will then be followed by the other types of post-mortems requested on the PM tape.

In the case where the programmer requests a PA PM without using a PM tape, he will receive a "pushbutton PA PM". This is executed in the same way as the PA PM obtained via the PM tape <u>except</u> that in this case there is no way of checking for G.D. numbers or interpreted instructions since there is no PM tape. The information furnished by a PA PM might appear like this: (all addresses in example are decimal)

```
4-17-54
100-04-367
John Paul Jones
(24,6) PA PM
stopped at 279    279| iex 493+c    499| -.12345678| +07    MRA| +.324566109| +22
1624| b| +.274901627| +14    1b| -.726401278| +11    2b| -.360072483| -35
1633| 0| 0,10    1||| 3,12    2| 0,0    3| 0,0    4| 0,0    5| 0,6
509| icp606    615| isp285    320| isp221    246| icp255    274| icp278
```

<u>line 1: date</u> ex: 4-17-54

<u>line 2: tape no</u>. ex: 100-04-367

<u>line 3: name</u> ex: John Paul Jones

This title information may or may not appear and if it does appear it may be in a different form or order. <u>The programmer may select his own title by having it inserted on his PM tape</u> (see "Preparation of PM Tapes" in Section V; <u>in this case, the entire title will appear on one line</u>.

<u>line 4: title</u> ex: (24,6) PA PM

---

* ii and G.D. numbers will be discussed in subsequent chapters.

*line 5: stop line

This line will always contain information about the interpreted instruction which was being executed or which was most recently executed. 279 is the address of the instruction that was being performed at the time the program stopped; iex 493+c is the instruction that was being performed; 499 is the effective result of 493+c (the address section of the instruction being performed); -.12345678| +07** is the G.D. number in registers 499 and 450; +.324566109| +22 is the content of the Multiple Register Accumulator. Notice that the mantissa of a number in 2 registers of storage is printed as an 8-digit number and the mantissa of a number in a buffer or the MRA is printed as a 9-digit number. (In the example above, the program stopped because the MRA contained a number that exceeded the capacity of storage and the iex attempted to store the content of the MRA in registers 499 and 500.)

*line 6: buffer line

line 7: counter line

1633| 0| 0,10 means that cycle counter zero contains a zero in the index register and 10 in the criterion register, and that register 1633*** and 1634 are the index and criterion registers respectively of counter zero; 1||| 3,12 means that cycle counter one contains a 3 in its index register and 12 in its criterion register and that cycle counter one is the most recent counter used (indicated by the number preceding the 3 vertical bars). If no counters are called for, the "counter line" will not be printed. The maximum number of counters per line is 10.

---

* See section VII for a more detailed discussion of lines 5 and 6.

** This means $-.12345678 \times 10^7$ (-.12345678 is often referred to as the mantissa of the number).

***In an advanced chapter, formulas will be presented which will enable the programmer to determine the location in storage of the various sections of his program, such as PA, cycle counters, etc. At that time the programmer will be able to verify 1633 and 1634 as the location of the zero-th cycle counter pair in the example given above.

line 8: jump table

The five most recently executed transfers on control (due to isp, icp (-)) and their locations are enumerated; the most recently executed transfer of control appears at the extreme right. If less than 5 transfers of control have been made by the time the PM was taken, only those will appear in the print. If no transfers of control had been made, then the words "no jumps" will be printed on this line. 509| icp606 means that the fifth last isp, icp (-) instruction that was executed is icp606 and it was contained in register 509 etc.; the last one executed is icp278 and is contained in register 274.

V. Other Post-Mortems

Before his program is run, the programmer would do well to prepare a post-mortem tape which he might want to use under certain circumstances. The programmer must specify on these PM tapes: (1) which registers he wants to examine (e.g., registers 100-130), (2) how he wants their contents printed (e.g., as instructions, numbers, or single-register integers; thus far such integers have only been used as the contents of the index and criterion registers) and (3) which mode he wishes used (e.g., direct, or delayed printing; the latter will be described in an advanced Chapter). The computer furnishes the programmer with the desired information automatically after the operator reads in the programmer's PM tape. After a Post-Mortem has been given, the contents of CM is automatically restored to whatever it was immediately preceding the Post Mortem. A PA PM is automatically given with each PM tape request under the conditions given in section IV above. However, the programmer may obtain a PA PM without preparing a PM tape if he doesn't require any other type of post-mortem.

A. Preparation of PM Tapes

Tapes are prepared on the standard Flexowriter typewriters by depressing the proper keys while in punch and type mode (see fig. 1 at end of chapter). A description of the information which must appear on these tapes follows:

1. fp must appear as the first characters of the tape. (These letters represent flexo post-mortem.)

2. Any title the programmer desires (e.g., name, date, tape number, etc.) must follow the fp notation of 1. Such a title, if used, must be followed by a carriage return.

3. The next characters must indicate the mode desired:

   a. del will give delayed printer. (One may punch out as much of the words "delayed printer" as he wishes but at least the first three letters must appear.)

   b. dir will give direct printer. (As in 3a. above, at least the first three letters must appear.) If no mode in indicated on the PM tape, then the programmer will automatically get delayed printer.

4. Next the programmer indicates whether he wishes decimal or octal addresses in both the locations of words and the address section of instructions.

   a. dec will give decimal addresses (whole word "decimal" may be used instead).

   b. oct will give octal addresses (whole word "octal" may be used instead).

If neither octal nor decimal is indicated on the PM tape, the programmer will automatically get decimal addresses.

5. Next the programmer indicates the address of the initial register of a block of registers whose contents is desired. This address may refer to CM or to Drum Memory (DM)*. When a tape is read into storage the first thing that occurs is that the contents of CM is recorded on DM Group 0. Consequently, as far as the programmer is concerned, he may obtain whatever the

---

*Drum Memory will be described in advanced Chapter.

<u>contents of register X was in CM before the PM tape was read in by requesting the contents of register X of DM Group 0.</u> This may be done in any one of several ways:

(ex. a) 0 - 45 means register 45* of DM Group 0 (Actually what we get is register 45 of Core Memory since Drum Group 0 contains whatever was in CM before the PM tape was read in.) 8 - 88 would mean register 88 of DM Group 8. (The "0" in 0 - 45 above could have been omitted. Whenever Drum Group 0 is selected with this type of notation, the zero and dash are optional.)

(ex. b) 16472 means register 16472 (decimal) of Drum Memory. (This is equivalent to using 8 - 88 in ex. a.)**

(ex. c) 0.40130 means register 40130 (octal) of the Drum. (This is equivalent to using 8 - 88 in ex. a or 16472 in ex. b.)**

6. Any one of the following 2 letter combinations may follow section A5. above to indicate the <u>form</u> in which the programmer wishes to have his instructions or numbers printed out:

a. ii means interpreted instructions ex: ica 47
b. wi means Whirlwind instructions ex: ca 47
c. of means octal fractions ex: 0.01763
d. di means decimal integers ex: 679
e. df means decimal fractions ex: -.6384
f. gd means generalized decimal numbers
ex: +.12345678| +22

7. This is followed by the <u>address</u> of the <u>final register</u> of the block of registers whose contents is desired. The address of the final register is indicated in the same way as the initial register (see A5. above).

8. The <u>terminating character</u> of the tape consists of two vertical bars ||.

---

*Decimal or octal depending upon what is selected as described in section 4.

** This notation will be described in an Advanced Chapter on Drum Memory.

B. Multiple Requests

1. Example One

One may use the address of a final register of a block as the initial address of a new block of registers in steps 5-7 above. If

0-45 ii 0-700 wi 0-823 df 0-1073

were typed for steps 5-7 in the preparation of a PM tape, the programmer would obtain the following:

register 45 CM to register 700 CM as interpreted instructions,
register 700 CM to register 823 CM as Whirlwind instructions,
register 823 CM to register 1073 CM as decimal fractions.

2. Example Two

The programmer need not use the address of a final register of a block as the initial address of a new block as in Ex. 1 above. If

0-45 ii 0-700 0-750 wi 0-823 1-850 df 1-1073 gd 2-97

were typed for steps 5-7 in the preparation of a PM tape, the programmer would obtain the following:

register 45 CM to register 700 CM as interpreted instructions,
register 750 CM to register 823 CM as Whirlwind instructions,
registers 850 to 1073 of Drum Group 1 as decimal fractions,
register 1073 of Drum Group 1 to register 97 of Drum Group 2 as Generalized Decimal numbers.

3. Example Three

The programmer may desire several different modes (e.g., delayed printer, direct printer) and/or both octal and decimal addresses. The following example shows how easily this may be done:

fp John Sampson April 3, 1954 Tape 263-49-16
oct 127 ii 700 dec 500 wi 700 dir oct 1-340
gd 1-751||

The above would give the programmer the following:

a. The title on the first line will be recorded on the delayed printer. (Since no mode was selected, the programmer automatically gets delayed printer.)

b. oct 127 ii 700 gives the programmer registers 127-700 (octal) of CM as interpreted instructions with octal addresses on the delayed printer. The location of these instructions will also be given in octal.

c. dec 500 wi 700 gives the programmer registers 500-700 (decimal) of CM as Whirlwind instructions with decimal addresses on the delayed printer. The location of these instructions will also be given in decimal. (Whenever changing from one mode to another or from one number system to another, place your new information, e.g., dec, dir, etc. <u>immediately</u> <u>before</u> the group of registers affected. dec 500 wi 700 is an example of this.

d. dir oct 1-340 gd 1-751 gives the programmer registers 340-751 (octal) of Drum Group 1 as generalized decimal numbers on the direct printer. The location of these numbers will be given in octal.

<u>NOTE</u>

The following specific cautions are given for the preparation of a PM tape so that the programmer may be able to prepare his own tape if the need arises.

C. <u>Cautions in the Preparation of a PM Tape</u>

1. In order for CS II to be able to distinguish the last digit of one number from the first digit of the next number (as in example two section B above where 0-750 follows 0-700) some character other than a number <u>must</u> be used to separate them. a space, tab, or carriage return will serve the purpose adequately. <u>The programmer should avoid writing the above example as 0-7000-750</u>.

2. The following numbers 1., 2., etc. are associated with those in the section entitled "Preparation of PM Tapes."

a-1. The letters in fp should not be separated by a character.

b-2. fp and the title must <u>not</u> be separated by a carriage return. However, the title <u>must</u> be followed by a carriage return.

c-3. The letters in del, etc. should not be separated by a character.

d-4. The letters in dec,etc. should not be separated by a character.

e-5. The digits and hyphen in 0-45, 0.12345, 16421 should not be separated by a character.

f-6. The letters in ii, wi, etc. should not be separated by a character.

g-7. Same as e-5. above.

h-8. The final two vertical bars should not be separated by a character.

3. Backspacing and manual interference with the Flexowriter carriage are illegal. Generally speaking, "if the typed copy of the PM tape looks correct, then the tape probably is correct." The programmer may combine as many PM tapes as he wishes into one tape.

VI. Flad Table

It is often convenient to know the absolute addresses assigned to the floating addresses used in a particular program. This is especially useful in checking a program with the results of a post-mortem since addresses in a post-mortem appear as absolute addresses. For this reason a floating address table is available to the programmer for his particular program.

Suppose a program had contained the following floating addresses: a1, a5, a5+3, a7, g2, g10, g10+6, s21, s22. Then the program's flad table might appear like this:

assigned flads

a1=108	a5=122	a7=64
g2=217	g10=250	
s21=718	s22=465	

The flads are listed alphabetically and then numerically according to the number in the flad. Notice that a5+3 and g10+6 do not appear in the flad table. The reason for this is that since a5=122 then a5+3=125 and since g10=250, g10+6=256.

If the programmer should erroneously refer to floating addresses that have never been assigned (i.e., used as tags), let us say g3, f4, and t8, then the following data will automatically be printed in addition to a flad table, so that the programmer might have sufficient information to correct his error:

unassigned flads

f4 at 91

g3 at 72

t8 at 104, 163, 319

If a flad, let's say al, is erroneously used in such a way that it has more than one value, then no flad table will be recorded and the following will automatically be printed:

duplicate flad is al

Advanced Section

The following section is included in this chapter for completeness, but should be considered as part of the more advanced section of this manual.

VII. PA PM Continued (refer to illustration on page 4)

A. Line 5 of PA PM in greater detail

If register 279 had contained a buffer instruction instead of iex 493+c, the address section of the instruction would contain the "b" notation. Thus one might get 279| imr 3b but the section 499| -.12345678| +07 will be omitted since the contents of the buffer will appear on line 6.

As another example: if the number in 499 had been an <u>improper</u> G.D. number, i.e., if it had not been scale-factored (defined in advanced chapter), the contents of registers 499 and 500 would not have been printed out as a G.D. number but as two octal fractions (discussed in advanced chapter) in the form 499| $d_0.d_1d_2d_3d_4d_5$ 500| $e_0.e_1e_2e_3e_4e_5$.

The above two examples are included in the following table which tabulates all the possibilities for line 5:

PA PM TABLE

Operation section of "instruction being executed"	Contents of "register(s) referred to" in address section of instruction being executed
ica, ics, iad, isu, imr, idv, its, iex	a. Proper G.D. number is printed as G.D. number. b. Improper G.D. number is printed as two octal fractions. c. If the address of the "instruction being executed" refers to buffer storage, nothing is printed out since the contents of buffer storage is printed on line 6 of the PA PM. (Buffer discussed in advanced Chapter and in VII-B.)
same operations as above with "c" appended If there is no cycle block, the operation section above will be printed as a WW instruction.	a. If there is a cycle block, same as a and b above. (Idea of cycle "block" to be discussed in advanced Chapter.) b. If there is no cycle block, this section will not be printed.
ita, isp, icp	a. Interpreted instruction printed as such. (If the address of this instruction is buffer j, i.e., jb, then the address is printed as 1784 + j.)* b. Instruction printed out as WW instruction if (1) it is illegal (operation positions 0, 30, 31 decimal). (2) it is a cycle instruction and there is no cycle block.
icx If there is no cycle block, the operation section above will be printed as a WW operation.	a. Printed as two decimal integers** N| i, N+1| n if there is a cycle block. b. If there is no cycle block, this section will not be printed.

* 1784 happens to be the address of the first register of the PA block.

** Discussed in advanced Chapter

Operation	Printing
<u>icr</u>, <u>ici</u>, <u>icd</u>, <u>isc</u> If there is no cycle block, the operation section above will be printed as a WW operation.	This section will not be printed.
<u>ispc</u> If there is no cycle block, the operation section above will be printed as a WW operation.	a. If there is a cycle block, then (1) an interpreted instruction is printed as such. If the address of this instruction is buffer j, i.e., jb, then the address is printed as 1784+j. (2) Illegal instruction is printed as a WW instruction. b. If there is no cycle block this section will not be printed.
<u>iat</u>, <u>iti</u>, <u>ict</u> If there is no cycle block, the operation section above will be printed as a WW operation.	a. If there is a cycle block, then (1) an interpreted instruction is printed as such, (2) an illegal instruction is printed as a WW instruction. b. If there is no cycle block this section will be printed out as a WW instruction.
<u>sp</u>	This section will be printed as a WW instruction.
illegal instructions (operation positions 0, 30, 31 decimal) The operation section above will be printed as a WW operation.	This section will not be printed.

B. Line 6 of PA PM (Multiple Buffers Line)

This line may appear as follows:

1624| b| +.274901627| +14 1b| -.726401278| +11 2b| -.360072483| -35

1624| b| +.274901627| +14 means that the content of b (buffer zero - which may also be written 0b) is the Generalized Decimal number +.274901627| +14 and that the address of the first register of buffer zero is 1624*. (Buffer zero is the first available buffer.) Notice that the mantissa of a number in buffer storage is expressed as a 9-digit number.

1b| -.726401278| +11 means that the contents of 1b (buffer one) is the G.D. number following the 1b etc. The maximum number of buffers appearing in any one line of the PA PM is five. If no buffers have been called for, then line 6 of the PA PM will be omitted.

Drawings attached

A-58536

* In an advanced chapter, formulas will be presented which will enable the programmer to determine the location in storage of the buffer triples b, 1b etc.

A-58536
F2335

FIG. I FLEXOWRITER TYPEWRITER

VII Subroutines

In preparing a program to solve a problem on a digital computer, the programmer frequently will find that his program naturally breaks down into a series of groups of instructions, each performing some necessary operation. One or more of such groups often are written to perform the operation denoted in one of the blocks of the flow diagram for the solution of the problem. Examples of such operations are the extraction of roots of a number, the calculation of the values of a function for values of the independent variables, etc. If such operations occur in many different programs, much programming time will be saved if these routines are available to the programmer without the necessity of his preparing them. Such groups of instructions, which perform particular operations, are called subroutines, and a collection of such subroutines is usually called a subroutine library. Even if particular routines are not available in the subroutine library, the programmer may still find it desirable to write these himself as subroutines in his program, both to simplify the logical structure, and to save space if the same routine is to be used at different points in the program.

As an illustration of a subroutine, let us assume that the polynomial function ax^2+bx+c is to be evaluated for a particular value of x which is in the MRA. A program to evaluate this function would be

```
its b1      store x
imr a1      form ax
iad a2      form ax+b
imr b1      form ax^2+bx
iad a3      form ax^2+bx+c
```

which uses the registers

a1,	a	coefficients of the polynomial, which will be
a2,	b	particular numbers depending on the particular
a3,	c	problem.
b1,	+0.	storage for x

If we wish to make this subroutine a self-contained block, then an <u>isp</u> order will be needed to skip around the registers containing numbers,

as

```
    ,     isp   pl

    al,   a
    a2,   b
    a3,   c
    bl,   +0.

    pl,   its   bl
          imr   al
          iad   a2
          imr   bl
          iad   a3
```

This subroutine is now ready for insertion where needed in a program. If this subroutine is a member of the subroutine library, there is a punched paper tape containing these instructions kept in a file, and this tape can be copied into the main program wherever desired. If a programmer is using a subroutine from the library, he must carefully ascertain exactly what the subroutine will do, how many registers it will occupy (if storage space is critical), where it places the result or results, what its accuracy is (if this is a factor), etc. If he is interested in the time required by his program, then the time required by each subroutine, if it can be determined, will be necessary.

RELATIVE ADDRESSES

If a floating address is used in a subroutine, whether written by the programmer or obtained from the subroutine library, the programmer must avoid using this same floating address in other parts of the same program, since the CS computer cannot handle the ambiguous situation of one floating address corresponding to two different absolute addresses. Since several subroutines from the library might be used in the same program, this also means that all library subroutines would have to use different floating addresses (and none could be used twice in the same program). For these reasons, floating addresses are not used in library subroutines. Also, absolute addresses cannot be used in library subroutines since the programmer must be permitted to place such subroutines at any point in his program. However, references to other registers in the subroutine are usually necessary; for this purpose relative addresses are

used. Thus, a register is labeled not by an absolute or floating address, but by its position relative to some arbitrary register called the <u>reference</u> register, which is usually the first register of the routine. Relative addresses are indicated by the <u>suffix</u> r, i.e., 3r refers to the third register after the first register of the subroutine.* When the program tape is fed into the machine, relative addresses are converted by the machine to absolute addresses by adding the relative address to the absolute address corresponding to the <u>reference</u> register. If the above example is written in terms of relative addresses, we have:

```
 0r,   isp   9r
 1r|         a
 3r|         b
 5r|         c
 7r|         +0.
 9r|   its   7r
10r|   imr   1r
11r|   iad   3r
12r|   imr   7r
13r|   iad   5r
```

The 0r, is used to specify the reference register, as will be explained in the following paragraphs. The 1r| assigns the instruction (or number) that follows to the register (or pair of registers) whose absolute address is the reference register plus one.

Not all the instructions or numbers in such a subroutine need be preceded by relative addresses. The use of relative addresses is similar to the use of absolute addresses in that counting of registers is required; if an instruction or number is omitted by the programmer, it may be necessary to renumber the registers and the cross-references in the routine after the insertion of the desired material.

There are two ways to indicate to the machine the absolute address of the reference register: (1) If the subroutine is to be started in a certain absolute register, say 100, then the programmer should write 100|0r, followed by the first instruction of the routine. Thus if it were desired to have the above subroutine begin at register 100, the

---

*Note: The relative address 3r should not be confused with the floating address r3.

programmer could write

```
     100|0r,     isp  9r
                 a
                 b
                 c
                 +0.
                 its  7r
                 imr  1r
                 iad  3r
                 imr  7r
                 iad  5r
```

This would appear in the machine as

```
     100|  isp   109
     101| }
     102| }  a
     103| }
     104| }  b
     105| }
     106| }  c
     107| }
     108| }  +0.
     109|  its   107
     110|  imr   101
     111|  iad   103
     112|  imr   107
     113|  iad   105
```

(2) If a programmer using floating addresses wishes this subroutine to start in al, he may write

```
     al,0r,        isp 9r
                1r|a
                3r|b
                5r|c
                7r|+0.
                   its 7r
                   imr 1r
                   iad 3r
                   imr 7r
                   iad 5r
```

Note: we could omit the relative address assignments 1r|,3r|, etc.

Since subroutines in the library have the Or, of the first address punched in the tape, the programmer can simply write "100|" or "al," before indicating that the subroutine is to be inserted at this point. Actually the "Or," is superfluous after the floating address tag "al," since the comma in a floating address tag makes the register so tagged the reference register. To indicate that a subroutine, say number 10, from the library is to be inserted at a particular point in the program, the programmer may write "LSR tape no. 10" on the line following the al, or the 100|. This may appear on the typewritten copy of the program and will be punched on the paper tape. The library tape containing the subroutine is then duplicated on the program tape. At the end of the subroutine, may be typed the words "END OF SUBROUTINE". These two groups of words are used only to indicate on the typewritten sheet the positions of various library subroutines (LSR), which helps make this copy of the program easier to follow. The machine ignores lines starting with LSR and END...

Unlike floating addresses, the same relative addresses may be used at many points in a program. In each block of instructions in which relative addresses are used, the reference register is the most recent one which contains a comma in the address-tag section, e.g., "al," in the above example. If we wished to evaluate the above polynomial for two values of x, say x_1 and x_2, stored in register d1 and d2, and to type the results on one line, we could write

```
    ispcl
d1,x1
d2,x2
c1,ica d1
c2,isp 9r  )
    a      |
    b      |
    c      |
    +0.    |
    its 7r |
    imr 1r |
    iad 3r |
    imr 7r |
    iad 5r )
```

```
     iTOA+nl.2345t
c3,ica d2
c4,isp 9r  )
     a      |
     b      |
     c      |
     +0.    |
     its 7r |
     imr 1r |
     iad 3r |
     imr 7r |
     iad 5r )
     iTOA+nl.2345c
```

If this program were Library Subroutine tape number 10, then the programmer would get the same program by writing

```
     ispcl
d1,x1
d2,x2
c1,ica d1
c2,
LSR Tape No. 10
     iTOA+nl.2345t
c3,ica d2
c4,
LSR Tape No. 10
     iTOA+nl.2345c
```

When this program appears in the computer, the absolute addresses in corresponding orders of the subroutine in its two positions will be different, since the reference registers are different in the two cases.

CLOSED SUBROUTINES

Obviously, it is wasteful of storage registers to place the same subroutines at two or more points in storage. Some saving could be realized by using floating addresses to tag the registers containing the constants in the subroutines, and then placing these at only one point in the program. For subroutines written by the programmer, this is feasible, but for library subroutines it would require changing these routines, which we wish to avoid. In addition, this probably would not amount to a substantial saving, since the constants in a subroutine normally do not occupy many registers of the routine. For these reasons, a special order has been built into the CS computer which permits the

programmer to leave his main routine, go to a subroutine to perform some particular operation, and then return to the next register of the main program. This order is ita.

ita al	transfer address	transfer, into the address section of the instruction in register al, the address that is one more than the address of the register containing the last isp (or icp with N(MRA)<0)

To illustrate the use of the ita al instruction, suppose we rewrite the program:

```
al,ica dl          pick up x1
   isp c3          go to subroutine
   iTOA+nl.2345t   print the resulting N(MRA) followed by a tab
   ica d2          pick up x2
   isp c3          go to subroutine
   iTOA+nl.2345c   print N(MRA) followed by a carriage return
     :             go on with program
c3,0r, ita 6r
   its 13r
   imr 7r
   iad 9r
   imr 13r
   iad 11r
   isp 0
   a
   b
   c
   +0.
dl,x1
d2,x2
```

Note: It does not matter what address is initially written in the isp instruction in 6r, since the ita instruction will write the correct return address in this instruction whenever the subroutine is entered by and isp or icp from the main program. This new construction of the

subroutine also removes the necessity for the isp formerly required to skip around the group of constants in the subroutine.

The above subroutine, starting with the ita in c3, could be placed anywhere in storage and can be entered from any other point in storage. It is a completely self-contained block of instructions which carries out a particular operation when entered with a value of x in the MRA and returns control to the main program when this operation has been completed. This type of subroutine is called a "closed subroutine" as contrasted with those subroutines (given in the first examples of this chapter) which must be placed in the main program wherever they are required and are called "open subroutines". When a closed subroutine has been placed in storage, we may regard the isp order which "calls" in the subroutine (like the isp c3 above) as representing a new order, in this case an order which evaluates the value of the polynomial for the particular value of x in the MRA. The subroutines in the library are of the closed type and therefore have an ita as their first instruction.

A library of subroutines can be a great asset to the programmer, particularly since most problems can be written as a sequence of smaller standard operations which are probably represented in the subroutine library. Time is saved by using the subroutine library, not only in the composing and writing of the instructions for the routine, but also in checking the program for mistakes, since the library subroutine has been tested and should be correct. If the programmer writes his program as a sequence of subroutines called in by a main program, it may simplify the work of writing the program and each new subroutine can be tested separately as it is written making it easier to isolate and correct any mistakes.

PARAMETERS

The subroutine that we have just evolved will evaluate the given polynomial for any value (within the storage limits of the CS computer) of the variable x.

Let us now suppose that we have a program in which we wish to evaluate a number of different polynomials each of the same degree but with different sets of coefficients. We could make use of a group of subroutines, one for each case, but these subroutines would all have a great deal in common and it would be a waste to store each one in full.

What is required is to be able to modify one copy of the subroutine to meet each case as it arises, or to have the subroutine modify itself as required. Somehow the user must be able to specify the information that is needed to modify the subroutine. This specification is called a parameter of the subroutine.

PROGRAM PARAMETERS

When a parameter is provided by the program it is called a program parameter. For example, sets of coefficients for the polynomial subroutine could be stored in the main program to be used when needed. Such program parameters need not be stored a priori in the program, but they can actually be determined as part of the program. The variable x itself is a good example of a program parameter. The value of x for which the value of the polynomial is to be found may be determined by the program.

The most convenient place for the program parameter is in the MRA since the contents of the MRA are unchanged by the isp. However, only one such parameter can be stored in this way. Also, since the MRA is used in the subroutine, its initial contents must be processed immediately or be lost. This places restrictions on the subroutine.

The next most convenient place for the program parameter is in the main program in the register or registers following the isp to the closed subroutine. The reason that this location is convenient is that the address of the register following the isp is available to the subroutine through the mechanism of the ita instruction. Unfortunately the CS computer does not contain any simple means for setting the necessary addresses to refer to these registers. The procedure for handling such addresses makes use of instructions and techniques that will be described at a later stage in the development of the CS logic. Consequently, further discussion of the use of program parameters will be postponed for a later chapter.

PRESET PARAMETERS

The use of program parameters permits the variation of a parameter from time to time during the execution of the program. In the case of a library subroutine, however, it frequently happens that although it is useful to be able to choose a value of the parameter to suit a particular program, it is no hardship to forego the ability to change the parameter during the execution of the program. This means that the parameter can

be fixed before the calculation begins, and need not be reset each time the subroutine is called in.

The setting of the appropriate parameter for a particular program must be done when the program is read into the machine. The form of the subroutine which is kept in the library files must be applicable to all permissible values of the parameter. If the fullest advantage is to be taken of the subroutine we want to be able to copy it directly onto a program tape without having to make any alterations. The machine itself must therefore adjust the subroutine according to the parameter value chosen. It does this as the program is read into the machine, so that by the time the whole program is in the machine the subroutine is in the form required by the particular program. Because the parameter is fixed before the execution of the program begins, it is called a preset parameter.

Various methods have been used with various machines for incorporating preset parmeters into the subroutine. They all require that the value of the parameter be defined (i.e., identified and specified) by suitable punching on the tape preceding the portion of the tape on which the subroutine itself is copied. During the read-in process the machine remembers the identity and specified value of the preset parameter. Hence when it reads in the subroutine it is able to incorporate the preset parameter correctly into the subroutine. A list of the pertinent preset parameters are always included in the description of the subroutine. For the convenience of the programmer, preset parameters are usually chosen so that if their values are not specified they automatically assume their most common values (which should be zero for subroutines to be used in the CS computer).

In the CS computer, preset parmeters are identified by the fact that they consist of two lower case letters followed by a decimal integer less than 41 but greater than zero. The first letter must be one of the following three: p, u, or z. The second letter can be any letter other than o or l. Care must be taken that the sum over all parameter letter pairs of the maximum numbers used for each letter pair does not exceed 40. For example, if the preset parameters pa 2, za 5, za 7, pd 7, zg 4, ug 6, ug 8, and zz 11 were used in a given program, the condition would be satisfied because $2+7+7+4+8+11=39<41$.

A value is specified for a preset parameter simply by writing down the parameter followed by an equal sign, the value to be assigned, and finally a tab or a carriage return. For example, if it is desired to set the preset parameter pa 2 to the value +8, one simply writes in his program: pa 2 = +8 (followed by a tab or carriage return)

Preset parameters may be set equal to any positive or negative integer not exceeding 32,767 in magnitude (this integer must not contain any decimal point - see Chapter). In addition, a preset parameter may be set equal to a floating address, an absolute address, or to another preset parameter provided they are assigned suitable integral values elsewhere in the program (the floating address by being used as a tag, the preset parameter by being explicitly assigned an integral value).

The following subroutine evaluates a polynomial $a_n x^n + \ldots + a_1 x + a_0$ $(11 > n \text{ (integer)} > 0)$, where the coefficients $a_0, \ldots\ldots, a_{10}$ are stored in fixed registers in the subroutine. (Such a polynomial might represent an approximation to an arbitrary function where the accuracy of the approximation can be varied by varying n.)

```
ppl=
0r,ita 8r
   its 9r
   icr ppl
   ica 11r
   iad 33r - ppl - ppl + c
   imr 9r
   ict 4r
   iad 33r
   isp 0
   +0.
   +0.
   a10  \
   a9    |
   .     |    Actually, the numerical value of the
   .     |    coefficients of the polynomial would
   .     |    appear here.
   .     |
   a0   /
```

If the programmer wanted a 5th degree polynomial then he would write ppl = 5 . If he wanted a 6th degree polynomial, then he would write ppl = 6, etc.

TEMPORARY STORAGE

In many routines, certain registers are used only to hold intermediate results. For example, in the program on page VII-1, the initial

contents of register b1 is immaterial. When it is desired to evaluate the polynomial for some value of x, the value of x is stored in register b1 and the evaluation is carried out. If this particular value of x is not needed elsewhere in the program, the contents of register b1 again becomes immaterial. Such registers whose contents are set and used when needed during the execution of the program and are otherwise immaterial are called <u>temporary</u> storage registers.

A programmer who finds it necessary to make use of such registers will simply set aside certain registers for this use. For example, registers 46, 47, 48, and 49 in the program on page I-8 were set aside to hold temporarily the indicated intermediate results. If such a program were used in conjunction with one or more subroutines which also made use of temporary storage registers, then it should be possible by the very nature of a temporary storage register for the main routine and the subroutines to make use of a common set of registers. The number of registers in this set will be determined by the maximum number of registers whose contents are needed in the program at any given time.

The difficulty that arises in using such common sets of temporary storage registers is that we need some way for each of the routines to refer to the common set. In the CS computer the label 0t denotes the first of a set of consecutive temporary storage registers, 1t the second, 2t the third, etc. The label "0t" is usually abbreviated as "t" (i.e., 0t and t are synonymous; both refer to the same register).

Temporary storage registers are specified in the same manner as are preset parameters. The programmer simply writes, for example, t=1400 (or, t=a1) and henceforth any reference to a temporary storage register is determined. For example ica 2t becomes ica 1402, its t becomes its 1400. (Similarly with t=a1, ica 2t becomes ica a1+2; its t becomes its a1.) Note that once t (or 0t) has been specified then all of the other temporary storage registers are also specified. Hence the programmer must be careful to set aside in sequence the proper number of temporary registers that will be needed. The number of registers required for any library subroutine is always included in the associated specifications.

Thus, for example, if the main program needs three temporary storage registers and if we use two subroutines one of which makes use of five temporary storage registers and the other subroutine only one, then we

would set aside in our program a block of five registers to be used as temporary storage registers. If this block began in register 1400 (or a1), then in our program (usually at the very beginning) we would write t=1400 (or, t=a1). Just as for preset parameters, it is necessary to specify **in the program** the location of the temporary storage registers <u>before</u> reference is made to these registers in the program.

Making use of this new notation, we can rewrite the subroutine on page VII-7 as follows (it is assumed that somewhere in the main program before we use any of the temporary registers, t will have been specified):

```
0r,ita 6r
   its t      (Store x in the temporary storage
               registers t and 1t.)
   imr 7r
   iad 9r
   imr t
   iad 11r
   isp 0
   a
   b
   c
```

Thus, by referring to t (and 1t), the main program could, if desired, also make use of the same two temporary storage registers. Note that since numbers occupy two storage registers, the instruction "its t" will actually store a number in registers t and 1t. Hence the above subroutine requires that two registers be set aside in the main program for temporary storage.

Temporary storage registers should not be confused with floating addresses. Recall that floating addresses are written as a lower case letter followed by a positive integer (<u>not</u> 0). Thus 1t refers to a temporary storage register whereas t1 is a floating address.

VIII. Cautions

(1) It is important that programmers when writing a vertical bar (e.g., 34|) make it long enough so that it cannot be confused with the numerical one. In general, programmers using properly numbered forms do not need to indicate vertical bars at all, as tape room personnel will add them when necessary.

(2) The initial word following the tape title (excluding such things as (24,6), NOT PA*, temporary storage or preset parameter indications) will automatically (unless otherwise assigned) go into the initial register of storage (i.e., register 32). However, if one tape contains several titles, such as might occur if a tape contained several parameters, the initial word after ensuing titles (excluding as above) must have an absolute address assignment "32|" if the initial word is to go into register 32. Also, if it is desired that a floating address, e.g., a2, should have the same absolute address assignment in all the parameters, it must be indicated in each parameter, e.g., "36| a2,".

(3) In deciding the number of registers being used in a program, remember that instructions occupy one register and numbers occupy two registers.

(4) Remember that t and Ot are synonymous (lt is the register following t);

that +. and +.0 and +0. are synonymous;
that "0,", "r,", and "Or," are synonymous;
and that r and Or are synonymous.

(5) Consider the following section of a program:

```
34|  isp g7
g7,  ica 73
     isp 76
4r|  isp a2+7
a2,  +.3
     -.0055
```

When this appears in the computer it takes the following form:

```
34|  isp 35
35|  ica 73
36|  isp 76
```

---

* see Advanced Chapter

37	---	Registers 37 and 38 each contains the integer +0 *only* if storage was previously cleared and if nothing was previously assigned to registers 37 and 38. If you want to have the number +.0 in 37 and 38, program +.0 in 2r or in 37.
38	---	
39	isp 47	
40 } 41 }	+.3	
42 } 43 }	-.0055	

(6) Remember that all numbers must have at least a sign and a decimal point. Also, if powers of 10 or 2 are used with positive exponents, do not specify a + sign in the exponent of 10 or 2.

CORRECT NUMBERS	INCORRECT NUMBERS
+2.7	2.7
$+2.7 \times 10^{6}$	$+2.7 \times 10^{+6}$
+0.102659	0.102659
+. or +.0 or +0.	+0

(7) Since the maximum magnitude of a number that can be stored in 2 registers of storage is about 9×10^{18} and the smallest non-zero magnitude is about 5.5×10^{-20}, caution must be exercised to keep numbers within these limits when transferring to storage from the MRA.

ex 1) -2.7×10^{-23} will go into storage as a number between -2^{-63} and -2^{-64}. (see advanced Chapter)

ex 2) $+2.7 \times 10^{21}$ is too large for storage (a check order alarm will result).

(8) In order to utilize floating address programming so that insertions and deletions can be made without the bother of renumbering,

al,
a2,
a3,
.
.
.

is preferred to

al,
al+1,
al+2,
.
.
.

This follows from the fact that a1, a2, a3,... are <u>independent</u> floating addresses.

(9) In storing numbers, the instruction "its 1t" transfers N(MRA) to 1t and 2t. Consequently, the next number to be transferred requires the instruction "its 3t" which will transfer N(MRA) to 3t and 4t; similarly using floating addresses, "its a1" transfers N(MRA) to a1 and a1+1, and "its b2" transfers N(MRA) to b2 and b2+1. Suppose we desired to transfer the numbers in c1 and c2 into a sequence of registers beginning at b2:

<u>CORRECT</u>	<u>INCORRECT</u>
ica c1	ica c1
its b2	its b2
ica c2	ica c2
its b2+2	its b2+1

(10) The following example is given to distinguish between floating, temporary, and relative addresses:

```
a1, ica t1      (floating address)
    its a1+7    (   "        "   )
    its 2t      (temporary storage address)
    its t3      (floating address)
    idv t1+2    (   "        "   )
    its 9r      (relative address)
    isp r3+2    (floating address)
```

(11) If a floating address tag, such as e1, is preceded by an address assignment (disregarding carriage returns and tabs), then this must be either an absolute address or a relative address assignment.

```
A. CORRECT          B. CORRECT        C. INCORRECT
   134| d1,+.0         d1,+.0            d1,+.0
          +.0             +.0               +.0
   140|                6r|              d1+6|
        e1,+.4         e1,+.4            e1,+.4
```

(12) It is important to note that even though an absolute address may interrupt the consecutivity of the assignment of registers, nevertheless this consecutivity may be resumed by the use of the proper

notation as illustrated below:

```
 50| g7,ica b3     the absolute address will be 50
      its c2       the absolute address will be 51
200| ica z4        the absolute address will be 200
 2r| isp d1        the absolute address will be 52 since the
                   reference address for the r was determined
                   by the g7,
```

(13) Consider the following portion of a program: (this is correct if one wants +.0 stored in registers a2, a2+1).

```
a1,+.75
a2,+.0
a3,+.5
```

On the other hand, the following routine is incorrect if one is intending to put zero into (a2,a2+1):

```
a1,+.75
a2,
a3,+.5
```

In this case, a2 and a3 are assigned the same absolute address and therefore the same content, namely +.5.

(14) One of the most common errors is to use a flad in the address section of an instruction without using that flad as a tag anywhere in the program. (See Chapter VI, Section on Flad Table).

(15) If only one counter is to be used throughout the program, it is not necessary to use an isc operation to select it. Cycle counter (or line) zero is automatically available if any counter instruction (other than ici, icd, or icx) or the cycle counter letter "c" appears in the original program.

Cycle counter line zero is the first counter line available. The instruction isc 1 selects the second counter line, isc 2 selects the third counter line etc. However, from a programmer's point of view it may be easier to think of it in the following way:

isc 0 selects counter line zero
isc 1 selects counter line one
isc 2 selects counter line two
etc.

(16) If the value of m in the instruction icr m is set by the program, so that there is a possibility of m being set to zero, then the following expediency may be used:

```
                icr(n)      the variable icr instruction
                icd 1
             al,ict fl ----------> fl,___ \
                  |                  ___  |
                  v                  ___  } do cycle
          cycles completed           ___ /
                                   isp al
```

(17) In using cycle control, remember that when the instruction isp 100+c is to be executed, the instruction isp(100+i) is effectively formed and then executed. The other operations which may be used with +c (namely: its, iex, ica, ics, iad, isu, imr, idv) experience the same transformation as the example above except that the i is replaced by 2i.

(18) A preset parameter cannot be specified by anything that could occupy more than one register of storage. Thus it might be a floating address, absolute address, sum and/or difference of flads or of other preset parameters, instructions, or integers written without a decimal point (see VII-14).

ACCEPTABLE VALUES FOR PRESET PAR.	UNACCEPTABLE VALUES FOR PRESET PAR.
+50	+.50
ica g2	___
+h3+z4-y7	___
-627	-627.
pb2+c19	___

(19) Preset parameters must be specified in the program before they are referred to in the program:

CORRECT	INCORRECT
pp5=7	ica b4
ica b4	its c3+pp5
its c3+pp5	.
.	.
.	.
	pp5=7

(20) Temporary registers must be specified in the program before they are referred to in the program:

CORRECT	INCORRECT
t = f6	ics b2+4
ics b2+4	imr t
imr t	.
.	.
.	.
.	t = f6

(21) The reference register referred to in the relative address in an instruction is the last tag (the last address followed by a comma).

```
al,ica 5r         5r refers to the fifth register after al
   its 7r         7r refers to the seventh register after al
   imr 5r
   its 5r
   isp bl
   +26.13
   +0.

bl,isu 6r         6r refers to the sixth register after bl
   idv 8r         8r   "    "   "  eighth   "     "    "
   its al+5       al+5 "    "   "  fifth    "     "    al
   imr al+7       al+7 "    "   "  seventh  "     "    "
   its 10r        10r  "    "   "  tenth    "     "    bl
   isp d4
   +3.14
   -26359.28
   +.0

d4, .
    .
    .
```

(22) Single letters may not be written without separating them by a plus or minus sign:

CORRECT		INCORRECT
imr+t+c or imr t+c	(Since "+" may be omitted between operation letters and single letters.)	imr tc

LSR Tape #		Registers	Temporary Registers
I. OUTPUT			
OT1	(24,6) Print Decimal No. (Direct Printer)	126	0
OT2	Delayed or Direct Printing of (30-j,j) Numbers	207	0
OD1	Format and Print G.D.Numbers	229	0
OD2	Print (24,6) on Delayed Printer	147	0
OD3	Single Length Delayed Decimal Print	86	0
OD5	Delayed Octal Number Print AC	63	0
OS1	(24,6) MRA Decimal Column Scope Layout	221	6
OS2	AC Decimal Integer, Column Layout Scope Display	129	4
II. FUNCTION EVALUATION			
FU1	(24,6) Exponential e^x	51	12
FU2	(24,6) MRA Square Root	33	4
FU3	(30-j,j) Logarithm (lnx)	56	4
FU4	sin x cos x	65	4
FU5	(24,6) Hyperbolic functions Sinh x and Cosh x	66	12
FU6	Arc Sin (24,6)	69	6
III. MATRIX SUBROUTINES			
MA1	Largest Eigenvalue of a Real Matrix with Real Eigenvalues	154	0
MA2	Solution of n Simultaneous Linear Equations (Craig's Method)	145	13+8n
MA3	Rectangular Matrix Multiplication (30j-j)	90	4k+2m
MA4	Matrix Diagonalization (n x n) (24,6)	366	4n
MA5	Symmetric Matrix Inversion or Square Root Inversion	459	4n
IV. DIFFERENTIAL EQUATIONS			
DE1	Runge Kutta, One step, fourth order, n differential equations	58+6n	0
V. SPECIAL			
SP2	Extract Integral Part of MRA	52	2

LIST OF LIBRARY SUBROUTINES AVAILABLE MAY, 1954

INSTRUCTION CODE OF THE MIT CS COMPUTER

Instr.		
*ica al	clear MRA; add N(al)	N(al)→N(MRA)
*ics al	clear MRA; subtract N(al)	-N(al)→N(MRA)
*iad al	add	N(MRA)+N(al)→N(MRA)
*isu al	subtract	N(MRA)-N(al)→N(MRA)
*imr al	multiply and roundoff	N(MRA)xN(al)→N(MRA)
*idv al	divide	N(MRA)÷N(al)→N(MRA)

*its al	transfer N(MRA) into (al,al+1)	N(MRA)→N(al)
ita al	transfer address	transfer into the address section of the instruction in register al, the address that is one more than the address of the register containing the last isp (or icp with N(MRA) $<$ 0)
*iex al	exchange N(MRA) with N(al)	N(MRA)→N(al); N(al)→N(MRA)
*isp al	transfer control	take the next instruction from register al and continue from there
icp al	conditionally transfer control (conditional program)	ditto, if N(MRA) $<$ 0; if N(MRA) $>$ 0, take the next instruction in sequence

+0 $\leq$ m and j are integers $<$ 2048

icr m	cycle reset	set i = +0, n = m
ict al	cycle count	increase i by 1 and if this new value of $\lvert i \rvert \geq \lvert n \rvert$, then reset i=+0 and take the next instruction in sequence; if $\lvert i \rvert < \lvert n \rvert$, take the next instruction from register al
ici m	cycle increase	increase the contents of the index register by m
icd m	cycle decrease	decrease the contents of the index register by m
icx al	cycle exchange	exchange C(index reg.) with C(al) and exchange C(criterion reg.) with C(al+1)
iat al	add and transfer	add C(index reg.) to the C(al) and store the result in the index reg. and reg. al
iti al	transfer index digits	transfer the right 11 digits of the index register into the right 11 digits of register al
isc j	select counter j	selects counter j as reference for all subsequent interpreted instructions using a counter until execution of the next isc instruction--each counter has its own index and criterion

iTOA+nl.2345c	type out N(MRA) in normalized form followed by a carriage return
sp 0	STOP

*These are the only instructions which may be used with the cycle letter "c" in the form it appears, for example, in ica al+c.

Appendix II. fp TAPES AND THE PA PM

If a PA routine is in Core Memory at the time an fp tape is read-in, a PA PM is given automatically. When this occurs, the following conditions apply.

1. A CS I PA PM is always recorded on the direct typewriter.

2. A CS II PA PM is recorded on the direct typewriter or the delayed printer depending on what unit was requested by the fp tape. If both have been requested, the delayed printer is used.

3. If the program has not executed any interpreted instructions the PA PM prints: "PA unused".

4. If an fp tape requests the ii mode and there is no PA routine in storage , the PA PM prints "no PA".

5. If an fp tape requests the gd mode and there is no PA routine in storage, the PA PM prints "no PA", and the gd numbers requested are printed as if they were (24,6) gd numbers. If there **is** a PA routine in storage, the gd numbers will be printed in the number system of that PA routine.

6. If an fp tape contains no ii or gd requests and there is no PA routine in storage, nothing is printed about a PA PM.

7. If an fp tape requests any results on the delayed printer, the recording on Unit 3 will be terminated by two carriage returns.