

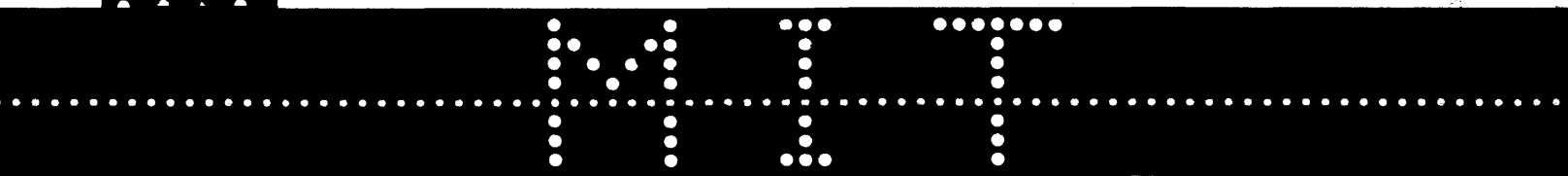


COMPREHENSIVE

SYSTEM

MANUAL

A SYSTEM OF AUTOMATIC CODING
FOR THE WHIRLWIND I COMPUTER



MIT

DIGITAL COMPUTER LABORATORY
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
CAMBRIDGE 39, MASSACHUSETTS

Memorandum M-2539-2

December, 1953

Revised April, 1954

Revised December, 1955

C O M P R E H E N S I V E S Y S T E M M A N U A L

A System of Automatic Coding
for the Whirlwind Computer

by

C. Adams
D. Arden
S. Best

H. Denman
J. Frankovich
F. Helwig

E. Kopley
J. Porter
A. Siegel

Digital Computer Laboratory
Massachusetts Institute of Technology
Cambridge 39, Massachusetts

FOREWORD

This volume represents the cooperative efforts of a number of former and present members of the staff of the M.I.T. Digital Computer Laboratory. Since this volume is based on work performed over a number of years, it is impossible to give proper credit to all individuals who were involved. The principal contributing authors were:

C. Adams	H. Denman	E. Kopley
D. Arden	J. Frankovich	J. Porter
S. Best	F. Helwig	A. Siegel

A portion of the contents of this volume was first prepared as a memorandum in the Digital Computer Laboratory in December, 1953. The original work was supported under ONR contract N5cri06001. The cooperation of the Mathematical Sciences Division of the Office of Naval Research in the preparation of this volume is gratefully acknowledged.

Since this volume is specifically written for the M.I.T. Whirlwind I Computer, it principally describes the two coding systems which are in current use; namely, the Comprehensive System (the interpreted CS Computer), and the basic Whirlwind code (WWI). The Comprehensive System employs multiple Whirlwind registers and programmed arithmetic to execute its instructions. However, it utilizes the basic Whirlwind Computer to implement each instruction. Thus, the Whirlwind Computer assumes different outward characteristics depending upon the coding system which is employed.

Although the coding systems described in this volume are based on the Whirlwind Computer, the basic coding and programming techniques may be employed on any stored program computer. It is hoped that this volume will help disseminate programming information to a larger group of people in the rapidly-expanding computer field.

F. M. Verzuh
Head, Scientific and
Engineering Computations Group

TABLE OF CONTENTS

INTRODUCTION

PART I. Introduction to Programming and Coding

CHAPTER I	THE CS COMPUTER - SIMPLIFIED VERSION
CHAPTER II	TRANSFER OF CONTROL - COUNTING
CHAPTER III	CYCLE COUNTERS - MODIFICATION OF ADDRESSES
CHAPTER IV	FLOATING ADDRESSES
CHAPTER V	INPUT AND OUTPUT
CHAPTER VI	ERRORS AND POST-MORTEMS
CHAPTER VII	SUBROUTINES
CHAPTER VIII	REVIEW

PART II. Advanced Coding Techniques

CHAPTER IX	SOME MORE FUNDAMENTALS
CHAPTER X	NUMBER SYSTEMS
CHAPTER XI	THE WHIRLWIND COMPUTER
CHAPTER XII	AUXILIARY STORAGE AND IN-OUT EQUIPMENT
CHAPTER XIII	PROGRAMMED ARITHMETIC
CHAPTER XIV	THE CONVERSION PROGRAM
CHAPTER XV	THE UTILITY CONTROL PROGRAM
CHAPTER XVI	AUTOMATIC INPUT-OUTPUT REQUESTS
CHAPTER XVII	GENERALIZED POST-MORTEMS

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.

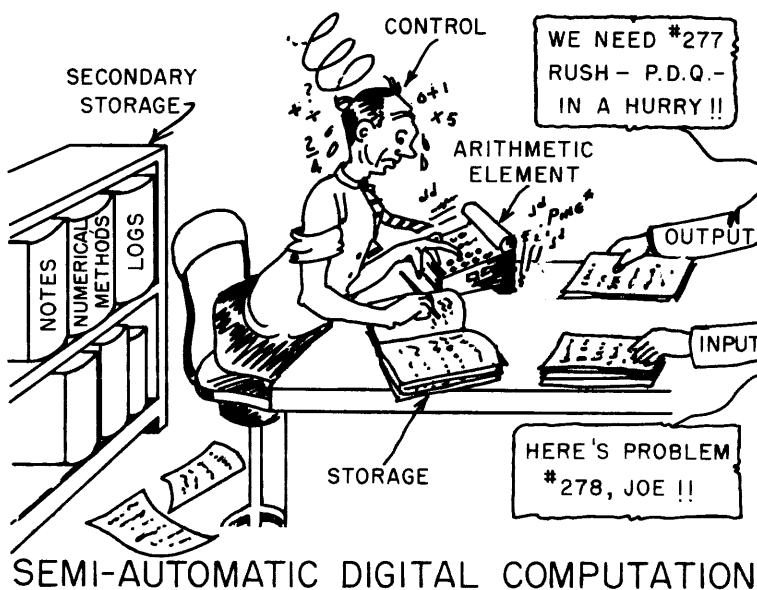
Modern electronics could speed this up almost a million fold. But

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.

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



PORTRAIT OF CHARLES BABBAGE

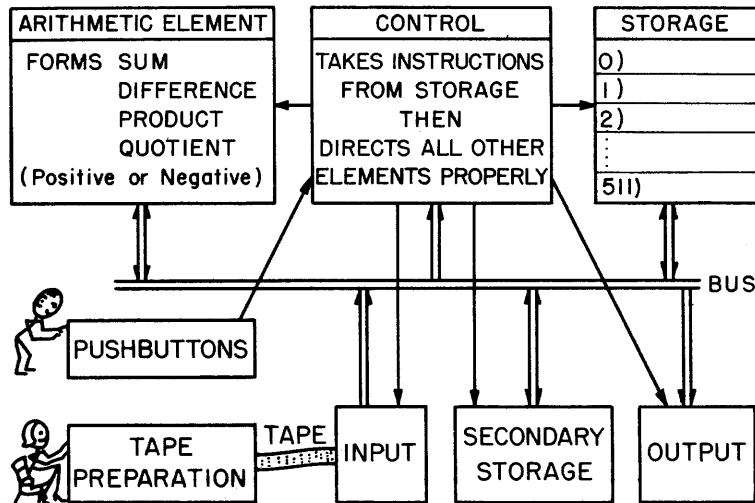


perform, automatically and without human intervention, a pre-described sequence of arithmetic operations and make predescribed logical decisions based on the results as it went along. In outlining his Analytical Engine in 1834, Babbage described all the important

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.

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



AUTOMATIC DIGITAL COMPUTATION

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 and 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.

When a human operator is to solve a program 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.

Part I of this manual deals with the Comprehensive System of Service Routines, which is usually abbreviated to "CS" or "CS computer". The instructions discussed in Part I are not performed directly by the electronic circuitry of the Whirlwind I computer, but are actually accomplished by more or less lengthy sequences of the actual instructions performed directly by the machine. The CS system is so designed that the programmer need not actually concern himself with this fact, but may use the instructions almost as if they were performed directly by the computer.

These CS or "interpreted" instructions have been so designed as to

make many commonly occurring functions very easy to program. Since they are actually composed of sequences of basic machine instructions, the same result can often be accomplished in less computer time but more program preparation time by coding in basic machine code. The basic machine is also capable of some functions which cannot easily be performed with CS instructions.

Since the basic principles of coding are illustrated very simply in the use of CS or interpreted instructions, Part I of this manual deals only with the CS computer. Part II describes the basic Whirlwind code and can be read independently of Part I, although thorough familiarity with Part I will help greatly in mastering Part II.

If it is desirable to get results quickly on a problem requiring only a small amount of computer time, the CS computer should be used as much as possible in order to minimize program preparation time. For problems requiring a large amount of computer time, careful consideration should be given to organization of the calculation and possible use of basic Whirlwind code. This can lengthen program preparation time by a considerable amount depending on the skill and experience of the programmer but may shorten computer time.

M-2539-2

COMPREHENSIVE SYSTEM MANUAL

PART I

Introduction to Programming and Coding

CHAPTER I: THE CS COMPUTER - SIMPLIFIED VERSION

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 primary storage element of the computer consists of approximately 1360 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 1360 different registers are 32, 33, 34, 35,...1390 and 1391.

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:

†129.7863

* The exact number of registers available can be determined by the tables given in Chapter XIII.

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.

<u>Symbol</u>	<u>Meaning</u>
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 instr. is obeyed
→	replaces
clear ...	set the contents of ... to zero

Thus " $N(MRA)+N(al) \rightarrow N(MRA)$ " 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 indicate that the instructions are interpreted by special programs stored in the Whirlwind I computer (see Chapter XIII). The time in milliseconds for executing each instruction is given in the last column.

ica al	<u>clear</u> MRA and <u>add</u> to it N(al)	$N(al) \rightarrow N(MRA)$	0.7ms
ics al	<u>clear</u> MRA and <u>subtract</u> from it N(al)	$-N(al) \rightarrow N(MRA)$	0.7ms
its al	<u>transfer</u> N(MRA) into (al,al+1)	$N(MRA) \rightarrow N(al)$	0.9ms
iad al	<u>add</u>	$N(MRA)+N(al) \rightarrow N(MRA)$	2.0ms
isu al	<u>subtract</u>	$N(MRA)-N(al) \rightarrow N(MRA)$	2.0ms
imr al	<u>multiply</u> and <u>roundoff</u>	$N(MRA) \cdot N(al) \rightarrow N(MRA)$	1.4ms
idv al	<u>divide</u>	$N(MRA) : N(al) \rightarrow N(MRA)$	2.2ms
iox al	<u>exchange</u> N(MRA) with N(al)	$N(MRA) \rightarrow N(al);$ $N(al) \rightarrow N(MRA)$	1.3ms
iTOA+nl.2345c	type out N(MRA) in normalized* form followed by a carriage return		75.ms
iSTOP	STOP		0.4ms

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

$$\pm d_1 \cdot d_2 d_3 d_4 d_5 \times 10^\alpha$$

where α is adjusted so that $d_1 \neq 0$ (unless the number itself is zero).

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 iSTOP.

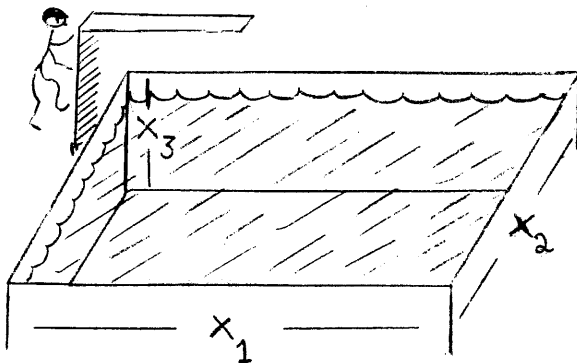
WHEN A NEW QUANTITY REPLACES AN OLD ONE, THE OLD ONE DISAPPEARS!

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

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

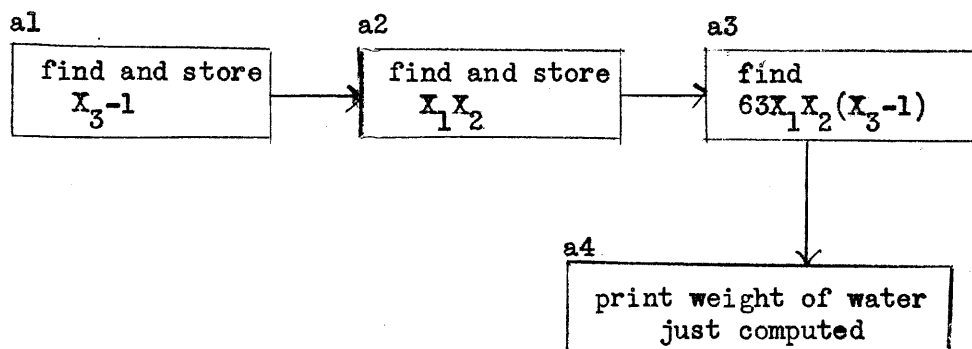
and the amount of paint in gallons is obtained by dividing the above number of square feet by 500, the number of square feet per gallon.

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

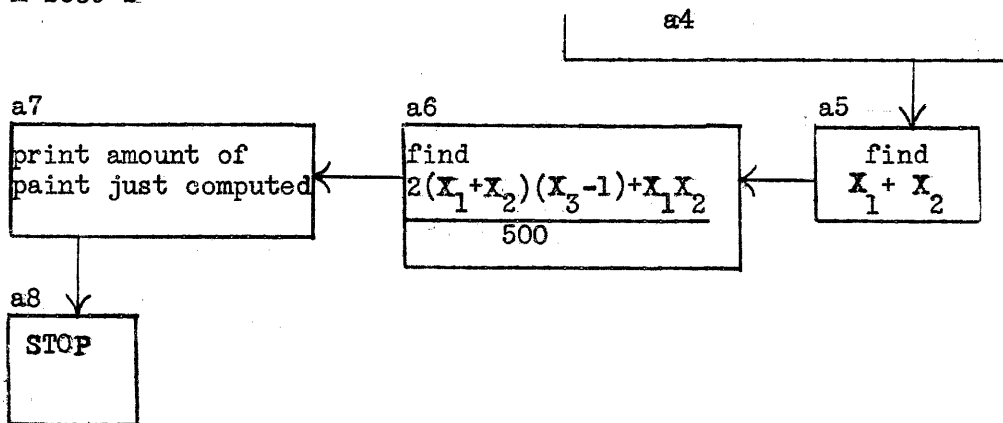
By collecting terms with an eye to reducing the number of multiplications which will be required in the computation, the two quantities desired can be rewritten as follows

$$\text{Paint} = \frac{2(X_1+X_2)(X_3-1) + X_1 X_2}{500} \quad \text{Water} = 63X_1 X_2 (X_3-1)$$

A procedure for finding these quantities might be:



(continued on page I-6)



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.)

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 $N(44) = \text{height}$ 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+n1.2345c	type out the numerical value of $N(\text{MRA})$ which is the desired result	
a5	{	59	ica 40	place X_1 in MRA
		60	iad 42	form $X_1 + X_2$ in MRA
a6	{	61	imr 34	form $2(X_1+X_2)$ in MRA
		62	imr 46	form $2(X_1+X_2)(X_3-1)$ in MRA
		63	iad 48	form " $+X_1X_2$ in MRA
		64	idv 38	form $\frac{2(X_1+X_2)(X_3-1) + X_1X_2}{500}$ in MRA equal to the amount of paint
a7	65	iTOA+n1.2345c	type out the amount of paint	
a8	66	iSTOP	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 pages I-5 and I-6. 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.

CHAPTER 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 prescribed 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 (<u>sub program</u>)	take the next instruction from register al and continue from there	0.5 ms
icp al	conditionally transfer control (<u>conditional program</u>)	ditto, if $N(MRA) < 0^*$; if $N(MRA) > 0$, take the next instruc- tion in sequence	0.4 ms

*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.

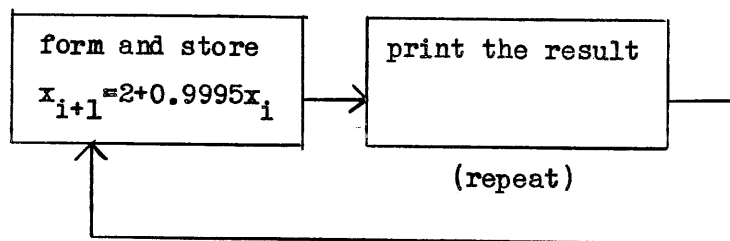
During each minute, 2 pounds of salt come in with the salt water. After 1 minute then, there will be 2 pounds of salt in the water. But during the second minute, not only do 2 more pounds come in, but a small quantity goes down the drain. The amount down the drain is $1/2000$ times the amount present, 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 i^{th} 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

$$x_{i+1} = 2 + x_i - x_i/2000 = 2 + 0.9995x_i \quad \text{during the } (i+1)^{\text{th}} \text{ minute}$$

A possible procedure for programming the problem would be:



The program can be written:*

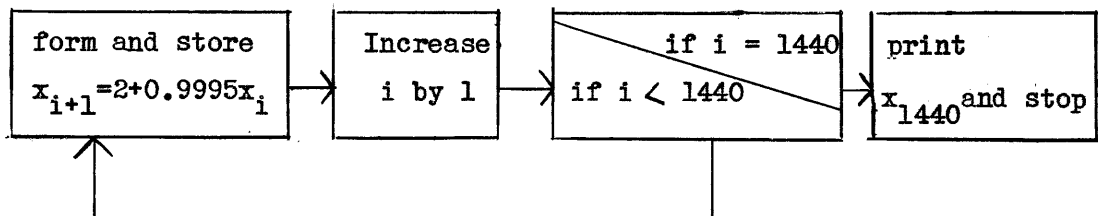
100	+2.	pounds of salt added
102	+0.9995	
104	+0.	amount of salt in pool (initially 0)
106	ica 104	place x_i in MRA
107	imr 102	form $0.9995x_i$
108	iad 100	form x_{i+1} in MRA
109	its 104	replace x_i by x_{i+1} in register 104
110	iTOA+n1.2345c	print x_{i+1}
111	isp 106	repeat; i.e., take next instruction 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:



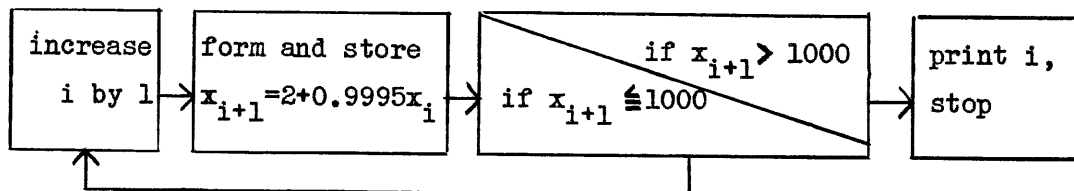
* As an exercise the student may attempt to shorten this program.

100		+1.	} constants
102		+2.	
104		+0.9995	
106		+1439.	
108		+0.	pounds of salt
110		+0.	minutes = i
112		ica 108	
113		imr 104	
114		iad 102	
115		its 108	form x_i in 108
116		ica 110	
117		iad 100	
118		its 110	
119		isu 106	form $i - 1439.$ in MRA
120		icp 112	<u>if</u> $N(MRA) < 0$, which occurs (since i changes in unit steps) if $i < 1440$, take the next instruction from 112 <u>if</u> $N(MRA) > 0$, which occurs when $i = 1440$, ignore this instruction
121		ica 108	} print x_{1440}
122		iTOA+n1.2345c	
123		iSTOP	

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.	}	constants
503	+2.		
505	+0.9995		
507	+1000.		
509	+0.		pounds of salt
511	+0.		minutes = i
513	ica 511	}	increase i by 1
514	iad 501		
515	its 511		
516	ica 509	}	calculate new x_{i+1}
517	imr 505		
518	iad 503		
519	its 509		
520	isu 507	}	If $x_{i+1} \leq 1000$, 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+nl.2345c		
524	iSTOP		

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 $\gg +0$. 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.)

Solutions to examples on page II-6:

- | | | | | | |
|--------|-----------|---------|--------|-----------|--------|
| 1) | ica 32 | | 6) | ica 32 | |
| | iad 34 | | | imr 32 | |
| | its 41 | | | imr 32 | |
| | iSTOP | | | its 115 | |
| 2) | ica 32 | | | imr 115 | |
| | imr 32 | | | imr 115 | |
| | imr 32 | | | its 115 | |
| | its 53 | | | iSTOP | |
| | iSTOP | | 7) 102 | ics 72 | |
| 3) | ica 100 | | | its 72 | |
| | imr 32 | | | ica 72 | |
| | iad 102 | | | iad 113 | |
| | imr 32 | | | its 72 | |
| | iad 104 | | | icp 109 | |
| | imr 32 | | | iSTOP | |
| | iad 106 | | | ica 115 | |
| | its 500 | | | imr 32 | |
| | iSTOP | | | its 115 | |
| 4) 100 | ica 32 | | | isp 104 | |
| | isu 34 | | | +1. | |
| | icp 106 | | | +1. | |
| | ica 32 | | 8a) 65 | ica 77 | 67 |
| | its 75 | | | imr 32 | ica 75 |
| | iSTOP | | | its 77 | iex 77 |
| | ica 34 | | | ica 73 OR | iad 75 |
| | isp 104 | | | iad 75 | icp 72 |
| | | | | its 73 | iSTOP |
| 5) | ica 32 | ica 32 | | icp 65 | iex 77 |
| | imr 32 | imr 32 | | iSTOP | imr 32 |
| | imr 32 OR | its 115 | | -40. | isp 68 |
| | imr 32 | imr 115 | | +1. | +1. |
| | its 115 | its 115 | | +1. | -41. |
| | iSTOP | iSTOP | | +1. | |

8b) 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
		iSTOP				iSTOP

CHAPTER 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, \dots (involving 9 multiplications) using a cycle of instructions. The program might be as follows:

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

64		icp 57
65		iSTOP

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 $C(\dots)$ = contents of ...

i = $C(\text{index register})$

n = $C(\text{criterion register})$

icr m <u>cycle reset</u>	Set $i = +0$, $n = m$ ($+0 \leq m$ is an integer < 2048)	0.4 ms
ict al <u>cycle count</u>	Increase i by 1 and if this new value of $ i \geq n $, then reset $i = +0$ and take the next instruction in sequence; if the new $ i < n $, take the next instruction from register al.	0.4 ms

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|   iSTOP

```

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.

	<u>i</u>	<u>n</u>	<u>MRA</u>
End of cycle	1	1	x^2
	2	2	x^3
	3	3	x^4
	4	4	x^5
	5	5	x^6
	6	6	x^7
	7	7	x^8
	8	8	x^9
	9 $\left\{ \begin{array}{l} 9 \\ \text{reset to} \\ 0 \end{array} \right\}$ 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). 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	iSTOP	

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:

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

ici m	cycle <u>i</u> ncrease	Increase the contents of the index register by m.	0.4 ms
icd m	cycle <u>d</u> ecrease	Decrease the contents of the index register by m.	0.4 ms
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	$\left\{ \begin{array}{l} i=0 \\ n=8 \end{array} \right.$
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:

		<u>i</u>	<u>n</u>
End of cycle	1	2	8
	2	4	8
	3	6	8
	4	$\left\{ \begin{array}{l} 8 \\ \text{reset} \\ \text{to 0} \end{array} \right.$	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 <u>select counter</u>	selects counter j as reference for all subsequent interpreted instructions using a counter <u>until execution of the next isc instruction; each counter has its own index and criterion registers.</u>	0.9 ms
(+0 ≤ j is an integer < 2048)		

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 $t = j \Delta t$ for $j = 1, 2, 3, \dots, 1000$ (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 $i = 5, 10, 15, \dots$), 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

```

* icx defined on page 7

```

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.

106|  isc 1
107|  ict 38
108|  ica 202
109|  iTOA+nl.2345c
110|  isc 0
111|  ict 38
112|  iSTOP
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. However, the first method is the preferred one since it is logically simpler.

icx al	<u>cycle exchange</u>	Exchange C (index register) with C(al) and exchange C (criterion register) with C(al+1).	0.6 ms
--------	-----------------------	--	--------

Second Method

```

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+n1.2345c

106| icx 206
107| ict 35
108| ica 202
109| iTOA+n1.2345c
110| icx 204
111| ict 33
112| iSTOP
200| +0. will contain F
202| +0. will contain G
204| +0. count for 1000 Δt
206| +0. count for G

```

Advanced Section

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>add</u> and <u>transfer</u>	add C(index register) to the C(al) and store the result in the index register and in register al	0.5 ms
iti al	<u>transfer</u> <u>index</u> digits	transfer the right 11 digits of the index register into the right 11 digits of register al	0.4 ms

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

CHAPTER IV: FLOATING ADDRESSES

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+nl.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+nl.2345c

```

and then we add to the program:

```

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

```

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 determining 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 floating addresses (abbreviated as flads) since the actual value of the address

* 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.

(called the absolute address to distinguish it) can not be determined until the program is complete.

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

```

100 | ica il
101 | imr d3
102 | iad b1
103 | its il
104 | iTOA+nl.2345c
105 | isp 100
{
  il, +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. (The letters o and l are excluded.)

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+nl.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 instructions 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 a 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:

```

al,ica xl
    iad cl
    its xl
    imr c2

```

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.,

```

...
...
bl,ica cl
its il
ica il
al,imr c3
...
isp bl+2

```

The instruction `isp bl+2` will transfer control to that register whose address is two more than `bl`. Note we can obtain the same result by writing `isp al-1`. This instruction would transfer control to that register whose address is one less than `al`. 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 $d1, d1+2, d1+4, \dots, d1+398$. 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:

- | | |
|--|--|
| <p>1. (a) icr 8
 ica 32
 al,imr 32
 ict al
 its 115
 iSTOP</p> <p> (b) icr 40
 ica 32
 al,imr 32
 ict al
 its 77
 iSTOP</p> | <p>4. ica 115
 icx 71
 icd 1
 al, ict a2
 its 115
 iSTOP
 a2, imr 32
 isp al</p> <p>115 +1.</p> |
| <p>2. ica c1
 iex c2
 its c1
 iSTOP</p> | |
| <p>3. icr 200
 ica c4
 al, iad d1 + c
 ict al
 its c4
 iSTOP
 c4,+ .0</p> | |

CHAPTER V: INPUT AND OUTPUT

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:

A. Instructions

Instructions are 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 about 1391*) followed by a carriage return or a tab shift. The only exceptions to this rule are the "iSTOP" 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.

B. Numbers

Numbers are typed as a plus or minus SIGN followed by as many as 8 significant digits if desired with a DECIMAL POINT, NO COMMAS, followed by a carriage return or a tab shift. Exponentials with base 2 or 10 may

* See Chapter XIII for detailed discussion of number of registers available.

be appended as factors each preceded by an x (e.g., $+1234.56 \times 2^{-3} \times 10^5$). Numbers may be zero or have any magnitude between about 5.5×10^{-20} and 9×10^{18} .

C. Absolute Address Assignments

Absolute address assignments are 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:

* See Chapter XIII for a discussion of the variation possible in this upper limit.

<u>Address of Register</u>	<u>Contents of Register</u>
32	ica 500
33	imr 502
34	its 36
35	isp 70
36	} +.0
37	
⋮ 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.

D. Floating Address Tags

Floating address tags are 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: cl,+.5000 tags the constant +.5000 so that it can be referred to elsewhere in the program (e.g., imr cl).

The general rules given above in section C. 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

```

al, ica 500
   imr 502
   its il
   isp 70
il, +.0
70| ica 712
   imr il
a2, its 602
   ⋮

```

the programmer would find in storage:

<u>Address</u>	<u>Contents</u>
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-1).

E. 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:

1. The first line should contain suitable identifying information. The first word of this line should be "fc". No other characters should precede the "fc". 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 123-45-6789 John Doe^{*}

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

F. 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.

G. Typographical Errors

If the typist makes a mistake while punching a tape on the Flexo-writer 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.

* The character "↵" is used to indicate a carriage return.

H. 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:

I. Floating Address Assignments

Floating address assignments are 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:

<u>Correct</u>	<u>Incorrect</u>
⋮	⋮
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
cl, +.5	cl, +.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 cl.

J. Ignored and Synonymous Characters

The 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. Backspaces are permitted in titles, but nowhere else. 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.

K. 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 need only be preceded by one tab and followed by none (i.e., followed immediately by an instruction or a number).

L. 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-16), references 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 I. 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, (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, and (3) an oscilloscope ("scope") with camera attachment. 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)	154 characters/line	133 characters/sec.
3) Scope(with camera attachment)	63 characters/line	200-500 characters/sec.

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, T for direct typewriter, or S for scope. 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.

A. 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 (2)

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 (3)

initial zeroes will be printed as spaces.

B. 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 (4)

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

C. 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 nl23.1234
```

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-nl23.1234
```

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

D. 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
```

E. 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.

F. 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).

G. 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:

iTOA+i123.1234 x 2^5 x 10^{-3} ss

H. 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 F. 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.

I. 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.

J. 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 FCR

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 D above).

Thus the request

iTOA+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 iMOAc (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.

V-15

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      +.0342744      +1716.226      n2,+1.0
+               +1.0x10-e
1TOA+1123.1234ss
icap1
1TOA
ica11
1TOA-112345.67c
isps2
i START AT a1
```

AN EXAMPLE OF A PROGRAM TYPED FOR THE CS COMPUTER

M-2539-2

TABLE 1. THE "FL" FLEXOWRITER CODE

Alphanumerical Sequence

Lower Case	Upper Case	Character 123456	Decimal Value	Octal Value	Lower Case	Upper Case	Character 123456	Decimal Value	Octal Value
a	A	000110	6	6	0	0	111110	62	76
b	B	110010	50	62	1	1	010101	21	25
c	C	011100	28	34	2	2	001111	15	17
d	D	010010	18	22	3	3	000111	7	7
e	E	000010	2	2	4	4	001011	11	13
f	F	011010	26	32	5	5	010011	19	23
g	G	110100	52	64	6	6	011011	27	33
h	H	101000	40	50	7	7	010111	23	27
i	I	001100	12	14	8	8	000011	3	3
j	J	010110	22	26	9	9	110110	54	66
k	K	011110	30	36			000101	5	5
l	L	100100	36	44	space bar		001000	8	10
m	M	111000	56	70	=	:	001001	9	11
n	N	011000	24	30	+	/	001101	13	15
o	O	110000	48	60	color change		010000	16	20
p	P	101100	44	54	.)	010001	17	21
q	Q	101110	46	56	,	(011001	25	31
r	R	010100	20	24	-	-	011101	29	35
s	S	001010	10	12	back space		100011	35	43
t	T	100000	32	40	tabulation		100101	37	45
u	U	001110	14	16	carr. return		101001	41	51
v	V	111100	60	74	stop		110001	49	61
w	W	100110	38	46	upper case		111001	57	71
x	X	111010	58	72	lower case		111101	61	75
y	Y	101010	42	52	nullify		111111	63	77
z	Z	100010	34	42					

TABLE 2. THE "FL" FLEXOWRITER CODE

Binary Numerical Sequence

Decimal Value	Octal Value	Character 123456	Lower Case	Upper Case	Decimal Value	Octal Value	Character 123456	Lower Case	Upper Case
0	0	000000	not used		32	40	100000	t	T
1	1	000001	not used		33	41	100001	not used	
2	2	000010	e	E	34	42	100010	z	Z
3	3	000011	8	8	35	43	100011	back space	
4	4	000100	not used		36	44	100100	l	L
5	5	000101			37	45	100101	tabulation	
6	6	000110	a	A	38	46	100110	w	W
7	7	000111	3	3	39	47	100111	not used	
8	10	001000	space	bar	40	50	101000	h	H
9	11	001001	=		41	51	101001	carr. return	
10	12	001010	s	S	42	52	101010	y	Y
11	13	001011	4	4	43	53	101011	not used	
12	14	001100	i	I	44	54	101100	p	P
13	15	001101	+	/	45	55	101101	not used	
14	16	001110	u	U	46	56	101110	q	Q
15	17	001111	2	2	47	57	101111	not used	
16	20	010000	color change		48	60	110000	o	O
17	21	010001	.)	49	61	110001	stop	
18	22	010010	d	D	50	62	110010	b	B
19	23	010011	5	5	51	63	110011	not used	
20	24	010100	r	R	52	64	110100	g	G
21	25	010101	1	1	53	65	110101	not used	
22	26	010110	j	J	54	66	110110	9	9
23	27	010111	7	7	55	67	110111	not used	
24	30	011000	n	N	56	70	111000	m	M
25	31	011001	,	(57	71	111001	upper case	
26	32	011010	f	F	58	72	111010	x	X
27	33	011011	6	6	59	73	111011	not used	
28	34	011100	c	C	60	74	111100	v	V
29	35	011101	-	-	61	75	111101	lower case	
30	36	011110	k	K	62	76	111110	0	0
31	37	011111	not used		63	77	111111	nullify	

CHAPTER VI: ERRORS AND POST-MORTEMS

Even the most carefully written and well organized program is likely to contain mistakes. In fact, the location and correction of these mistakes often constitutes most of the effort required in the development of a working coded program.

Consequently, several facilities have been included in the Comprehensive System to aid in the location of mistakes. The general term "post-mortem" is used to denote any information printed or displayed by the computer expressly for the purpose of locating, or aiding the location of, a mistake.

Several such post-mortem facilities are available. The present chapter will give only an introduction to the use and interpretation of the post-mortem facilities. A more detailed exposition will be given in Chapter XVII.

In the process of reading and converting a Flexowriter program tape (fc tape) a check is made by the conversion program for certain clerical errors which can be recognized before the program is actually performed by the computer. One or more mistakes discovered by this check result in a "conversion" post-mortem. This conversion post-mortem consists of a short phrase giving a brief description of the mistake which is typed on the direct printer (the Flexowriter connected directly to the computer). Examples of two of the most common conversion post-mortems are

```

unassigned flads
    al at 240
and
duplicate flad is b5

```

The first of these results indicates that a word at 240 has referred to a tag al which has not been defined by the program. Every tag referred to by an instruction must be located or assigned somewhere in the program by the notation "al," followed by the information to be placed at this location. If the quantity to be located at al during program read-in is immaterial, nevertheless some quantity, if only +0., should be placed at the location al. E.g., al, +0.

On the other hand, if the notation "al," occurs at two distinct points of the program, the symbol al is ambiguously defined and will result in the second conversion post-mortem mentioned above.

A complete list and explanation of the various possible conversion post-mortems is given in Chapter XIV.

In addition to the conversion post-mortem, the comprehensive system includes the programmed arithmetic post-mortem (PAPM) and a post-mortem tape (fp tape) program.

The PAPM, if requested, gives the programmer a statement of the contents of certain standard registers in the programmed arithmetic subroutines associated with his program. The information recorded by the PAPM can not be controlled by the programmer, but in the majority of cases suffices for the location of the mistake. The fp tape, on the other hand, allows the programmer to obtain information from any drum or core memory register which may be useful in the detection of a mistake.

The PAPM can be requested by a programmer by an appropriate check on his performance request if no fp tape is submitted. An fp tape causes an automatic PAPM if programmed arithmetic subroutines have been called for and used (for more detailed comments see Chapter XVII).

CS PA Post-mortem

A sample CS PAPM is given here:

fc 191-25-62 JONES 0622.1 11-10-55

(24,6) PA PM

stopped at 279 279 | iex493+c 499 | -.12345678 | +7 MRA | +.12345678 | +22
 1633 | 0 | 0,10 1 ||| 3,12 2 | 0,0 3 | 0,0 4 | 0,7 5 | 6,6
 509 | icp606 615 | isp285 320 | isp221 246 | icp255 274 | icp278

The PAPM results (see the above sample) are divided into five sections.

Section 1. Identification. This usually occupies the first two lines and includes the title of the last tape converted plus the date and time on the first line and the number system followed by the letters PAPM on the second line.

Section 2. In the example shown the computer stopped while performing* the instruction in register 279, which was iex493+c. The index of the most recently selected counter was such as to give an effective address

*This section contains information about the interpreted instruction which was being executed or which was most recently executed.

of 499. Register 499 (and 500) contained the gd number $-.123456789 \mid +7$. The contents of the register(s) referred to by the interpreted instruction on which the program stopped are printed out in a manner deemed most useful to the programmer. The details are contained in Chapter XVII.

Occasionally you may get a PAPM showing that the program stopped while executing isp or icp or ict. This can happen only when the computer was stopped manually, and is probably due to the kind of manual stop described in Chapter XIII.

The contents of the MRA as a 9-digit gd number is recorded at the end of the section.

Section 3. The counter section. The contents of the index and criterion registers, respectively, of all the counters called for by the program are printed here. The address at the beginning of the section is the decimal address of the index register of the zeroth counter. The number of the counter most recently used is followed by two extra vertical bars. In the example, counter 1 was most recently used. The index of counter 1 is such as to yield an effective address of 499 in the $iex493+c$ instruction. Counters are printed 10 to a line. If counters are not called for by the program, this section will not appear in the PAPM.

Section 4. The jump table. The PA routine keeps a record of the registers containing the 5 most recent isp or effective icp, i.e., transfer of control or jump, instructions. This is a continuing "delay line" kind of table - entries come in one end and go out the other end 5 jumps later. Transfers due to ict instructions are not entered in the table. When a PAPM is given, the addresses in this table are printed out and each address is followed by a vertical bar and the contents of that register as an interpreted instruction.* The most recent jump appears last in the section, that is, the entries read chronologically from left to right. It should be emphasized that the contents of the registers are

* All automatic output requests become a non-interpreted transfer of control to a routine or routines, automatically assembled in Core Memory immediately preceding the PA routine. All interpretive automatic output (e.g., iMOA---) routines return control interpretatively to the main program by an isp. Such isp's will be as much a part of the jump table as any other interpreted transfer of control. They can usually be identified by a large address preceding the vertical bar.

printed as they appear when the PAPM is given, which may not necessarily agree with the contents when the jump was actually executed. If less than 5 jumps have been executed, only those will be printed. If no jumps have been executed by the interpretive routine, the phrase "no jumps" is recorded.

If in the event of a mistake, information not included in the PAPM is desired, such information can be obtained by means of a post-mortem request tape (fp tape). If an fp tape is used, request for a PAPM is not necessary since a PAPM is provided automatically for a program which includes one or more instructions beginning with the letter i (iad, ict, etc.).

In typing an fp tape it is essential that the first two characters on the punched tape be the letters fp. No other characters may precede or intervene between this pair including characters which may not appear on the typed copy such as delete, carriage return, space, tab, color shifts or black space. The characters fp may be followed by any identifying information desired, provided that no carriage return is typed except at the end of this information where one is required. Following this carriage return requests for the contents of groups of registers recorded either as interpreted instructions or generalized decimal (floating point) numbers may be requested in the following form:

346 ii 741

973 gd 112

The abbreviations ii and gd stand for interpreted instructions and generalized decimal, respectively. The location of the initial register of the group requested is typed before the abbreviation and the location of the final register requested is typed following the abbreviation.

The unit on which the post-mortem information is recorded is determined by the nearest one of the three letter abbreviations DEL, DIR, or SCO preceding the request. These abbreviations cause the post-mortem information to be recorded on the delayed typewriter (via magnetic tape), direct typewriter or film (via the oscilloscope) respectively. Since computer time is used for printing on the direct typewriter, this form of recording should be avoided. In the absence of any specific designation of output unit the delayed printer will be used.

The last request of a post-mortem tape must be followed by two vertical bars (||).

A typical post-mortem tape might resemble the following sample:

fp Ebenezer Aloysius Doe, Esq. - Tuesday

175 ii 175 SCO 176 gd 201

DEL 205 ii 209||

The contents of register 175 would be typed on the delayed printer as an interpreted instruction, the registers from 176 to 201 inclusive would be recorded as 13 generalized decimal numbers on film and the 5 registers from 205 through 209 would be typed from magnetic tape by the delayed printer as interpreted instructions.

Since a request for generalized decimal numbers always involves an even number of registers, these requests must include both an even and odd location. If this is not the case, only the numbers within the specified range will be recorded.

CHAPTER 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 bl           store x
        imr a1           form ax
        iad a2           form ax + b
        imr bl           form ax2 + bx
        iad a3           form ax2 + 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.
bl,   +0. storage for x

```

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

as

```

        isp   pl

a1,    a
a2,    b
a3,    c
bl,    +0.

pl,    its   bl
        imr  a1
        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 subroutines 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 reference register, which is usually the first register of the routine. Relative addresses are indicated by the suffix 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 reference register. If the above example is written in terms of relative addresses, we have:

0r,	isp	9r	The 0r, is used to specify the reference
1r,		a	register, as will be explained in the
3r,		b	following paragraphs. The 1r tags
5r,		c	the instruction (or number) that fol-
7r,		+0.	lows as the register (or pair of regis-
9r,	its	7r	ters) whose absolute address is the
10r,	imr	1r	reference register plus one
11r,	iad	3r	
12r,	imr	7r	
13r,	iad	5r	

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

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

programmer could write

```

100 |Or,      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 } a
102 }
103 } b
104 }
105 } c
106 }
107 } +0.
108 }
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,Or,      isp  9r
            lr, a
            3r, b
            5r, c
            7r, +0.
            its  7r

```

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

(cont. on next page)


```

imr 1r
iad 3r
imr 7r
iad 5r

```

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 become 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 will 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 tape appears the words "END OF SUBROUTINE." These two groups of words are used to indicate on the typewritten sheet the positions of various library subroutines (LSR), which helps make this copy of the program easier to follow.

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 determined by the most recent word which contains a comma in the address-tag section, e.g., "Or," 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
cl,ica d1

```

(continued on next page)

```

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.

<pre>ita al <u>transfer</u> <u>address</u></pre>	<pre>transfer, into the address section of the instruction in register al, the address that is one more than the address of the register con- taining the last <u>isp</u> (or <u>icp</u> with $N(MRA) \leq 0$)</pre>	<pre>0.4ms</pre>
---	---	------------------

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

```
al,ica d1      pick up  $x_1$ 
isp c3         go to subroutine
iTOA+nl.2345t  print the resulting  $N(MRA)$  followed by a tab
ica d2         pick up  $x_2$ 
isp c3         go to subroutine
```

(continued on next page)

```

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.
d1,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 an 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 parameters 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 parameters 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 XIV). 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, ~~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 $a_n x^n + \dots + a_1 x + a_0$ ($11 > n$ (integer) > 0), where the coefficients a_0, \dots, 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=
Or,ita 8r
  its 9r
  icr ppl
  ica llr
4,iad 33r - ppl - ppl + c
  imr 9r
  ict 4r
  iad 33r
8,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 temporary 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-6 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 before reference is made to these registers in the program.

Making use of this new notation, we can rewrite the subroutine on page VII-8 as follows (it is assumed that somewhere in the main program before we use any of the temporary registers, t will have been specified):

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

Thus, by referring to t (and lt), 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 lt. 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 (not 0). Thus lt refers to a temporary storage register whereas t1 is a floating address.

A. Summary of the Instruction Code of the Simplified CS Computer
(see definition of symbols in Table I)

<u>Instr.</u> *	<u>Op. Time</u> <u>in msec.</u>	<u>Meaning</u>	<u>Definition</u>	<u>Alarm</u> ***
its al+c	1.0	(cycle) <u>transfer</u> N(MRA) into (al+2i, al+2i+1)	$N(MRA) \rightarrow N(al+2i)$	B, F
iex al+c	1.4	(cycle) <u>exchange</u>	$N(MRA) \leftrightarrow N(al+2i)$	B, F
ica al+c	.79	(cycle) <u>clear</u> MRA; <u>add</u> N(al+2i)	$N(al+2i) \rightarrow N(MRA)$	F
ics al+c	.81	(cycle) <u>clear</u> MRA; <u>subtract</u> N(al+2i)	$-N(al+2i) \rightarrow N(MRA)$	F
iad al+c	2.2	(cycle) <u>add</u>	$N(MRA) + N(al+2i) \rightarrow N(MRA)$	C', F
isu al+c	2.2	(cycle) <u>subtract</u>	$N(MRA) - N(al+2i) \rightarrow N(MRA)$	C', F
imr al+c	1.6	(cycle) <u>multiply</u> and <u>roundoff</u>	$N(MRA) \times N(al+2i) \rightarrow N(MRA)$	C', F, K
idv al+c	2.3	(cycle) <u>divide</u>	$N(MRA) \div N(al+2i) \rightarrow N(MRA)$	C', E', F, K
isp al+c	.67	(cycle) <u>transfer of control</u>	Take the next instruction from reg. (al+i) and continue from there	D, F
isc j **	.90	<u>select</u> <u>counter</u>	Select cycle count line j	A
icr m **	.40	<u>cycle</u> <u>reset</u>	Set i=0, n=m	
ict al	.45	<u>cycle</u> <u>count</u>	Increase i by 1; if $ i_k \geq n $, reset i=0 and take next instruction in sequence; if $ i_k < n $ take next instr. from register al	D, J
iat al	.45	<u>add</u> and <u>transfer</u>	Add C(index reg.) to the C(al) and store the result in index reg. and register al	I
iti al	.40	<u>transfer</u> <u>index</u> <u>digits</u>	Transfer the right 11 digits of the I index reg. into the right 11 digits of register al	
ici m **	.40	<u>cycle</u> <u>increase</u>	Increase contents of index reg. by m	G
icd m **	.40	<u>cycle</u> <u>decrease</u>	Decrease contents of index reg. by m	H
icx al	.60	<u>cycle</u> <u>exchange</u>	Exchange C(index reg.) with C(al) and exchange C(criterion reg.) with C(al+1)	
ita al	.38	<u>transfer</u> <u>address</u>	Replace the address section of the instr. in reg. al with the address that is one more than the address of the reg. containing the last <u>isp</u> (or <u>icp</u> with N(MRA) negative)	
icp al	.38	<u>conditionally</u> <u>transfer</u> <u>control</u> (<u>conditional</u> <u>program</u>)	Take the next inst. from reg. al and continue from there, if N(MRA) is neg.; if N(MRA) is pos. take the next instruction in sequence.	
its al	0.9	<u>transfer</u> N(MRA) into (al, al+1)	$N(MRA) \rightarrow N(al)$	B
iex al	1.3	<u>exchange</u> N(MRA) with N(al)	$N(MRA) \leftrightarrow N(al)$	B
ica al	.65	<u>clear</u> MRA; <u>add</u> N(al)	$N(al) \rightarrow N(MRA)$	
ics al	.68	<u>clear</u> MRA; <u>subtract</u> N(al)	$-N(al) \rightarrow N(MRA)$	
iad al	2.0	<u>add</u>	$N(MRA) + N(al) \rightarrow N(MRA)$	C
isu al	2.0	<u>subtract</u>	$N(MRA) - N(al) \rightarrow N(MRA)$	C
imr al	1.4	<u>multiply</u> and <u>roundoff</u>	$N(MRA) \times N(al) \rightarrow N(MRA)$	C, K
idv al	2.2	<u>divide</u>	$N(MRA) \div N(al) \rightarrow N(MRA)$	C, E, K
isp al	.46	<u>transfer</u> <u>control</u> (<u>subprogram</u>)	Take next instr. from reg. al and continue from there	D

* For Output Instructions see Table III.

** m and j are positive integers less than 2,048.

*** Consult Alarm Table in this chapter.

Table I - DEFINITION OF SYMBOLS

<u>Symbol</u>	<u>Meaning</u>
MRA	Multiple register accumulator
al	Let al represent any floating address, absolute address
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
C(...)	Contents of ...
i	C(index register)
i_k	New C(index register)
n	C(criterion register)
N(al+2i)	The number stored in registers al+2i and al+2i+1 before the instruction is obeyed
→	Replaces
clear ...	Set the contents of... to zero

Table II - ALARMS

(C', D' are same as C,D except that al+c replaces al)

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 so that it has become greater than the largest j in the isc j instructions before the program was performed).
- (B) Exponent of N(MRA) ≥ 64
- (C) $0 < |C(al)| < 1/2$
- (D) When control is transferred to an undefined instruction an alarm occurs on the undefined instruction.

Divide Error Alarm

- (E) $C(al) = 0$

Arithmetic Overflow Alarms

- (F) The contents of the index register could be large enough to cause an alarm; i.e., when $al+c > 32,767$.
- (G) $C(\text{index register}) + m > 32,767$
- (H) $C(\text{index register}) - m < -32,767$
- (I) $|C(\text{index register}) + C(al)| > 32,767$
- (J) $i = 32,767$ before the ict is executed
- (K) $|Result| > 7.0 \times 10^{9863}$ or $|Result| < 7.1 \times 10^{-9864}$

Table III - OUTPUT INSTRUCTIONS

- A. Specifications using either iTOA, iMOA or iSCA
 iTOA abcdefg Record N(MRA) on direct printer
 iMOA abcdefg Record N(MRA) on delayed printer
 iSCA abcdefg Record N(MRA) on scope (film)

Example: iTOA $\overset{+}{\underbrace{\quad}_a} \overset{i}{\underbrace{\quad}_b} \overset{1}{\underbrace{\quad}_c} \overset{2}{\underbrace{\quad}_d} \overset{3}{\underbrace{\quad}_e} \overset{4}{\underbrace{\quad}_f} \overset{5}{\underbrace{\quad}_g} \times 2^{-3} \times 10^2$

- | | | |
|-----|----------------------|--|
| (1) | <u>Sign</u> | <u>Meaning</u> |
| | + | all numbers will be preceded by sign |
| | - | only negative numbers will be preceded by sign |
| | nothing | numbers will not be preceded by sign |
| (2) | <u>Initial Zeros</u> | <u>Meaning</u> |
| | i | initial zeros will be skipped |
| | p | initial zeros will be replaced by spaces |
| | n | all numbers will be printed in a normalized form |
| | nothing | initial zeros will be printed as zeros |
| (3) | <u>Digits Left</u> | |

The programmer indicates the number of digits he wishes to have printed to the left of the decimal point by actually writing a sample number containing the same number of digits to the left of the decimal point as he wishes printed. Thus:

iTOA + 123.4567

specifies that the programmer wishes to have his numbers printed with 3 digits to the left of the decimal point. (The magnitude of the digits one writes in the sample number has no effect whatsoever on the program.) If the number actually contains more than three digits to the left of the decimal point, all these digits will automatically be printed out.

- | | | |
|-----|----------------------|---|
| (4) | <u>Decimal Point</u> | <u>Meaning</u> |
| | . | A decimal point will appear in all numbers |
| | nothing | No decimal point will be printed. (Used when programmer desires only integral part of number.) |
| | r | No decimal point will be printed. (Used when programmer expects all results to be less than one; if the number in the MRA should unexpectedly exceed unity, then the integral |

part of the number will also be printed out but no decimal point will separate the integral from the fractional part of the number.

(5) Digits Right

If a programmer were to use the sample number illustrated in (3), he would get four digits printed out to the right of the decimal point (or to the right of where the decimal point should be).

(6) Scale Factors

Powers of 2 and 10 may be used as scale factors to multiply the number in the MRA before it is printed out:

(a) Every factor must be preceded by a lower case x.

(b) $|\alpha|, |\beta| \leq 99$

(7) Terminal Characters (characters used to terminate a number)

s	space
ss	2 spaces
sss	3 spaces
ssss	4 spaces
c	carriage return
t	tab
nothing	carriage of typewriter will remain exactly where it was after the last number was typed
f	format (see section C below)

Examples:

(1)	-7.953261	iTOA + 123.1234s	-007.9532 space	} <u>DIRECT</u>
		iTOA + i123.1234	-7.9532	
		iTOA pl23.1234c	7.9532 car.ret.	
		iTOA - n123.1234t	-795.3261 -02 tab	
(2)	+795.3261	iMOA 123.123ss	795.326 space space	} <u>DELAYED</u>
		iMOA - i1234.5x10 ² c	79532.6 car.ret.	
		iMOA + pl2r34	+79532	
		iMOA n1.234t	7.953 +02 tab	

B. Special Characters

To print out a single special character (such as a decimal point, space, tab, sign, or carriage return) the programmer follows the iTOA or iMOA by the single symbol representing the desired character (as indicated in sections (1), (4) and (7) of table III above).

Example:

iTOAc

will cause a carriage return to be typed on the "direct" typewriter.

C. Format Specification

This facility provides the programmer with an automatic device for obtaining a suitable layout of his output data.

If "f" is used as a terminal character in an output request (see Table III, section A-7) then the instruction and 3 program parameters

iFOR

α

β

γ

must appear somewhere in the program before the first output instruction containing the "f". (This will furnish the CS output section with the necessary layout information before a number is printed out.)

- α represents the number of words/line (maximum number of characters per line is 155)
- β represents the number of spaces between words (A tab is obtained by setting $\beta=0$)
- γ represents the number of words per block (The maximum is 32,767. Since the block counter is automatically reset after each block is completed, the upper limit, 32,767, for is not a significant limitation.)

Example: Supposing the programmer wishes to have 2500 words typed out and uses:

iFOR
+12
+2
+400

The output request iTOA+12.345f will then give 12 words per line, 2 spaces between words and 400 words per block. THE BLOCKS ARE SEPARATED BY 2 CARRIAGE RETURNS. In this example there will be six blocks of 400 words and one block of 100 words. (The programmer should provide carriage returns at the beginning and at the end of his print-out if the latter doesn't coincide with the end of a block.)

B. Cautions

1. It is important that programmers write a vertical bar (e.g., 34 |) long enough so that it cannot be confused with the numeral one.

2. The initial word following the tape title (excluding such special words 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 to be 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 Chapter XIV.

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

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

+2.7
+2.7 x 10⁶
+0.102659
+. or +.0 or +0.

INCORRECT NUMBERS

2.7
+2.7 x 10⁺⁶
0.102659
+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.

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

Example 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,

a1,
a2,
a3,
.
.
.

is preferred to

a1,
a1+1,
a1+2,
.
.
..

This follows from the fact that a_1, a_2, a_3, \dots are independent 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 a_1 and a_1+1 , and "its b_2 " transfers $N(MRA)$ to b_2 and b_2+1 . Suppose we desired to transfer the numbers in c_1 and c_2 into a sequence of registers beginning at b_2 :

<u>CORRECT</u>	<u>INCORRECT</u>
ica c_1	ica c_1
its b_2	its b_2
ica c_2	ica c_2
its b_2+2	its b_2+1

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

al, ica t_1	(floating address)
its a_1+7	(floating address)
its $2t$	(temporary storage address)
its t_3	(floating address)
idv t_1+2	(floating address)
its $9r$	(relative address)
isp r_3+2	(floating address)

11. If a floating address tag, such as e_1 , is preceded by an address assignment (disregarding carriage returns and tabs), then this must be either an absolute address or a relative address assignment.

<u>A. CORRECT</u>	<u>B. CORRECT</u>	<u>C. INCORRECT</u>
134 $d_1, +.0$	$d_1, +.0$	$d_1, +.0$
+.0	+.0	+.0
140	6r	$d_1+6 $
$e_1, +.4$	$e_1, +.4$	$e_1, +.4$

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 illustrated as follows:

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 dl	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, Page VI-1).

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 ics) or the cycle counter letter "c" appears in the original program.

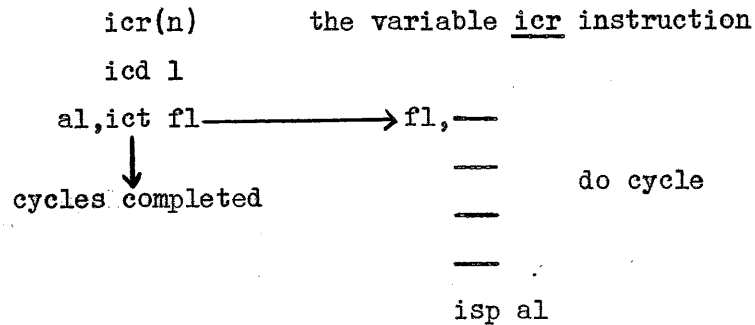
Cycle counter line zero is the first counter line available. The instruction isc 1 selects the second counter line, isc 2 selects the third counter line, etc. However, from a programmer's point of view it may be easier to think of it in the following way:

```

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:



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 Chapter VII, Page VII-9)

<u>ACCEPTABLE</u>	<u>UNACCEPTABLE</u>
<u>VALUES FOR PRESET PAR.</u>	<u>VALUES FOR PRESET PAR.</u>
+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:

<u>CORRECT</u>	<u>INCORRECT</u>
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:

<u>CORRECT</u>	<u>INCORRECT</u>
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 refers to the eighth register after bl
its al+5	al+5 refers to the fifth register after al
imr al+7	al+7 refers to the seventh register after bl
its l0r	l0r refers to the tenth register after 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:

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

M-2539-2

COMPREHENSIVE SYSTEM MANUAL

PART II

Advanced Coding Techniques

CHAPTER IX: SOME MORE FUNDAMENTALS

In the preceding chapters, the basic features of the CS computer have been presented. We have seen how the computer can be applied to solve the type of problems which arise in scientific calculations. Although the problems presented were simple, they served to illustrate the characteristics of the computer: its ability to perform arithmetic operations, to make decisions based on calculated results, to modify the effective address of its instructions during cyclic operations. We have described, also, such concepts as mnemonic coding, floating addresses, preset parameters, and decimal input. These are not primarily attributes of the computer itself, but rather are features of the conversion routine which has been written for it. The conversion routine is the program which reads Flexowriter tapes and translates them into the numbers and instructions which are to be stored in the computer memory. Together, the computer and its conversion routine constitute a powerful instrument for carrying out scientific and engineering calculations.

A few words about the motivation and the design of the CS computer are appropriate at this point. This computer is designed specifically to handle the numbers commonly used by scientists and engineers in their calculations. It is at once apparent that these numbers may with equal likelihood be exceedingly small or exceedingly large, for the scientist is concerned with figures which may vary over a tremendous range of magnitudes. In a digital computer, numbers are represented as a count of how many times some basic increment is contained in the number. In order to have an accurate representation for small numbers, the basic increment must be made small. A small increment size, however, means that the count of increments contained in a large number will be large. It would appear, therefore, that the CS computer should have both a small increment size (to distinguish between small numbers) and a large register length (to permit large numbers to be stored).

If the computer were designed in this way, we would indeed be able to use it to solve scientific problems. We would soon notice, however, that at any time only a small portion of each register actually contained useful and significant information. Scientific data almost always are expressed to a fixed number of significant digits, the remaining digits being uncertain. Our storage registers, because they were required to accommodate both very large and very small numbers, would necessarily contain many more digit positions than we could effectively use at one time. Those digit positions for which we had no information would be set to zero and would be wasted. This inefficient use of storage capacity could be avoided if, in some way, we could devise a scheme for storing only the significant digits of our numbers. But this, of course, is quite simple. All it would require is that we associate with each number an indication of the position of the decimal point relative to the digits; the two pieces of information, taken together, would then completely specify the number.

The CS computer actually employs a scheme of this sort, which is called floating-point representation, for its numbers. Each number consists of two distinct parts. One of these contains the significant digits of the number, and the other indicates the position of the point. The conversion routine automatically translates numbers into this form during read-in, and the computer itself automatically performs arithmetic operations on numbers which are in this form. Since it permits handling of numbers with widely differing magnitudes and retains only significant digits, the floating-point arithmetic facility is by far the most useful feature of the CS computer.

It will be recalled that instructions and numbers in the CS computer are essentially different in form. Because the floating-point representation is applicable to numbers but not to instructions, the reason for this distinction is now evident. It is, however, very frequently useful to modify instructions during the course of a program, and it would be desirable to have some means for operating on instructions as well as on numbers. It would then be possible, for instance, for a

program to compute an address (which is, after all, nothing but a special kind of number) and insert it into an instruction to be obeyed at a later point in the program. In the CS computer, the difference between numbers and instructions makes this almost impossible.* What we need is another set of instructions, to supplement those of the CS computer, which operate on other instructions rather than on floating-point numbers.

This set of instructions is found, not in the CS computer, but in another digital computer, called Whirlwind I. In order to write effective and efficient programs for the CS computer, it is necessary to be familiar also with Whirlwind I. The two computers complement each other, each being most efficient for the type of computation for which it was designed. The relationship between the two computers is a curious one, for both may be used during the course of the same problem. It is possible to switch from CS operation to Whirlwind I operation or vice versa, whenever it is desired to do so.** The reason for this has been hinted at in an earlier chapter. The CS computer is actually simulated by Whirlwind I; it does not exist as a collection of circuits and components. The simulation is carried out by interpretive routines, in the Whirlwind I instruction code, which are automatically executed each time a CS instruction is to be obeyed. The routines needed to carry out any CS instructions that appear in a tape are stored in the higher-numbered registers of core memory by the conversion routine. The same conversion routine is used to translate both CS tapes and Whirlwind I tapes. Therefore, all the facilities of the conversion routine (mnemonic coding, floating addresses, preset parameters, etc.) are available in the Whirlwind I computer, just as they are in the CS computer. This makes the facilities of both computers equally available to the coder.

*The CS computer instructions iti al and iat al permit such computations to be carried out, but are awkward to use.

**The special instructions IN and OUT, which are required to do this, are described in a later chapter.

A distinction must, of course, be drawn between the conversion routine and the interpretive routines which comprise the CS computer. The conversion routine simply translates from Flexowriter code to the binary code of Whirlwind I, assembling in storage the program it has read from paper tape plus any additional routines required by that program. The conversion routine operates during read-in, before the program itself starts to operate. The interpretive routines, on the other hand, actually appear in core memory during the operation of a program, and are used automatically each time a CS instruction is executed. The interpretive routines are compiled by the conversion routine, for later use during the running of the program being converted.

In summary, then, we can say that the CS computer provides features, not found in Whirlwind I itself, which simplify the task of coding for scientific and engineering calculations. Its usefulness is limited to these special features, however, and judicious use of the Whirlwind I computer in conjunction with the simulated computer can result in both greater ease of coding and in reduced operating time. To make most effective use of the facilities of the Digital Computer Laboratory, one should be familiar with the advantages (and the disadvantages) of both computers.

Because Whirlwind I is a binary computer, some understanding of non-decimal number systems is desirable before discussing it. Such number systems are the subject of the next chapter. Coding for Whirlwind I itself is described in Chapter XI.

The more advanced section of this manual follows Chapter XII. The basic principles involved in programming and coding are covered in Chapters X through XII and are quite adequate for effective use of the computer by most coders. The advanced chapters serve primarily to extend the principles developed in Part I to cases which, for clarity of presentation, were not treated earlier. All the features of the Comprehensive System of Service Routines are described in sufficient detail to enable the more proficient coder to utilize the full flexibility of the

system.

In addition, the advanced section contains a description of some of the methods by which the features of the Comprehensive System were realized. The Comprehensive System includes not only the compiling and assembly routines of the conversion program, but also the interpretive-arithmetic and automatic-output routines of the CS computer and the generalized post-mortem routines. Hence, this description provides the advanced coder with background material for a fuller understanding of the principles and techniques involved in such a compiling system.

CHAPTER X: NUMBER SYSTEMS

I. Introduction

The accident of ten fingers led man quite naturally to use decimal numbers as a means for computation. The preferred position of the decimal system remained undisturbed until the advent of the high speed electronic digital computer, when it was discovered that another system (based on 2 instead of 10) was more suitable.

In this chapter we show that any integer greater than 2 can be used as a base for a number system and discuss the problems of **converting between number systems.**

II. The Positional Notation

Let N be an integer and $R \geq 2$ be an integer. If $|N|$ is divided by R there results a unique quotient, q_0 , and a unique remainder, r_0

$$|N| = q_0 R + r_0 \quad q_0 \geq 0 \quad 0 \leq r_0 < R$$

Since q_0 is an integer the process can be repeated with q_0 as dividend.

$$q_0 = q_1 R + r_1 \quad q_1 \geq 0 \quad 0 \leq r_1 < R$$

The process terminates when $q_{i-1} < R$ whence we obtain

$$q_i = 0 \quad r_i = q_{i-1}$$

N has thus been expressed in the form

$$N = \pm (r_i R^i + \dots + r_1 R + r_0)$$

The above expression can be shortened to

$$N = \pm r_i r_{i-1} \dots r_1 r_0$$

which is called a positional representative of N

R is called the radix of the representation and the integers, r_j , are called the digits of the representation. R distinct characters are required to express integers in positional notation with radix R .

If F is a fraction, $0 \leq |F| < 1$, the positional representative of F is obtained as follows:

Multiplying $|F|$ by R we obtain

$$R|F| = r_{-1} + F_{-1}$$

where

$$0 \leq r_{-1} < R \qquad 0 \leq F_{-1} < 1$$

Similarly multiplying F_{-1} by R we obtain

$$RF_{-1} = r_{-2} + F_{-2} \qquad 0 \leq r_{-2} < R \qquad 0 \leq F_{-2} < 1$$

and

$$RF_{-2} = r_{-3} + F_{-3} \qquad 0 \leq r_{-3} < R \qquad 0 \leq F_{-3} < 1$$

$$\dots\dots\dots \qquad \dots \qquad \dots$$

$$RF_{-(n-1)} = r_{-n} + F_{-n} \qquad 0 \leq r_{-n} < R \qquad 0 \leq F_{-n} < 1$$

$$\dots\dots\dots \qquad \dots \qquad \dots$$

F has thus been expressed in the form

$$F = + (r_{-1} R^{-1} + r_{-2} R^{-2} + \dots + r_{-n} R^{-n} + \dots)$$

which is shortened to

$$F = + \cdot r_{-1} r_{-2} \dots r_{-n} \dots$$

If $F_{-n} = 0$ for some value of n , then

$$r_{-(n+1)} = r_{-(n+2)} = \dots = 0$$

and the positional representative is said to terminate.

For example, in the decimal system

$$1/8 = + .12500\dots$$

has a terminating representative.

III. Radix Systems

Any integer, $R \geq 2$, can be selected as radix for a positional notation. The following names have been given to systems which arise when radices other than 10 are chosen

- 2 Binary
- 3 Ternary
- 4 Quaternary
- 5 Quinary
- 6 Senary
- 7 Septenary
- 8 Octal

- 9 Novenary
- 10 **DECIMAL**
- 11 Undecimal
- 12 Duodecimal

It was noted previously that R distinct characters are required for the digits in a positional notation with radix, R. For $R \leq 10$ it is most convenient to employ the usual symbols, 0, 1, ..., 9. For $R > 10$ new digit symbols must be introduced, e.g. we could choose letters and let a represent the "digit" 10, b represent the "digit" 11, etc.

Some examples follow:

$$\begin{array}{ll}
 101 = 1 \cdot 2^2 + 0 \cdot 2 + 1 & \text{Radix 2} \\
 2221 = 2 \cdot 3^3 + 2 \cdot 3^2 + 2 \cdot 3 + 1 & \text{Radix 3} \\
 61 = 6 \cdot 7 + 1 & \text{Radix 7} \\
 aa75 = 10 \cdot 11^3 + 10 \cdot 11^2 + 7 \cdot 11 + 5 & \text{Radix 11}
 \end{array}$$

IV. Arithmetic within a Radix System

Arithmetic within a radix system is based upon two tables (addition and multiplication tables) and a set of rules of combination. The rules of combination are exactly those learned in high school for dealing with decimal numbers. The tables are a function of the system. The addition and multiplication tables for the binary (radix 2) and ternary (radix 3) systems are given below.

Binary

Addition

$$\begin{array}{r|l}
 & 0 \quad 1 \\
 0 & 0 \quad 1 \\
 1 & 1 \quad 10
 \end{array}$$

Multiplication

$$\begin{array}{r|l}
 & 0 \quad 1 \\
 0 & 0 \quad 0 \\
 1 & 0 \quad 1
 \end{array}$$

Ternary

Addition

$$\begin{array}{r|l}
 & 0 \quad 1 \quad 2 \\
 0 & 0 \quad 1 \quad 2 \\
 1 & 1 \quad 2 \quad 10 \\
 2 & 2 \quad 10 \quad 11
 \end{array}$$

Multiplication

$$\begin{array}{r|l}
 & 0 \quad 1 \quad 2 \\
 0 & 0 \quad 0 \quad 0 \\
 1 & 0 \quad 1 \quad 2 \\
 2 & 0 \quad 2 \quad 11
 \end{array}$$

Arithmetic in any system other than decimal is an unrewarding experience.

V. Conversion between Radix Systems

We consider the following problem: given a number, x , to find a positional representative of x with radix, R .

Since x can be expressed in the form

$$x = N + F$$

where N is an integer and F is a fraction this problem can be split into two parts.

(1) Given an integer, N , to find a positional representative of N with radix, R , and

(2) Given a fraction, F , to find a positional representative of F with radix, R .

VI. Integer Conversion between Radix Systems

(1) Let N be an integer which is to be converted to a positional representative with radix, R . Such a representative exists.

$$N = r_m R^m + \dots + r_1 R + r_0$$

Dividing by R , we have

$$\frac{N}{R} = r_m R^{m-1} + \dots + r_1 + \frac{r_0}{R}$$

Thus the remainder after division is the digit, r_0 , of the required representative and the quotient is

$$N_1 = r_m R^{m-1} + \dots + r_2 R + r_1$$

If N_1 is divided by R , the new remainder is the digit, r_1 , of the required representative. The process is repeated until all the digits have been obtained.

The arithmetic of the algorithm (forming N/R , N_1/R , ...) is performed in the radix system to which N was originally expressed. Thus the algorithm is of most value when arithmetic in the original radix system is relatively easy.

As an illustration of the algorithm we convert the decimal integer, 612, to a positional representative with radix 7.

$$\begin{array}{rcl}
 612/7 & = & 87 + 3/7 & 3 \\
 87/7 & = & 12 + 3/7 & 3 \\
 12/7 & = & 1 + 5/7 & 5 \\
 1/7 & = & 0 + 1/7 & 1
 \end{array}$$

The required representative is thus, 1533.

(2) If arithmetic in the final radix system is relatively easy the following algorithm may be used.

Let N be an integer which is initially expressed in positional notation with radix, S , and which is to be converted to a positional representative with radix, R . Then

$$N = s_m S^m + \dots + s_1 S + s_0$$

and

$$N = (\dots((s_m S + s_{m-1}) S + s_{m-2}) S + \dots)$$

The latter expression can be directly evaluated using the arithmetic of the final radix system.

As an illustration of the algorithm we convert the ternary (radix 3) integer, 21211, to a decimal integer.

$$\begin{array}{l}
 2 \cdot 3 + 1 = 7 \\
 7 \cdot 3 + 2 = 23 \\
 23 \cdot 3 + 1 = 70 \\
 70 \cdot 3 + 1 = 211
 \end{array}$$

The required representative is thus 211.

A simplification of the algorithm occurs if there is available a table of powers of the initial radix expressed in positional notation with radix R . In this case the expression

$$N = s_m S^m + \dots + s_1 S + s_0$$

may be directly evaluated using the arithmetic of the final radix system.

Thus given the table

$$\begin{array}{l}
 3^4 = 81 \\
 3^3 = 27 \\
 3^2 = 9 \\
 3^1 = 3 \\
 3^0 = 1
 \end{array}$$

the decimal equivalent of the ternary integer, 21211, can be evaluated as follows

$$2 \cdot 81 + 1 \cdot 27 + 2 \cdot 9 + 1 \cdot 3 + 1 = 162 + 27 + 18 + 3 + 1 = 211$$

VI Fraction Conversion between Radix Systems

(1) Let F be a fraction which is to be converted to positional notation with radix R . Such a representation exists.

$$N = r_{-1} R^{-1} + r_{-2} R^{-2} + \dots$$

Multiplying by R , we have

$$NR = r_{-1} + r_{-2} R^{-1} + r_{-3} R^{-2} + \dots$$

Thus the integer part of NR is the digit, r_{-1} , of the required representative and the fractional part of NR is

$$N_{-1} = r_{-2} R^{-1} + r_{-3} R^{-2} + \dots$$

If the product, $N_{-1}R$, is formed, then the integer part of $N_{-1}R$ is the digit, r_{-2} , of the required representative. The process can be repeated until it terminates (some $N_{-i} = 0$) or until a sufficient number of digits have been obtained.

The algorithm requires that arithmetic (in forming NR , $N_{-1}R$, ...) be performed in the original radix system and is most convenient when arithmetic in that system is relatively easy.

As an illustration of the algorithm we convert the decimal fraction, $+.125$, to a binary fraction.

$$\begin{array}{r} + .125 \cdot 2 = + 0.250 \qquad 0 \\ + .250 \cdot 2 = + 0.500 \qquad 0 \\ + .500 \cdot 2 = + 1.000 \qquad 1 \end{array}$$

The required binary number is thus $+.001$.

(2) If arithmetic in the final radix system is relatively easy the following algorithm may be used.

Let F be a fraction which is expressed in finite positional notation with radix, s , and which is to be converted to an equivalent positional representative with radix, R . Then

$$F = s_{-1} s^{-1} + s_{-2} s^{-2} + \dots + s_{-m} s^{-m}$$

and

$$F = \left(\dots \left(\left(s_{-1} \cdot \frac{1}{s} + s_{-2} \right) \frac{1}{s} + s_{-3} \right) \frac{1}{s} + \dots \right) \frac{1}{s}$$

The latter expression can be directly evaluated using the arithmetic of the final radix system.

As an illustration of the algorithm we convert the binary fraction, $+.101_2$, to an equivalent decimal fraction.

$$\begin{aligned} 1/2 + 0 &= 0.5 + 0 = +0.5 \\ +0.5/2 + 1 &= +0.25 + 1 = +1.25 \\ +1.25/2 &= +0.625 \end{aligned}$$

The required decimal fraction is thus $+0.625_{10}$.

A simplification of the algorithm occurs if there is available a table of powers of the initial radix expressed in positional notation with radix R . In this case the expression

$$N = s_{-1} R^{-1} + s_{-2} R^{-2} + \dots + s_{-m} R^{-m}$$

may be directly evaluated using the arithmetic of the final radix system.

Thus, given the table

$$\begin{aligned} 2^{-1} &= +0.5 \\ 2^{-2} &= +0.25 \\ 2^{-3} &= +0.125 \end{aligned}$$

The decimal equivalent of the binary fraction, $+.101_2$, is evaluated as follows

$$1 \circ 0.5 + 0 \circ 0.25 + 1 \circ 0.125 = +0.5 + 0.125 = +0.625$$

VII. Binary to Octal Conversion

Let N be an integer expressed in positional notation with radix, R .

$$N = r_0 + r_1 R + r_2 R^2 + \dots$$

and let $S = R^j$, where j is a positive integer. By suitably grouping terms in the above expression we obtain

$$\begin{aligned} N &= (r_0 + r_1 R + \dots + r_{j-1} R^{j-1}) \\ &+ (r_j + r_{j+1} R + \dots + r_{2j-1} R^{j-1}) R^j \\ &+ \dots \\ &+ (r_{ij} + r_{ij+1} R + \dots + r_{ij+j-1} R^{j-1}) R^{ij} \\ &+ \dots \end{aligned}$$

Letting

$$s_j = r_{ij} + r_{ij+1} R + \dots + r_{ij+j-1} R^{j-1}$$

this becomes

$$N = s_0 + s_1 R^1 + \dots + s_j R^{1j} + \dots$$

or, since $R^1 = R^j$

$$N = s_0 + s_1 S + \dots + s_j S^{1j}$$

Thus conversion from positional notation with radix, R , to positional notation with radix, $S = R^j$, can be done by inspection if there is available a table giving the digits of the final radix system in positional notation with radix, R .

The required table for binary to octal conversion (and conversely) is the following:

Binary	Octal
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

As an example we convert to octal the binary number, 10110110

$$\begin{array}{ccc} \underline{10} & \underline{110} & \underline{110} \\ 2 & 6 & 6 \end{array} \quad 266$$

The binary radix system is important because of its frequent use in electronic digital computers. The octal radix system has a reflected importance since arithmetic in octal is easier than arithmetic in binary (fewer digits, hence fewer chances for error).

IX. Exponential Representations

Let X be a positive (non-zero) real number. There exists a real number, Z , such that

$$X = 2^Z$$

namely

$$Z = \log_2 X$$

\mathbb{Z} can be expressed in the form

$$(9.1) \quad \mathbb{Z} = y + i$$

where i is an integer. Thus, $X = 2^{y+i} = 2^y \cdot 2^i$ and letting

$$x = 2^y$$

we have

$$X = x \cdot 2^i$$

We abbreviate the above notation to

$$X = (x, i)$$

and call the couple, (x, i) , an exponential representative of X .

X is called the mantissa of the representative and y is called its exponent.

We can perform arithmetic operations with exponential representatives. For example

$$(1) \quad (x_1 \cdot 2^{y_1}) \cdot (x_2 \cdot 2^{y_2}) = x_1 x_2 \cdot 2^{y_1+y_2}$$

$$\text{or} \quad (x_1, y_1) \cdot (x_2, y_2) = (x_1 x_2, y_1 + y_2)$$

$$(2) \quad (x_1 \cdot 2^{y_1}) + (x_2 \cdot 2^{y_2}) = (x_1 \cdot 2^{y_1} 2^{y_2} 2^{-y_2}) + (x_2 \cdot 2^{y_2} 2^{y_1} 2^{-y_1})$$

$$= (x_1 2^{-y_2} + x_2 2^{-y_1}) \cdot 2^{y_1+y_2}$$

$$\text{or} \quad (x_1, y_1) + (x_2, y_2) = (x_1 2^{-y_2} + x_2 2^{-y_1}, y_1 + y_2)$$

$$(3) \quad (x_1 \cdot 2^{y_1})^{-1} = (1/x_1) \cdot 2^{-y_1} = x_1^{-1} \cdot 2^{-y_1}$$

$$\text{or} \quad (x_1, y_1)^{-1} = (x_1^{-1}, -y_1)$$

$$(4) \quad -(x_1 \cdot 2^{y_1}) = (-x_1) \cdot 2^{y_1}$$

$$\text{or} \quad -(x_1, y_1) = (-x_1, y_1)$$

A number, x , does not have a unique representative since the integer, i , defined in (9.1) is not unique. For example

$$8 = 8 \cdot 2^0 = 4 \cdot 2^1 = 2 \cdot 2^2 = 1 \cdot 2^3 = 0.5 \cdot 2^4$$

Of particular interest is the form of the representatives for 1 and 0.

(1) Since

$$1 = 2^{-j} \cdot 2^{+j}$$

for any integer j , the representatives of 1 all have the form

$$(2^{-j}, j)$$

(2) Since

$$0 = 0 \cdot 2^j$$

for any integer j , the representatives of 0 all have the form

$$(0, j)$$

Let X have the representative, (x, i) . This is not unique. If a representative for X is multiplied by a representative for 1, the result is a representative for X

$$(9.2) \quad (x, i) \cdot (2^{-j}, j) = (x \cdot 2^{-j}, i + j)$$

It can be shown that all representatives of X will be obtained if j ranges over the values, $j = 0, \pm 2, \dots$

A normalization rule is any rule for selecting one of the couples in (9.2) to be the unique representative of X . The one selected is called the normalized exponential representative. Arithmetic carried out with normalized representatives will not in general yield normalized representatives so that the normalization rule will have to be applied to each result. We shall be concerned with two normalization rules.

In the first case we select the integer, j , in (9.2) so that

$$1/2 \leq x \cdot 2^{-j} < 1$$

Thus the normalized representative for 8 is

$$8 = +0.5 \cdot 2^4 \quad \text{or} \quad (+0.5, 4)$$

The resulting normalized representation is called a floating exponent representation.

(2) In the second case we select the integer, j , in (9.2) so that

$$i + j = 0$$

This is a rather trivial case in which the representative for X is the couple

$$(X, 0)$$

This representation is called a fixed exponent representation.

In both of the above cases the normalized representative of zero is simply defined to be the couple

$(0, 0)$

CHAPTER XI: THE WHIRLWIND I COMPUTER

I. Some Basic Concepts

A. The Whirlwind I computer (WWI) uses words in its operation. A word is simply an array of sixteen binary digits. For reference, the sixteen binary digits of a word are numbered from 0 to 15, counting from left to right.

B. A word may be used as either a number or an instruction. The distinction is made by the manner in which the word is used, not by the form of the word itself. The importance of the fact that numbers and instructions have exactly the same form cannot be overemphasized.

C. WWI has an arithmetic element in which words are processed. When a word is used in the arithmetic element, it is usually being treated as a number.

D. WWI has a control element in which words are obeyed. When a word is used in the control element, it is always being treated as an instruction.

E. WWI has a magnetic-core memory (CM) which contains 2016 storage registers or locations, numbered from 32 to 2047. The number which is used to refer to a register is called its address. Each register can hold one word. The registers of CM constitute the primary storage of WWI. Both the arithmetic element and the control element obtain the words they need from CM.

F. WWI has an auxiliary storage consisting of two magnetic drums and five magnetic tapes. Words may be transferred in either direction between CM and auxiliary storage. Words which are in auxiliary storage are usually brought into CM before they are used by the arithmetic or control elements.

G. To solve a problem using WWI, a sequence of words must initially be read in to CM. This sequence of words, comprising the numbers and instructions required to solve a problem, is called a coded program or a routine. The procedure of determining a suitable method for solving a problem is called programming; the process of translating this method into a coded program is called coding.

II. Numbers

A. When a word in WWI is considered as a number, digit 0 represents its sign and the remaining fifteen binary digits specify its magnitude. When the number is positive, digit 0 contains a binary 0 and digits 1 through 15 contain the magnitude as a binary number. The representation of a negative number is formed from the corresponding positive number by changing all its zeros to ones and all its ones to zeros. Thus, in a negative number digit 0 contains a binary 1 and digits 1 through 15 contain the complement of the magnitude.

Positive number: 0 000 110 011 001 101 $\hat{=}$ +.1000

Negative number: 1 111 001 100 110 010 $\hat{=}$ -.1000

It is not necessary to know the binary structure of a number when coding. Numbers may be read into and obtained from WWI in the decimal notation (the conversion to and from binary being done by the computer), so that the binary operation need not be of direct concern. It should, however, be remembered that no more than fifteen binary digits are available to represent the magnitude of a number. This is equivalent to about 4.7 decimal digits, and is the maximum precision obtainable without using special programming techniques.

B. All numbers are fractions; no number can be represented if it is greater than or equal to unity. Both positive and negative fractions may be used. During input, numbers may be specified as decimal fractions by typing them in the form

+ .2743

- .1279

+ .0

Decimal fractions are typed with an algebraic sign (+ or -) followed by a decimal point followed by the decimal digits of the fraction. More than four decimal digits may be typed if desired; the computer rounds off to fifteen binary digits automatically.

C. If he finds it convenient to do so, the programmer may affix to decimal fractions factors of the form $\times 10^\alpha \times 2^\beta$. For instance, the decimal fraction +.2743 may equally well be typed as

$$+.27.43 \times 10^{-2} \quad \text{or} \quad +.0002743 \times 10^3$$

Similarly, the fraction 1/4 will be obtained by typing any of the following:

$$+.25 \quad +25. \times 10^{-2} \quad +1. \times 2^{-2}$$

or even $+100. \times 10^{-2} \times 2^{-2}$. Note that these numbers are in generalized decimal form, and hence, it is necessary that the number obtained after taking all multiplying factors into account be less than unity.

D. Since the smallest discernible increment is 1×2^{-15} , all numbers may be expressed as $\pm N \times 2^{-15}$, where N is an integer. It is often useful to consider a number to represent the integer $\pm N$ (with the factor 2^{-15} always understood).* This is helpful for counting or for modifying addresses, as will be seen later. Numbers may be typed as decimal integers:

$$+17 \quad 19 \quad -3562 \quad +0$$

Decimal integers have an algebraic sign (+sign may be omitted) and no

*WWI always treats numbers as fractions. Since $(A \times 2^{-15}) \pm (B \times 2^{-15}) = (A \pm B) \times 2^{-15}$, it is immaterial which viewpoint is adopted when adding or subtracting. Care must be exercised, however, when multiplying or dividing numbers which the programmer is considering as "integers"; it is easy to see that the result is not another "integer".

decimal point. No factors of the form $x 10^\alpha \times 2^\beta$ may be used with decimal integers.

E. Occasionally, it is useful to specify a number in the octal notation, rather than in the decimal number system. Because of the direct correspondence between one octal digit and groups of three binary digits, use of the octal system is an easy way to specify the exact array of binary digits to go into a register. The octal system may be used also when the programmer knows the octal value of a number and does not want to bother to convert it to decimal.

Numbers may be typed as octal fractions. To specify an octal fraction, due account must be taken of the way positive and negative numbers are stored in WWI. The first symbol typed must be the sign digit, which is either 0 or 1. This is followed by a point (.), and then by exactly five octal digits to specify the contents of the remaining positions in the register. The numbers given in paragraph (A) above may be written as octal fractions:

0.06315	\doteq +.1000
1.71462	\doteq -.1000

Note that octal fractions do not contain the symbols + or -; the same information is specified instead by using the digits 0 or 1.

F. The number zero has two distinct representations in WWI. One of these has a positive sign and is written +0; the other, which has a negative sign, is written -0.* Either form of the number zero may be used in an arithmetic operation and will yield the correct result. When the answer to a computation is zero, there are simple rules for determining which of the two possible zeros results. These rules will be discussed later with the arithmetic instructions to which they apply.

* +0 consists of 16 binary 0's; -0 of 16 binary 1's. The reader may verify that this is in accord with the rules for positive and negative numbers in WWI.

III. Instructions

A. When a word in WWI represents an instruction, digits 0 to 4 are devoted to the operation section and the remaining eleven digits to the address section. The operation section indicates the nature of the instruction (addition, multiplication, etc.) and the address section normally contains the address of a register whose contents are to be used in the operation. In a few instructions, the address section is not the address of a register at all, but is used for other purposes. This will be explained when the instructions are discussed individually.

B. For input to WWI, instructions are typed as a two-letter mnemonic code followed by a floating, relative, or absolute address. These forms of address have been discussed earlier in connection with the CS computer, and the conventions are exactly the same for WWI. However, the two-letter WWI instructions should never be confused with the three-letter CS instructions. Examples of WWI instructions are:

cam5	dmt2+4	(floating addresses)
mr3r	cp2r	(relative addresses)
si384	cs32	(absolute addresses)

IV. The Instructions ca x, cs x, cm x; the Accumulator (AC); and the B-Register (BR)

A. Words are processed in WWI by use of the arithmetic element. This element contains within it several registers which take part in the information processing. The most important of these is the accumulator (AC), a sixteen-binary digit register which is used in most of the WWI instructions. It is the AC in which sums and products, for instance, are formed. Another 16-digit register, the B-register (BR), can be viewed in many cases as an extension to the right of AC and in some cases as a completely separate register. The arithmetic element also contains two other registers: the A-register (AR) and special-add memory (SAM). The uses of these registers will become apparent later.

B. It is usually necessary to bring the contents of a storage register into AC preparatory to further operations upon the word which it contains. For this purpose the following WWI instructions are defined:

ca x	<u>C</u> lear AC and BR and <u>a</u> dd C(x) to AC*
cs x	<u>C</u> lear AC and BR and <u>s</u> ubtract C(x) from AC
cm x	<u>C</u> lear AC and BR and add <u>m</u> agnitude of C(x) to AC

These instructions provide flexibility in bringing the desired form of a word in CM into AC. Note that they all clear BR and that they leave C(x) unchanged.

V. The Instructions ad x, su x, dm x, ab x, ao x; The Arithmetic-Check (Overflow) Alarm

A. The WWI computer has several instructions which are used for addition and subtraction. The simplest and most straightforward of these are

ad x	<u>A</u> dd C(x) to C(AC) and store sum in AC
su x	<u>S</u> ubtract C(x) from C(AC) and store difference in AC

Both of these instructions leave C(BR) and C(x) unchanged.

B. The instruction dm x is used for forming the difference of the magnitudes of two numbers.

dm x	Place in AC the quantity $ C(AC) - C(x) $, place the previous C(AC) in BR, leave C(x) unchanged
------	--

The fact that the previous contents of AC appears in BR after dm x is

*The descriptions of WWI instructions in these notes are brief and are intended only to point out the more significant features of the instructions. A complete listing of all WWI instructions is given on the attached drawing D-55192-2. Further reference should be made to it.

executed is often useful.

C. $C(BR)$ may be added to $C(x)$ by using the instruction

ab x Add $C(BR)$ to $C(x)$, store sum in AC and in x.
 Leave $C(BR)$ unchanged.

Since $C(BR)$ is unchanged, the ab x instruction facilitates adding the same quantity to the contents of several registers.

$C(BR)$ is treated by ab x just as the contents of any storage register. In particular, BR digit 0 is considered to be a sign digit, not a numerical digit. The need for emphasizing this point will become clear when those instructions which treat BR as an extension of AC are discussed later. The instruction dm x places numbers in BR in the form required by ab x.

D. It is very often desired to increase the contents of a storage register by 1×2^{-15} . If $C(x)$ is an instruction, this increases its address section by one. The instruction ao x makes this special case of addition very easy.

ao x Add one times 2^{-15} to $C(x)$ and store the sum both in x and in AC. Leave $C(BR)$ unchanged.

E. It has been mentioned that zero has two representations in WWI, +0 and -0. In general, a zero resulting from addition or subtraction is -0. In two cases only, +0 will be obtained. These cases are $(+0) + (+0) = +0$ and $(+0) - (-0) = +0$.

F. All numbers in WWI are fractions. It is obviously possible for the sum or difference of two such numbers to equal or exceed one. If this happens, the result cannot be represented in the computer and, in general, an alarm will occur. The alarm caused by this type of error is called the arithmetic-check or overflow alarm. When any alarm occurs, the computer stops with the contents of the registers of the arithmetic and control elements displayed in lights on the control panel.

Should a routine stop on an alarm, it indicates that a programming or coding mistake has been made, and it is, of course, necessary for the programmer to locate and correct the mistake. The "post-mortem" routines which are available to aid the programmer in this task are described in Chapter XVII.

VI. The Instructions ts x, td x, and ex x -- Storing Results

A. In order to retain a result which has been produced in AC, it is usually necessary to place the result in a storage register. This is accomplished by the instruction

ts x Transfer C(AC) to storage register x.
 Leave C(AC) unchanged.

The previous C(x) is lost after the instruction ts x is executed.

B. It will be remembered that the last eleven binary digits of an instruction constitute its address section. Often only the address section of an instruction in a storage register is to be modified, the operation section being unchanged. For this purpose there is provided the instruction

td x Transfer last 11 digits of C(AC) to last 11
 digits of register x. Leave C(AC) and first
 five digits of C(x) unchanged.

If C(x) is an instruction, td x causes its address section to be replaced by the address section of the word in AC. If C(x) is a number, td x may be executed, but the resulting C(x) will be a mixture of five digits remaining from the previous number and eleven new digits from AC. In general, this is meaningless. td x is a logical, not a numerical, operation.

C. An instruction which is frequently very convenient is

ex x Exchange C(x) and C(AC); i.e., place C(x) in AC
 and place previous C(AC) in x.

This instruction permits the coder to bring $C(x)$ into AC, but at the same time it also stores the previous $C(AC)$ in x . It thus combines two logically separate functions in one instruction. Note that $ex\ x$ does not change $C(BR)$.

VII. Transfer of Control, The Instructions $sp\ x$ and $cp\ x$ --
The A-Register

A. The WWI computer obeys instructions in sequence until a specific instruction which breaks this sequence is encountered. The instruction

$sp\ x$ Take next instruction from register x and
 continue obeying instructions in sequence
 from there.

permits the coder to specify a break in the sequence of control. The instruction $sp\ x$ is always obeyed; it is an unconditional transfer of control.

B. An extremely valuable instruction is $cp\ x$, which permits the programmer to make a transfer of control conditional on the result of the immediately preceding calculation.

$cp\ x$ If $C(AC)$ is negative, proceed as in $sp\ x$; if
 $C(AC)$ is positive, ignore this instruction and
 go on to the following instruction in sequence.

$cp\ x$ is the only conditional transfer of control instruction available in the WWI computer. All "decisions" in the computer which choose one sequence of operations rather than another are made using this instruction. It is possible to reduce virtually any criterion for choice among a number of possible routines to a sequence of suitable "yes-no" decisions. It is therefore possible, using only the $cp\ x$ instruction, to realize even extremely intricate and elaborate decision criteria in the computer.

C. Whenever an instruction is executed by WWI, that instruction must, of course, be located in a storage register. Each storage register has an address, y . Whenever an $sp\ x$ or $cp\ x$ instruction is executed in register y , the address $(y+1)$ is stored in the A-register (AR), another register of the arithmetic element. The AR is a sixteen-binary-digit register, but only the last eleven digits are affected by $sp\ x$ or $cp\ x$. The address y is the register in which $sp\ x$ or $cp\ x$ is located; it should not be confused with register x , the address within the $sp\ x$ or $cp\ x$ instruction itself. The address $(y+1)$ is stored in AR on $cp\ x$ even when $C(AC)$ is positive and $cp\ x$ is otherwise effectively ignored.

VIII. Closed Sub-Routines -- The Instruction $ta\ x$,

A. The use of sub-routines in the solution of a problem has already been discussed in connection with the CS computer. It will be recalled that a subroutine is a sequence of instructions which may be entered from several points in a larger routine. Most often, it is desired that a subroutine be closed; i.e., it is desired that it return, when it is finished, to the main routine at the point from which it was entered.

B. In writing closed subroutines for the WWI computer, the instruction

$ta\ x$	<u>Transfer</u> the last eleven digits of the <u>A-register</u> to the last eleven digits of register x . Leave the operation section of register x and the contents of all other registers unchanged.
---------	--

is invaluable. Subroutines are invariably entered using the operations sp or cp . The address section of the AR is set equal to $(y+1)$ by these instructions. If the initial register of a subroutine contains the instruction $ta\ x$, the address $(y+1)$ may be inserted into any register x of the subroutine. In particular, it may be inserted into the sp or cp operation which is used to leave the routine, and in this case the subroutine is closed.

Another use for `ta x` will be discussed with the instruction `sf x`.

C. One caution must be observed in using the instruction `ta x`. While it has not been explicitly stated in these notes, most WWI instructions change `C(AR)`. `ta x` must be used immediately after `cp x` or `sp x` before any instructions which modify `C(AR)` are executed. `ta x` should be the first instruction of a closed subroutine.

IX. The Instructions `mh x`, `mr x`, `dv x` -- The Divide-Error Alarm

A. WWI has two instructions which are used to multiply numbers.

<code>mh x</code>	<u>M</u> ultiply <code>C(AC)</code> by <code>C(x)</code> and <u>h</u> old the full thirty-binary-digit product in <code>AC</code> and the first 15 digits of <code>BR</code> , treating <code>BR</code> as an extension to the right of <code>AC</code> .
<code>mr x</code>	<u>M</u> ultiply <code>C(AC)</code> by <code>C(x)</code> and <u>r</u> ound off the product to fifteen binary digits in <code>AC</code> . Clear <code>BR</code> .

It is clear that the product of two fifteen-digit numbers is a thirty-digit number. The full thirty-digit product may be retained by using `mh x`; the product may be rounded off to one register length by using `mr x`.

B. In both the instructions `mh x` and `mr x` the sign of the product is determined according to the usual rule for multiplication. In particular, this is true when one of the factors is `+0` or `-0`; the sign of the product is still determined from the ordinary rule, giving to each of the zeros its appropriate sign.

C. As was mentioned earlier, negative numbers in WWI are stored as the complement of the corresponding positive number. The fact that the complementary form is used within WWI is normally of little significance to the coder. However, it is a peculiarity of the computer

that the B-register is never complemented when the result of a multiplication extends into it. The digits in BR always appear as their positive magnitude, even though the digits in AC are complemented if the product is negative. The sign which should be associated with C(BR) in this case is the sign of C(AC). Digit 0 of BR is not a sign digit; it contains the first numerical digit of C(BR). Digit 15 of BR will contain a zero.

In order to obtain the digits in BR after the mh x instruction with their proper sign, it is most convenient to use a suitable numerical-shift instruction to move them into AC. The shift instructions will be described in detail later.

D. WWI has one divide instruction:

dv x Divide C(AC) by C(x), storing the quotient in BR.

After the execution of dv x, AC contains a zero of the same sign as the quotient. The quotient is in BR, but it is uncomplemented (just as in the case of mh x) even though it may be negative. Again, a numerical-shift instruction is required in order to bring the quotient into AC with its proper sign,

E. The quotient which appears in BR has sixteen numerical digits. There is room in AC (or in a storage register) for only fifteen numerical digits. Consequently, the sixteenth digit cannot, in general, be retained. The extra digit, however, permits round-off to fifteen digits to be accomplished if a rounded quotient is desired. dv x should, in general, be followed by slh 15 or, to obtain round-off, slr 15.

F. If the dividend exceeds the divisor, the result will exceed the capacity of the computer. Should this mistake occur, the computer will stop on a divide-error alarm.

X. The Numerical-Shift Instructions -- slh n, slr n, srh n, srr n.

A. We are all familiar with the procedure of multiplying a

decimal number by 10^N simply by moving the decimal point to the right N places. An analogous procedure exists in the binary number system, in which moving the binary point to the right n places multiplies a binary number by 2^n . Similarly, moving the binary point to the left n places divides a binary number by 2^n .

B. In WWI, the binary point is fixed at the left immediately following the sign. The binary point cannot be moved, but the same effect may be achieved by shifting the number itself with respect to the fixed binary point. If the number is shifted to the right, it is equivalent to moving the binary point to the left, and vice versa.

C. The numerical-shift instructions in WWI provide a means for shifting the combined contents of AC and BR to the right or left, thereby dividing or multiplying by the corresponding power of two. The shift-left instruction is useful for bringing $C(BR)$ into AC. Since these are numerical shift instructions, the sign digit (digit 0) of AC is not shifted; only those digits of AC and BR which actually are numerical take part in the shift. When the sign of AC is negative, $C(AC)$ is assumed to be complemented, but $C(BR)$ is not. This corresponds to the manner in which negative results are stored in AC and BR after the instructions `mh x` and `dv x`.

`slh n` Shift the combined AC and BR to the left n^* places. Hold all digits in AC and BR after the shift.

`slr n` Shift the combined AC and BR to the left n places. Round off the $C(AC)$ on the basis of the magnitude of $C(BR)$ after the shift. Then clear BR.

`srh n` Same as `slh n`, only shift is to the right.

* n is taken modulo 32.

srr n Same as slr n, only shift is to the right.

In all four of these instructions, digit 0 of AC is not shifted, any digits shifted left out of AC 1 or right out of BR 15 are lost, and if $C(AC)$ is negative, AC is complemented before and again after the shift.

D. Note that these instructions differ somewhat from the form of the other WWI instructions discussed so far. First of all, three letters are required to specify the shift operations instead of two. All three must be typed or ambiguity will result.

A more significant difference is that the address section of these instructions does not refer to a storage register at all, but specifies by how many places the number is to be shifted. Since only AC and BR are involved in these instructions, no storage register need be specified and the address section may be used for this purpose.

The operation sections of slh n and slr n are identical, as are the operation sections of srh n and srr n. The distinction between these instructions is made not by the operation section but by digit 6 (the second digit of the address section). Since the address n is small, digit 6 may be used without causing any difficulty. In slh n and srh n, digit 6 must be a one; in slr n and srr n, it must be a zero. If the address of one of these instructions is changed by using a td x instruction, the coder must remember to preserve the correct value of digit 6.

E. Note that after a shift instruction, $C(AC)$ may be so large that round-off can cause it to equal unity.* In this case, if the instruction calls for round-off, the arithmetic-check or overflow alarm will result.

XI. The Logical Cycle Instructions -- clh n and cle n.

A. On occasion it is desirable to move the digits of a word as it stands to the left or right without regard to the numerical significance

* This occurs when $C(AC) = 1-2^{-15}$

of the digits. In this case, the sign digit is treated like any other digit; the process is simply one of reorienting the digits without regard to their meaning either as a number or an instruction.

Two instructions are available for executing this operation:

clh n Cycle the combined AC and BR to the left by n places. Carry any digits cycled out of AC 0 into BR 15, Hold all digits at the end of the cycle.

clc n Same as clh n, except clear BR after the cycling.

B. Note that the AC and BR are treated as a closed ring; digits cycled out of the left of AC appear at the right of BR. No round-off occurs. Digit 6 of clc n must be zero; digit 6 of clh n must be 1.

Logically, the shift and cycle instructions are quite different; the programmer should remember that the shift instructions are numerical and the cycle instructions are logical in function.

XIII. Scale Factoring -- The Instruction sf x.

A. Fractions may, in general, have one or more zeros between the binary point and the first significant digit. These zeros are not significant, in the sense that the fraction may equally well be expressed as another fraction (having no initial zeros) times 2^{-N} , where N is the number of initial zeros in the original fraction. This latter form has the advantage that its numerical value may be expressed to full fifteen-binary-digit precision. However, in performing arithmetic operations on numbers expressed in this form, due account must be taken of the factor 2^{-N} associated with each number; they cannot be combined directly using WWI arithmetic instructions.

B. The instruction

sf x Scale factor the combined AC and BR; i.e., shift the contents of AC and BR left until there are no initial zeros. Store N, the number of times shifting was necessary, in the address section of C(x) and of C(AR).

permits numbers to be expressed easily in scale-factored form. Note that the procedure for handling the scale-factored numbers must be coded by the user; in the WWI computer, no automatic facilities exist for taking scale factors into account. sf x treats C(AC + BR) numerically, just as do the shift instructions.

Since N appears in AR as well as in register x, it may be placed in other registers also by using the ta operation immediately following sf x. The operation section of C(x) is unchanged by sf x; this should normally be zero before the sf x instruction is executed. If C(AC + BR) is +0, N is set equal to 33.

XIII. The Instruction sa x -- Special-Add Memory

A. Normally, if the result of an addition is as large as unity in magnitude, an overflow alarm occurs. Occasionally, the coder will find it convenient to permit an overflow to occur without alarm, taking account of the overflow later in the routine. The instruction sa x, special-add x, differs from the instruction ad x only in its behavior in the event of overflow.

sa x Add C(x) to C(AC), store the fractional part of the sum in AC. Store the integer part of the sum (0, +1, or -1) times 2^{-15} in special-add memory (SAM). Give no overflow alarm.

SAM is a special register of the arithmetic element. Its only possible contents are 0, +1, or -1 (times 2^{-15}), and these are stored in SAM only

by the sa x instruction.

B. The contents of SAM after the instruction sa x is executed may be used by executing one of the operations ca, cs, or cm. When these operations were first discussed, it was not mentioned that C(SAM) is added to the word which is brought into AC by these instructions and that the sum is left in AC. When $C(SAM) = 0$, this addition does not change C(x) and the ca x, cs x, and cm x instructions behave as was described earlier. SAM is always cleared (i.e., set equal to zero) by the instructions ca x, cs x, or cm x after they have used its contents. Note that an overflow alarm will occur on these instructions if the addition of C(SAM) causes the number which is to be placed in AC to equal unity.

C. The execution of any of the following instructions clears SAM without using its contents: ab x, ad x, su x, ao x, dm x, mr x, mh x, dv x, slr n, slh n, srr n, srh n, sf x.

D. SAM may always be assumed to be clear after the read-in of a program tape.

XIV. The Instructions md x and sd x

A. A logical instruction useful for retaining certain digits of a word while setting the other digits equal to zero, is

md x Logically "multiply" each digit of C(AC) by the same digit in C(x) and place result in AC,

The effect of this instruction is to set the digits of AC which correspond to zeros in x equal to zero, and to leave the digits of AC which correspond to ones in x unchanged.

B. Another useful logical instruction is

sd x Logically, form the "sum" of each digit of C(AC) and the same digit in C(x) and place the result in AC.

The effect of this instruction is to place 1's in those digit positions of AC whose original contents differ from the corresponding digits in x and to place 0's in those positions in which C(AC) and C(x) were originally the same. One possible use for sd x is in checking the identity of two computed results.*

C. md x and sd x are non-arithmetic instructions; they treat C(AC) and C(x) simply as arrays of digits, without regard for their numerical significance.

The two instructions are summarized in the following two tables, which show the final contents of each digit position in AC as a function of the initial contents of that digit position and the contents of the same digit position in register x.

		AC
	md x	0 1
	0	0 0
x	1	0 1

		AC
	sd x	0 1
	0	0 1
x	1	1 0

In the terminology of symbolic logic, the function represented by md x is "and", the function given by sd x is "exclusive or."

* The ck x instruction (see next page) performs such a check, but offers no alternative except stopping the computer if the results do not agree.

XV. The Instruction ck x -- The Check-Register Alarm

A. The instruction

ck x Compare C(AC) and C(x). If they are identical, proceed to the next instruction; if they differ, stop the computer in a check-register alarm.

enables the coder to stop the computer in case a computed word does not agree exactly with some predetermined value.

B. The instruction ck x is rarely used in mathematical computations. It finds its widest application in coding which involves use of auxiliary equipment by the computer. It may be employed, for instance, to insure that information has been transferred correctly from an external unit to the computer.

XVI. The Instructions si 0 and si 1

The following instructions can be used to stop the computer:

si 0 Stop the computer.
 si 1 Stop the computer if the "STOP ON si 1" switch is on, otherwise continue to the next instruction.

The "STOP ON si 1" switch is on the control console and can be set by the computer operator.

The coder should not normally use these instructions to stop the computer. Instead he should use the special instructions:

STOP Stop the WWI computer
 iSTOP Stop the CS computer

Reasons for this will be given subsequently.

XVII. Some Examples of Operations on Instructions

The use of the WWI computer instructions for executing operations on numbers is relatively straightforward. The arithmetic operations, such as ca x, ad x, su x, mr x, dv x, are used in much the same way as the analogous CS computer instructions. It must be remembered, however,

that the WWI instructions operate on single-register WWI numbers, while the instructions of the CS computer operate on the two-register numbers of the CS computer. The distinction between the two computers must always be kept clearly in mind by the coder.

Since numbers and instructions in the WWI computer have the same form, there is nothing to prevent the coder from executing arithmetic operations on instructions instead of on numbers. At first glance, it might appear that this is a mistake which should be avoided. However, quite the contrary is true; in fact, it is the ability to operate on instructions and numbers with equal facility that gives the WWI computer much of its flexibility and usefulness.

As a simple example of this concept, let us consider the problem of modifying the address section of an instruction. Let us suppose that we have (as part of our coded program) the instruction, su b3, and that this instruction is in register m5. This means, as you recall, that we had typed in our routine

```

      .
      .
      .
m5,sub3
      .
      .
      .
      .

```

We wish, let us say, to modify this instruction during the course of the computation so that it becomes sub3-6. That is, we wish to change the instruction so that it no longer refers to register b3 but refers instead to six registers before b3. This may be accomplished with the following sequence of instructions

```

      .
      .
      .
m5,sub3      .          cl,-6 Constant
      .
      .
      .

```

(continued on next page)

```

cam5      Obtain sub3 in AC
adcl      Form sub3 + -6 = sub3-6
tsm5      Store sub3-6 in m5
.
.
.

```

Here, an instruction (in m5) and a number (in c1) have been added to produce an instruction with a modified address.

Note that the number in c1 is stored as a decimal integer (i.e., the factor $x2^{-15}$ is understood). This is consistent with the fact that the address section occupies digits 5-15 of an instruction; both the address section and the decimal integer occupy the final digit positions of the register.*

The same change in the sub3 instruction may be made in other ways. Two possible procedures are:

```

.
.
.
m5,sub3      c2, sub3-6
.
.
.
cac2      Obtain sub3-6 in AC
tsm5      Store sub3-6 in m5
.
.
.

```

or, using the td x operation,

```

.
.
.
m5,sub3      c3,adb3-6      (or any instruction
.                                     which has the ad-
.                                     dress section b3-6)
.
.
cac3      Obtain adb3-6 in AC
(continued on next page)

```

* If the sum of the address and the integer exceeds 2047 (or is less than 0), the addition will affect the digits devoted to the operation section, producing not only a new address but also a new instruction.

tdm5 Transfer the address section only from AC to m5.

•
•
•

The coder often has considerable flexibility in using the WWI instructions to manipulate other instructions. It is generally best to use the procedure which requires the fewest additional registers or (if the routine is so short that storage is not a problem) the shortest possible time.

Observe that the word in register c3 has the form of an instruction, although, as we have used it, it is never obeyed. If any instruction with the address section b3-6 had already been employed in our routine, it would not have been necessary to add another word having this address section; we would simply have tagged as c3 the register containing the instruction of the desired form. It invariably shortens a routine if the same word can be used at different times for more than one purpose.

Let us consider now the problem which is handled in the CS computer by using the cycle-counting facility. This facility, as you remember, permits going through a "loop" of instructions a predetermined number of times, appropriately indexing selected instructions within the loop at each repetition. There is no cycle-counting facility in the WWI computer. Counting and address modification must be coded explicitly using the WWI instructions. An example of a loop which is repeated 40 times is given below.

a1,cab3

•
•
•
•

a2,sub4

•
•
•
•

a3,tsd1

•
•
•
•

c1, -39 Counter

} Cycle of instructions to be repeated 40 times.
Instructions in registers a2 and a3 are to be indexed each cycle.

(continued on next page)

```

    aoa2          Index instructions
    aoa3
    aoa1          Index counter
    cpal          If counter is negative, repeat.  When counter
    .             becomes positive, continue to next instruction
    .             in sequence
    .

```

Note the use of the `ao` operation for indexing and for counting. (The cycle is executed 40 times. Why is the counter in `cl` set to `-39` instead of `-40`?)

The words in `a2`, `a3`, and `cl` are actually changed by the `ao` operation. Hence, these are left with their "completed" contents after the cycle is finished. If the cycle is used only once, this is quite permissible. However, if the same cycle is to be reused, the changed registers within the routine must be reset to their initial values.

```

ENTRY   cac4          c4,-39          Constants for
POINT  tscl          c2,b4          resetting purposes
       cac2          Reset changed   c3,d1
       tda2          registers      cl,0
       cac3
       tda3
       al,cab3
       .
       .
       .
       a2,su0
       .
       .
       .
       a3,ts0
       .
       .
       .
       aoa2
       aoa3
       aoa1
       cpal
       .
       .
       .

```

Registers which are reset
to initial values

The six instructions preceding the loop simply restore the initial contents of registers which are changed while the loop is executed.

The reader will find it instructive to rewrite the routine just presented so that it is somewhat shorter than the version just given.

To do this, eliminate entirely the need for a counter set to -39 by utilizing the fact that the instruction a0a3 places the new contents of a3 in AC as well as in a3. Remember that subtraction of two instructions is permissible.

XVIII. Parity Alarm

The coder may occasionally encounter an alarm which has not been discussed with any of the WWI instructions.

Each time a core-memory register or a register on the magnetic drum is referred to, a so-called "parity check" is carried out to determine whether one of the digits in the register has changed since it was recorded. In virtually all cases the parity check is completed successfully and computer operation continues.

Occasionally, however, a computer malfunction causes a faulty recording or a change in the information recorded in a register. When the parity check detects this, the computer stops on a parity alarm. A parity alarm may also be obtained if a non-existent magnetic-drum group is selected by the coder.

Should a routine stop on a parity alarm not traceable to an improper drum reference, it may be some consolation to the coder to know that the alarm results from an error made by the computer and not by coder.

XIX. Test Storage -- Registers 0 and 1

A. It has been mentioned that the registers of core memory are numbered starting at address 32. Registers 0 through 31, which are not part of core memory, exist and are referred to as test storage. Most of the registers of test storage have their contents set into them by toggle switches; the contents cannot be changed by a program. Five of the test storage registers are flip-flops, a form of storage register whose contents can be changed.

B. In general, instructions referring to test storage should not be used. Some exceptions exist.

C. The coder may make use of the contents of registers 0 and 1. Register 0 always contains the number, +0; register 1 always contains the number, +1. These constants are frequently needed in a program. Any instruction which tries to alter the contents of a test storage register will be executed but will not succeed in changing the contents of the register. Thus, the instruction, ao 1, can be used to place the integer, +2, in the AC.

D. The instructions, sp 0 or cp 0, can be used to stop the computer since the integer, +0, is also the instruction si 0.

XX. The Control Panel

A. Various buttons on the control panel are used by the computer operator to start and stop the computer. An understanding of these is of value to the coder.

B. The START-OVER button enables the operator to start the computer at any selected register of core memory. The address of the register must first be entered by the operator into a set of toggle switches called the PC RESET switches. For this reason the address should be specified in octal.

C. The START OVER AT 40 button starts computer operation at register 32 (this is equivalent to 40 in the octal system).

D. The STOP stops the computer and is used when it is necessary to stop the computer manually. Normally, programmed stops are used.

E. The RESTART button causes the computer to re-commence operation at the register immediately following the one in which it stopped. It may be used either following the instructions si 0, si 1, STOP, or i STOP, or after the STOP button has been pushed.

CHAPTER XII: AUXILIARY STORAGE, INPUT-OUTPUT EQUIPMENT

I. Introduction

The primary storage of WWI consists of 32 toggle switch registers (TS) numbered, 0, 1, ..., 31, and 2016 registers of core memory (CM) numbered, 32, 33, ..., 2047. The primary storage is directly addressable (addresses of instructions can denote operands in the storage) and has random access (a fixed amount of time, 8 usec, is required to obtain a word from the storage, regardless of its address).

Additional forms of storage (called auxiliary storage) are available on WWI which are not directly addressable. These are the following:

- (1) Magnetic Drums
- (2) Magnetic Tapes
- (3) Punched Paper Tapes

Information in auxiliary storage must be brought into CM before computations can be performed with it. Devices provided for realizing this transfer (along with certain others) are called input-output devices.

The primary input medium for WWI is punched paper tape (generally prepared on a Flexowriter typewriter). Paper tapes can be read into the computer on one of the following input devices:

- (1) A mechanical tape reader.
- (2) A photoelectric tape reader.

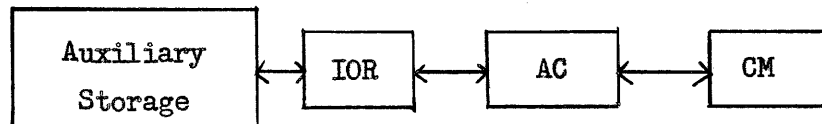
The following output devices are available to WWI:

- (1) A display oscilloscope
- (2) A paper tape punch
- (3) A Flexowriter typewriter

In addition an auxiliary tape punch and typewriter have been provided which can proceed under the control of characters recorded on magnetic tape.

II. The Input-Output Instructions

Transfers between auxiliary storage and CM along with use of the input-output equipment proceed under the control of 5 WWI instructions called in-out instructions. These make use of a special 16 bit register called the in-out register (IOR). Transfers between auxiliary storage and CM proceed through the IOR and AC.



The instruction

si n select input-output device n

selects the in-out device and mode of operation designated by n. An si instruction is normally followed by one of the other in-out instructions (rd, rc, bi, bo). Any piece of equipment can be deselected by selecting a new piece of equipment.

The instruction

rd- read

copies C(IOR) into AC and clears IOR. The contents of its address section is immaterial.

The instruction

rc- record

copies C(AC) to auxiliary storage (via IOR). Its address section is also immaterial.

The instructions

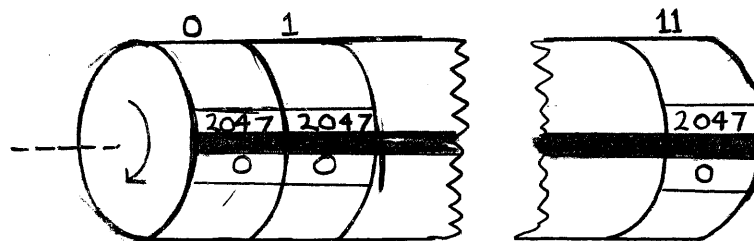
bi x block input to x

bo x block output from x

transfer blocks of words between CM and auxiliary storage. The address, x , specifies the initial register of CM involved.

III. The Magnetic Drums

Two magnetic drums, called the auxiliary drum (AD) and the buffer drum (BD) are available to WWI. The two drums are physically similar: metal cylinders, 8.5 in. in diameter, 13 in. long, rotating at approximately 60 revolutions per sec. (16.7 ms per revolution). Each drum is divided into 12 parallel sections called groups and numbered 0, 1, ..., 11. Each group consists of 2048 words which are numbered 0, 1, ..., 2047. A word is composed of 16 binary digits.



All 12 groups on the AD are available to WWI, however groups 0 and 11 are not available without restriction to the coder (this will be subsequently explained). Only groups 2, 3, ..., 7 of the BD are available to WWI and none of these are available without restriction to the coder (also to be explained later).

The address (n) of the i -th word of the g -th group is defined by

$$n = 2048g + i$$

In other words, the auxiliary drum registers have addresses ranging from 0 to 24575, and the buffer drum from 4096 to 16383. If n is written as a 15 binary digit number it will be noticed that digits 1-4 contain the group number (g) and digits 5-15 contain the location within the group (i).

A. Recording on the Drum

The drum is selected by one of the instructions

si 967 select AD, record mode.

si 975 select ED, record mode.

The address on the drum in which recording is to take place is specified by C(AC).

The instruction, rc, copies C(AC) into IOR and thence to the drum. The average time required to get the word from IOR to the drum is 1/2 revolution (8.3 ms)*. Any instruction other than an in-out instruction can be executed during this time. Any number of rc instructions can follow the si. Words are recorded on successive drum registers.

Any rc instruction can be replaced by the instruction, bo x, which records the block of n words (where $C(AC) = n \cdot 2^{-15}$) beginning at CM register x on the drum. 8.3 ms (1/2 revolution) is required to record the first word. Succeeding words, however, require only 16 μ s for the transfer.

As an example we record C(a1) in drum register d1, C(a2) and C(a2+1) in d1 + 1 and d1 + 2, and C(a3) in d1 + 3.

ca m1	} Select AD, reg. d1	m1, d1
si 967		m2, + 2
ca a1	} Record C(a1)	
rc		
ca m2	} Record C(a2)	a1, C(a1)
bo a2		and C(a2 + 1)
ca a3	} Record C(a3)	C(a2 + 1)
rc		a3, C(a3)
STOP		

* This cannot take place until the drum register in which the word to be recorded is under the recording heads of the drum.

B. Reading from the Drum

The drum is selected by one of the following instructions.

```

    si 963          select AD, read mode
    si 971          select BD, read mode
  
```

The drum address is specified by C(AC). The si brings the contents of the specified drum register into IOR. The average time required for the transfer is 1/2 revolution (8.3 ms). Any instruction other than in-out instructions can be done during this time. One and only one rd or bi instruction must occur after each si.

The instruction, rd, copies C(IOR) into the AC and clears IOR.

The instruction, bi x, copies a block of n words (where $C(AC) = n \cdot 2^{-15}$) into a block of registers in CM whose initial address is x. Only 16 μ s per word is required for the transfer.

As an example we copy the contents of BD register a1 into CM register b1.

```

    ca ml } Select BD, reg. a1      ml, a1
    si 971 }
    rd    } Store in b1
    ts b1 }
    STOP
  
```

IV. The Magnetic Tapes

Five magnetic tape (MT) units (numbered 0, 1, 2, 3a and 3b) are available to WWI. The reels of tape are from 800 - 1000 ft. long. Tapes are set in motion by means of si instructions (which specify the unit, the direction of motion and the mode of operation). Once selected a tape continues its motion until another si is executed. Tape moves under the reading-recording heads at a speed of 30 in. per sec. If an si instruction affects the motion of an MT unit (e.g., a moving tape is deselected, a tape moving forward is selected to move backward, a new tape is selected), approximately 5.5 ms delay is required for the tape unit to complete the mechanical change.

MTO is not available to the programmer. MTI may be used only under certain restrictions (later explained). Only one of the units, 3a and 3b can be connected to WWI in the same program. For coding purposes both of these are called MT3.

The normal position for a reel of tape is for the reading-recording heads to be at one end of the tape (this point is called the limit stop). The forward direction on tape is by definition away from the limit stop.

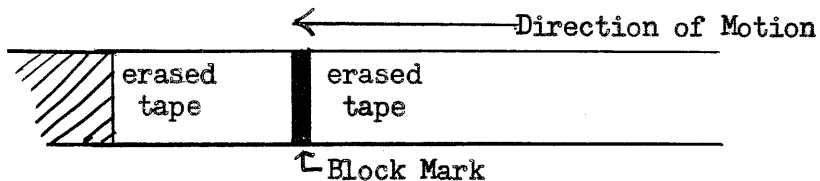
A. Recording on Magnetic Tape

The following si instructions are used to select the unit and its direction of motion

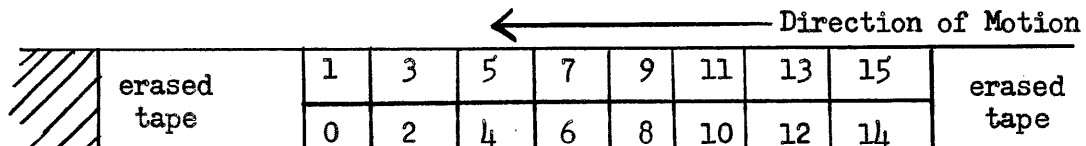
si 70 + 8j select MT j, record forward j = 0, 1, 2, 3

si 71 + 8j select MT j, record reverse J = 0, 1, 2, 3

The si instruction switches the unit to the record mode* and starts it in the specified direction. After a 14.3 ms delay (5.1 of which may be required to bring the tape to full speed) a special character called a block mark is recorded on the tape. Any instructions other than in-out instructions may be done during this time. The unit is left in the record mode.



The instruction, rc, records C(AC) on the tape (via IOR). This requires a 2.6 ms delay during which no in-out instructions can be performed. Any number of rc instructions may follow an si instruction. a 16 bit word is recorded on magnetic tape in the following form



The unit is left in the record mode.

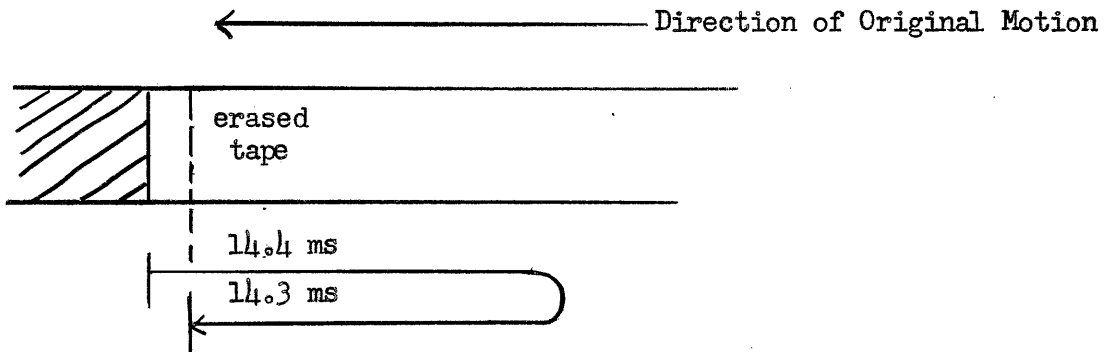
* If a tape unit is in the record mode, tape which passes under the reading-recording heads is erased.

The following si instructions should be used to deselect the tape unit after recording new information.

```

si 68 + 8j      select MT j, stop after record forward
si 69 + 8j      select MT j, stop after record reverse
                j = 0, 1, 2, 3
    
```

The si instruction delays for 14.4 ms, switches the unit to the record mode in the opposite direction, delays for 14.3 ms and deselects the unit.



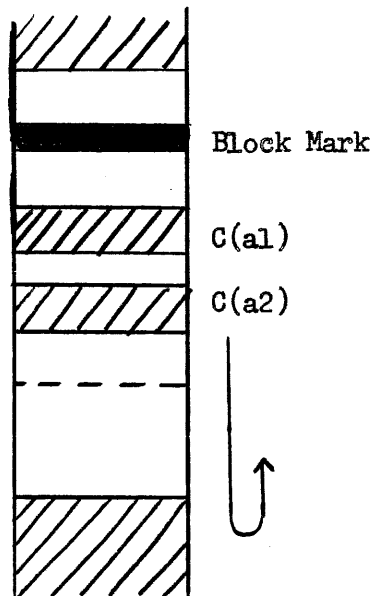
Use of this si instruction guarantees that a tape unit will stop in erased area. If a tape unit is stopped by selecting another piece of equipment it is switched immediately to the read mode and in coasting to a stop may pass over previously recorded information. This technique for stopping a unit may be used, however, if it is known in advance that the tape has been erased.

As an example we record C(a1) and C(b1) on MT2 in the forward direction.

```

si 86
ca al
rc
ca bl
rc
si 84
STOP
    
```

↑
Forward
Direction



B. Reading from Magnetic Tape

The following si instructions are used to select the unit and the direction of motion

si 66 + 8j select MT j, read forward j = 0, 1, 2, 3
 si 67 + 8j select MT j, read reverse

The si instruction switches the unit to the read mode (tape is not erased) and starts it in the specified direction. After a delay of 5.1 ms (to allow the unit to attain full speed) the unit looks for a block mark and reads the next 8 lines on tape into the IOR to form a 16 bit word. Any instruction other than an in-out instruction can be done during this time. The si instruction must be followed by at least one rd or bi instruction.

The instruction, rd, copies C(IOR) into the AC and clears IOR. Any number of rd instructions can follow an si. Since magnetic tape is a free running device words keep coming into the IOR as fast as they appear at the reading heads. A program alarm will occur unless information is removed from the IOR (by rd instructions) before new information appears.

Any si instruction selecting another piece of equipment will serve to deselect a tape unit in the read mode. If no specific si instruction is required then the instruction

si 408 deselect in-out equipment
 should be used.

The instruction, bi x, copies the next n words (where $C(AC) = n \cdot 2 - 15$) into a block of CM registers whose initial address is x. An si instruction must be given for each bi instruction used. Words recorded in one direction can be read in the opposite direction but will reappear in a scrambled form. Thus if the word

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
---	---	---	---	---	---	---	---	---	---	----	----	----	----	----	----

is recorded in the forward direction and read in the reverse direction it appears in the AC as

14	15	12	13	10	11	8	9	6	7	4	5	2	3	0	1
----	----	----	----	----	----	---	---	---	---	---	---	---	---	---	---

i.e., in reverse order by block of 2 bits. The unscrambling of the word to its original form is an intriguing coding exercise.

C. Recording on Tape

The following si instructions are used to select the unit and the direction of motion.

si 64 + 8j select MT j, rerecord forward

si 65 + 8j select MT j, rerecord reverse

The si instruction switches the unit to the read mode (tape is not erased) and starts it in the specified direction. After a delay of 5.1 ms (to allow the unit to attain full speed) the unit starts looking for a block mark and after finding one switches to the record mode. Any instructions other than in-out instructions can be done during this time. The si can be followed by any number of rc instructions.

The rerecord si can be used to skip over blocks of recorded information without disturbing them. This is done by switching the unit to the read mode (by a read si or another rerecord si) immediately after the block mark has been found. Once the desired block has been found its contents can be changed using rc instructions.

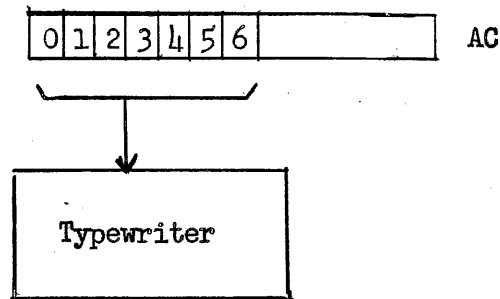
V. The Typewriter

The typewriter is selected by the instruction

si 149 select typewriter

Each character to be typed must be sent to the unit from the right-hand 6 bits of the AC by an rc instruction. An arbitrary six bit code has been defined by the manufacturer and is given below.

The instruction, bo x, can be used to print a block of n characters (where $C(AC) = n \cdot 2^{-15}$). The n characters must be stored (in the form described above) in a block of CM registers whose initial address is x.



It takes 125 ms to print each character (not a tab or carriage return). It takes up to 900 ms to print a tab or carriage return. A table of the Flexowriter characters printed by the typewriter is given in Chapter V.

VI. The Oscilloscope and Camera

Individual points can be displayed on a cathode ray oscilloscope by specifying the coordinates, (x,y) , where $|x| < 1$ and $|y| < 1$.

The instruction

```
si 384      select oscilloscope
```

is used to select the scope and specify the y coordinate (the first 11 digits of the AC, including the sign digit, are used for this purpose).

The instruction, rc, is used to specify the x coordinate (the first 11 digits of the AC are again used) and to display the point. Any number of rc instructions can follow the si. A delay of 170 μ s occurs after each rc during which time no in-out instructions can be performed.

Points displayed on the scope are automatically photographed by a camera whose shutter is always open. The instruction

```
si 4        index camera
```

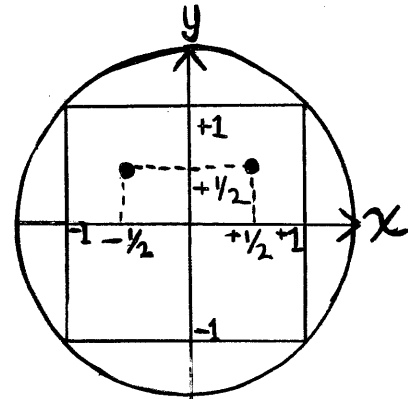
is used to index the film in the camera. A delay of some 500 ms occurs after an si 4 during which time no in-out instruction can be performed.

As an example we display two points with coordinates, $(-1/2, 1/2)$ and $(1/2, 1/2)$, on the oscilloscope

```

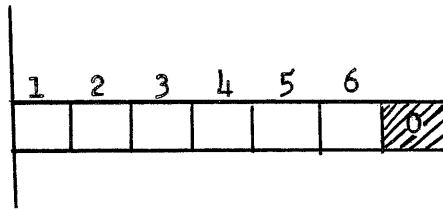
si 4      Index camera
ca      }
si 384 }   Select y = 1/2
ca      }
rc      }   Display (-1/2, 1/2)
ca      }
rc      }   Display (1/2, 1/2)
STOP

```



VII. The Photoelectric Tape Reader

The photoelectric tape reader (PETR) is the primary input device for punched paper tape. The paper tape contains 7 parallel channels, 6 of which contain information (under the convention that a hole in a channel denotes a 1). A hole is punched in the 7-th channel whenever information occurs in the other 6 channels.

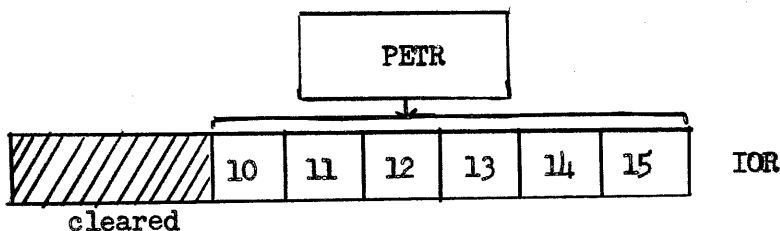


10 lines of information can be stored per in. on tape. The PETR reads information from tape either one line at a time (line by line) or 3 lines at a time (word by word) and places the information in the IOR.

The instruction

```
si 137      select PETR, line by line
```

starts the PETR and reads the first line of information into the right-hand 6 bits of the IOR. The si instruction must be followed by at least one rd or bi instruction.



The instruction, rd, transfers C(IOR) to the AC and clears IOR. Any number of rd instructions can follow the si. The PETR is a free running device (at 210 lines per sec.) so that lines of information go into IOR as soon as they appear on tape. If no blank lines occur on tape this means that lines of information will arrive every 4 ms and that rd instructions removing information from IOR must occur more frequently.

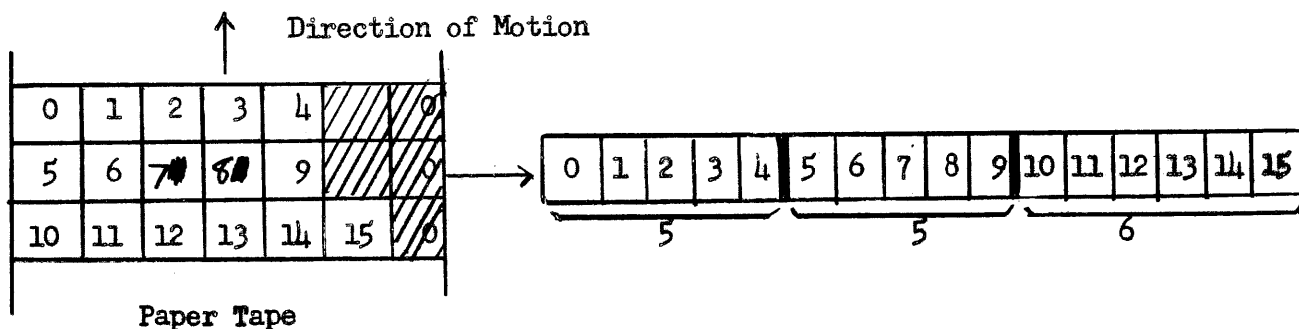
Any si instruction selecting another piece of equipment may be used to deselect the PETR. If this is done within 2 ms after reading a line then the next line on tape may be read by selecting the PETR again.

Any rd instruction can be replaced by the instruction, bi x, which reads the next n lines (where $C(AC) = n \cdot 2^{-15}$) from tape and stores them (in the form previously described) in a block of CM registers whose initial address is x.

The instruction

si 139 select PETR, word by word

behaves very much like the si 137, except that 3 lines on tape (instead of 1) are read and assembled in the IOR to form a 16 digit word as follows:



New words arrive in the IOR every 12 ms.

VIII. The Mechanical Tape Reader

The mechanical tape reader (MTR) behaves exactly like the PETR. We have the following si instructions:

si 128 select MTR, line by line

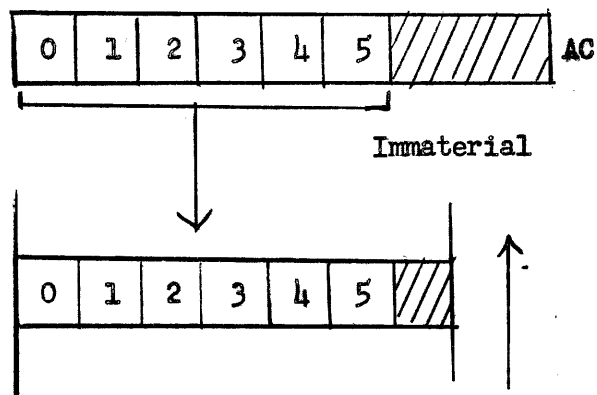
si 130 select MTR, word by word

Each rd instruction requires 105 ms in the line by line mode and 315 ms in the word by word mode. No in-out instruction can be performed during this time.

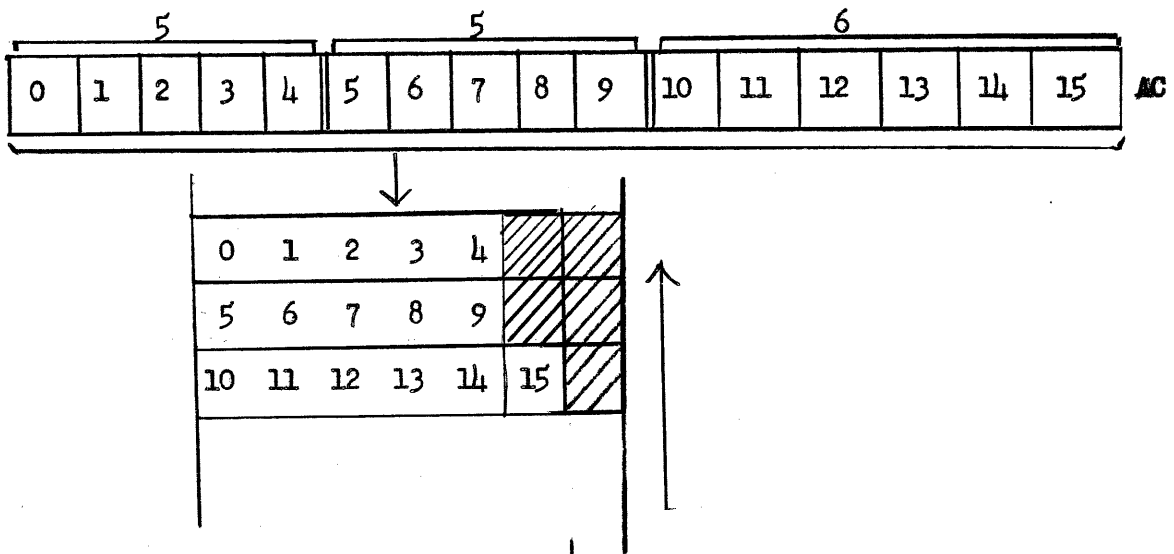
The MTR is not a free running device so that no timing problems arise in reading information. It is not necessary to deselect the MTR.

IX. The Mechanical Punch

Information to be punched may be recorded one line at a time from the left-hand six digits of the AC



or 3 lines at a time from the entire AC



Either form may be recorded with or without 7-th holes. The following si instructions are used:

si 132	select punch, line by line, no 7-th hole
si 133	select punch, line by line 7-th hole
si 134	select punch, word by word, no 7-th hole
si 135	select punch, word by word, 7-th hole

Each line (word) to be punched requires an rc instruction. Any number of rc instructions can be used after the si. Any rc instruction can be replaced by a bo instruction. Each rc instruction requires 80 ms in the line by line mode and 240 ms in the word by word mode. No in-out instruction can be performed during this time.

It is not necessary to deselect the punch.

X. The Delayed Printer and Punch

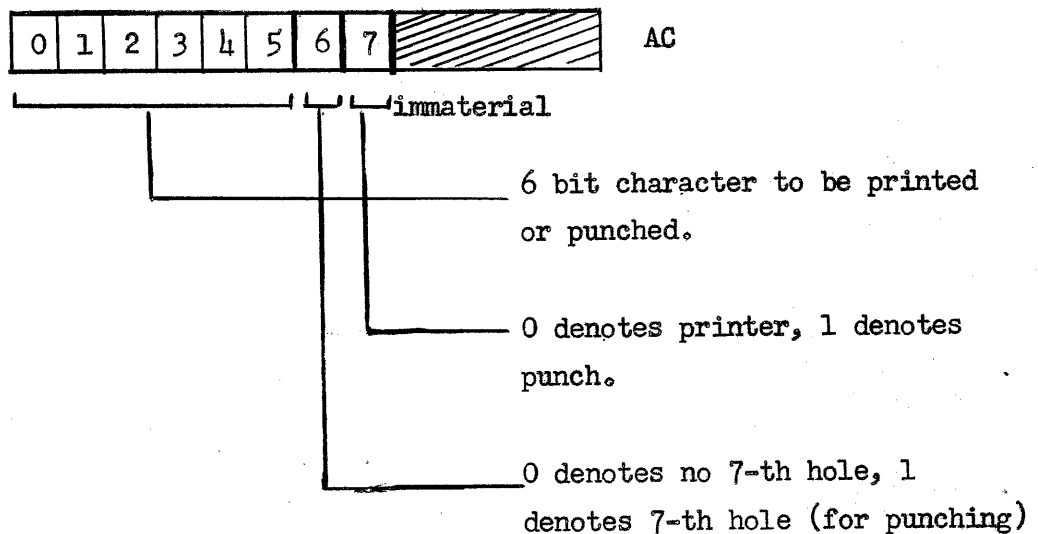
Coded information which can be used to control a flexowriter printer and punch can be recorded by the computer on MT units 2 and 3. The following si instructions are used to select the tape unit:

```

si 118    select MT2, record forward for delayed output.
si 126    select MT3, record forward for delayed output.

```

The instruction, rc, records C(AC) on the tape. The digits of the AC have the following meaning



An rc instruction must be given for each character to be printed or punched. Any number of rc instructions may follow the si. Each rc instruction requires 7.6 ms during which time no in-out instruction can be performed.

The code for the printer is exactly that described in the section on the flexowriter typewriter used for direct output from the computer.

Characters for delayed output can be recorded by the computer at the approximate rate of 133 per sec.

XI. Alarms using In-Out Equipment

Most of the difficulties in using in-out equipment are associated with the free running devices. Program alarms, inactivity alarms and parity alarms can be generated by using in-out devices illegally.

(1) If a piece of equipment is selected in the read mode and an rd instruction is given before the next word arrives in IOR, the computer must wait until a word arrives. Once a word has arrived in the IOR no new word can arrive from the unit until the IOR has been cleared (presumably by an rd instruction). If a new word arrives before this occurs, a program alarm is given by the computer. This occurs only if a free running unit is selected.

Another difficulty (not necessarily leading to an alarm) can occur here since the si instruction clears the IOR also. This means that a word can arrive in the IOR from a free-running unit and be lost if the next in-out instruction is an si (clearing IOR) and not an rd. Care must be taken to stop these units quickly enough if no more words are desired.

(2) A parity alarm will probably occur if an si instruction selecting the drum is given when C(AC) is an illegal drum address.

(3) si instructions with undefined addresses may cause the computer to stop on an inactivity alarm or may be ignored by the computer.

(4) An rc instruction after an si selecting a read mode or an rd instruction after an si selecting a record mode will probably lead to an inactivity alarm.

(5) Failing to follow a read si selecting the drum by a single rd (or bi) will probably lead to an inactivity alarm.

(6) The drums have a parity check on reading and recording similar to that generated in CM. An improper reading or recording on the drum can cause a parity alarm (in this case not due to the program but to the computer).

CHAPTER XIII: PROGRAMMED ARITHMETIC

- I. Introduction
- II. Interpretive Subroutines
- III. Generalized Decimal Numbers
 - A. (15, 15) Numbers
 - B. (30 - j, j) Numbers
 - C. (15, 0) and (30, 0)
 - D. (m, n) Control Words
 - E. Examples
- IV. (30 - j, j) Interpretive Subroutines
 - A. Instructions
 - B. Cycle Counter Instructions
 - C. Buffer Registers
- V. Automatic Compilation
- VI. Mistake Anticipation

I. Introduction

In this chapter we begin to discuss the programs written for WWI which form the CS computer.

The term, programmed arithmetic, is used to describe any systematic use of a set of routines for performing arithmetic and logical tasks which in general are not built into the computer hardware.

For example, computations using 15 binary digit numbers will not always yield results having sufficient accuracy. One possible remedy is to perform arithmetic using 30 digit numbers stored in two consecutive WWI registers. Since operations on this type of numbers are not directly available they must be programmed.

Two such numbers

$$A = a_1 2^{-15} + a_2 2^{-30} \quad 0 \leq |a_i| < 2^{15}$$

$$B = b_1 2^{-15} + b_2 2^{-30} \quad 0 \leq |b_i| < 2^{15}$$

whose sum

$$A + B = (a_1 + b_1) 2^{-15} + (a_2 + b_2) 2^{-30}$$

is less than unity can be added by the following program

```

p1, ca a2 }
sa b2 } (a2 + b2) → N (c2)
ts c2 }
p2, ca a1 }
ad b1 } (a1 + b1) → N (c1)
ts c1 }
STOP
a1, a1 } A
a2, a2 }
b1, b1 } B
b2, b2 }
c1, c1 } A + B
c2, c2 }

```

Any overflows which occur in forming, $a_2 + b_2$, are properly picked up as carries in forming, $a_1 + b_1$. Overflows which occur in forming, $a_1 + b_1$, stop the computer. ^⓪

^⓪ Find two pathological cases in which this could occur.

Programs forming $A=B$, AB and A/B can also be written and extended calculations performed on 30 digit numbers. Individual operations are performed by copying down these programs and specializing the addresses as needed.

However, computer time can be traded for space and convenience if each of the required operations is written (once) as a closed subroutine. Address specialization (via program parameters) is then performed by the subroutines.

If the operations are binary it is necessary to assign three program parameters, two giving the addresses of the operands in storage and one giving the address at which the result is to be placed.

Thus if a closed subroutine for addition is stored beginning at register, sl , the sum, $A + B$, is formed as follows

sp	sl	
$a1$		Address of operand, A.
$b1$		Address of operand, B.
$c1$		Address for result, C.

Operations using closed subroutines with a single program parameter may be used if a multiple register accumulator (MRA) is introduced.

An MRA for 30 digit numbers consists of two consecutive **WWI** registers which have the following relationship with a set of closed one-parameter subroutines:

(1) One of the operands associated with the subroutines is stored in the MRA.

(2) The result of the operation is stored in the MRA.

The single parameter of the subroutines is the address in storage of the remaining operand.

It is convenient to write closed one-parameter subroutines for performing the following logical operations:

(1) Copy the contents of the MRA into a specified storage location.

(2) Copy the contents of a specified storage location into the MRA.

Suppose that the following closed one-parameter subroutines have been written:

(1) sp s1 } al }	$N(a1) \rightarrow N(MRA)$
(2) sp s2 } al }	$N(a1) + N(MRA) \rightarrow N(MRA)$
(3) sp s3 } al }	$N(a1) \cdot N(MRA) \rightarrow N(MRA)$
(4) sp s4 } al }	$N(MRA) \rightarrow N(a1)$

Then letting $A = N(a1)$ and $B = N(a2)$, the following program stores $(A + B)A$, in cl.

p1, sp s1 } al }	A
p2, sp s2 } a2 }	A + B
p3, sp s3 } al }	(A + B)A
p4, sp s4 } cl }	(A + B)A \rightarrow N(cl)
p5, STOP	

II. Interpretive Subroutines

The preceding program forming $(A + B)A$ could also be carried out by executing the following program

r1, ca p1 } ts r3 }	Store first sp instruction of previous program in r3
r2, ca p1 + 1 } ts r4 }	Store its program parameter (address of operand) in r4
r3, + 0 } r4, + 0 }	Execute the operation
ao r2 } td r1 } ao r2 } sp r1 }	Set up to do the next operation

This technique is obviously inefficient since it replaces one sp instruction (initiating the operation) by several instructions (bringing

the initiating sp instruction and the program parameter to a fixed location). It illustrates an important point; that words can be used merely to indicate instructions which themselves now become parameters (in a sense) for a higher order subroutine.

One technique of this type has great practical importance since it again trades computer time for space by condensing the information required to specify an instruction into a single word. This is done as follows:

Each standard closed subroutine is assigned a numerical value, e.g.

0	$N(a1) \rightarrow N(MRA)$
1	$N(a1) + N(MRA) \rightarrow N(MRA)$
2	$N(a1) \circ N(MRA) \rightarrow N(MRA)$
3	$N(MRA) \rightarrow N(a1)$

An address parameter need only occupy 11 bits in a word. The remaining (left hand) 5 bits of the word are used for the operation parameter (allowing for 32 operations). The following program translates parameters into references to standard closed subroutines:

r1, ca p1 } td r4 }	Store address parameter (p1 contains the word representing the operation).
clh21	Bring operation number to right-hand 5 bits of AC.
ad r5 } td r2 } r2, ca - } ts r3 }	Form sp instruction referring to a standard closed subroutine
r3, + 0 } r4, + 0 }	Execute the instruction
ao r1 } sp r1 }	Set up to execute next instruction
r5, r6	
r6, sp s1 } sp s2 } sp s3 } sp s4 }	0 } 1 } 2 } 3 } Table of sp instructions which transfer control to the various standard closed subroutines

The preceding program is an example of a type of subroutine called an interpretive subroutine. The parameters decoded by interpretive subroutines are called interpreted instructions.

III. Generalized Decimal Numbers

We return now to the subject of exponential representations and describe how such representations are stored in the memory of the WWI computer. To begin with we define an (m, n) representative for $n \neq 0$.

An (m, n) representative of a number, $X \neq 0$, is a floating exponent representative of the number in which m binary digits are used to represent the mantissa and n binary digits are used to represent the exponent.

Only numbers in the range
 (3.1) $2^{-(2^n-1)} \cdot (1/2) \leq |X| \leq 2^{+(2^n-1)} \cdot (1 - 2^{-m})$
 can have (m, n) representatives.

Algebraic operations on (m, n) representatives will not in general yield (m, n) representatives. However if the result lies in the range (3.1) it can be approximated by an (m, n) representative. If the result lies outside the interval (3.1) it is called an overflow or an underflow according as its exponent is too large or too small.

An $(m, 0)$ representative of a number, X , where $|X| < 1$, is a fixed exponent representative in which m binary digits are used to represent the mantissa.

Only numbers in the range
 $-1 + 2^{-m} \leq X \leq +1 - 2^{-m}$
 can have $(m, 0)$ representatives.

The CS contains facilities for converting to binary the following kinds of numbers:

- (1) Integers.
- (2) Octal Fractions.
- (3) Generalized Decimal (GD) Numbers.

GD numbers are written in the following form

$$\pm r_m \dots r_1 r_0 \cdot r_{-1} \dots r_{-n} \times 2^p \times 10^q$$

GD numbers differ in appearance from integers and octal fractions in that they must contain both a printed sign and a decimal point. This, however, is all they need consist of, so that

$$\pm . = \pm . 0$$

is a GD number.

GD numbers can be converted into several kinds of (m,n) representatives (both floating exponent and fixed exponent representations can result). These are now described.

A. (15,15) Numbers

A (15,15) representative (of a GD number) has a 15 binary digit mantissa and a 15 binary digit exponent. By convention two consecutive WWI registers are used for the number and the mantissa is stored in the first register.

al	mantissa
al + 1	exponent

The one's complement notation is used for storing negative mantissas and exponents.

Only numbers lying in the range

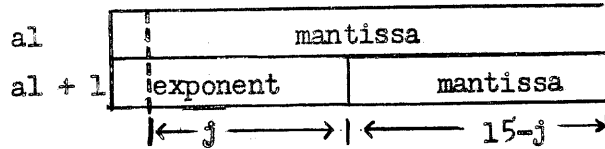
$$2^{-2^{15}} \leq |x| \leq 2^{+2^{15}-1} \cdot (1 - 2^{-15})$$

can have (15,15) representatives.

The (15,15) representative of the real number, 0, is defined to have the mantissa, 0, and the exponent, $-2^{15} + 1$.

B. (30 - j, j) Numbers

A (30 - j, j) representative (of a GD number), $j = 1, 2, \dots, 14$, has a 30 - j binary digit mantissa and a j digit exponent. By convention two consecutive WWI registers are used for the number and the 15 most significant digits of the mantissa are stored in the first register. The j digits of the exponent are stored in the left most j digits of the second register and the 15 - j least significant digits of the mantissa are stored in the right most 15 - j digits of the second register. The sign digit of the first register is the sign of the mantissa and the sign digit of the second register is the sign of the exponent.



The one's complement notation is used for storing negative mantissas and exponents.

The following convention is used for storing the minor part of the mantissa.

The magnitude of the minor part of the mantissa is stored in the second register if the exponent is positive, otherwise the one's complement of the magnitude is stored.

Only numbers lying in the range

$$2^{-2^j} \leq |x| \leq 2^{+2^j-1} \cdot (1 - 2^{-30+j})$$

can have $(30 - j, j)$ representatives.

The $(30 - j, j)$ representative of the real number, 0, is defined to have the mantissa, 0, and the exponent, $-2^j + 1$.

C. $(15, 0)$ and $(30, 0)$ numbers

A $(15, 0)$ representative (of a GD number) is a 15 binary digit number whose radix point is at the left-hand end of the number. Such numbers are stored in a single WWI register according to the one's complement notation. The WWI instruction code deals with $(15, 0)$ numbers.

A $(30, 0)$ representative (of a GD number) is a 30 binary digit number whose radix point is at the left-hand end of the number. By convention two consecutive WWI registers are used for the number and the 15 most significant digits are stored in the first register. The one's complement notation is used for storing negative numbers. Each half of the $(30, 0)$ number will have a sign digit but, by convention, these are assumed to agree.

D. The (m, n) Control Word

GD numbers can be converted to any of the (m, n) representations previously described. The representation obtained depends on the previous appearance in the manuscript of control words of the form, (m, n) . The following rules apply:

(1) If no previous (m, n) control word has occurred, a GD number is converted to a $(15, 0)$ representative.

(2) If an (m, n) control word occurs, then all GD numbers following it are converted to (m, n) representatives until another (m, n) control word occurs.

For example:

fc 100-0-0		
+ .6792	$+1.23 \times 10^{-2}$	(15, 0) Numbers
(30, 0)		
+ .77	$+0.23 \times 2^{-5}$	(30, 0) Numbers
(23, 7)		
+12.34	$-0.0036 \times 10^{+5}$	(23, 7) Numbers

If a GD number exceeds the range of the representation, the translation process is stopped. For example

fc 100-0-0
(30, 0)
+1.234

E. Examples of (m, n) Numbers

(1) (15, 0)

Decimal	+ .12345	
Octal	+ .07715	
Register		x) 0.07715

(2) (30, 0)

Decimal	+ .123456789	
Octal	+ .07715 33515	
Register		x) 0.07715
		x+1) 0.33515

(3) (15, 15)

Decimal	$+1234.5 = +.12345 \times 10^4 = +.60278 \times 2^{11}$	
Octal	$+ .56451 \times 2^{13}$	
Register		x) 0.56451
		x+1) 0.00013

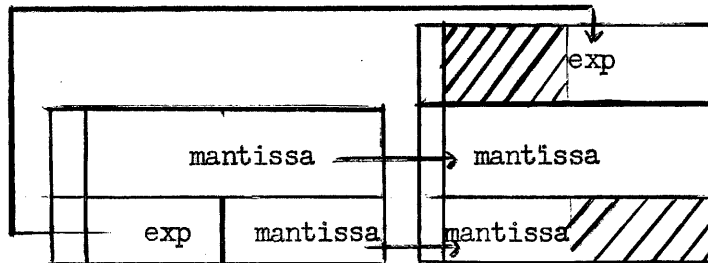
(4) (24, 6)

Decimal	$+1234.567 = +.1234567 \times 10^4 = +.6028159 \times 2^{11}$	
Octal	$+ .56451 045 \times 2^{13}$	
Register		x) 0.56451
		x+1) 0.13 045

IV. The (30 - j, j) Interpretive Subroutine

Operations on (m, n) numbers other than (15,0) numbers must be programmed. An interpretive subroutine which performs operations on (m, n) numbers is called an (m, n) interpretive subroutine. A (30 - j, j) interpretive subroutine, for j = 1, 2 .., 14, can be automatically compiled during the translation process if the coder so specifies.

The (30-j, j) interpretive subroutine performs operations on (30 - j, j) numbers. In coding operations on these numbers it is convenient to first unpackage the numbers to a three register form in which the mantissa occupies two full registers and the exponent one full register. This is illustrated below.



The additional digits required for the mantissa (exponent) in the three register form are all zeros or ones according as the mantissa (exponent) is positive or negative. Three-register numbers of this form described above are called (30, 15) numbers.

Accordingly the MRA in the (30 - j, j) interpretive subroutine contains a (30, 15) number. Operations requiring that a number in the MRA be placed in storage always form a packaged (30 - j, j) approximation to the number in the MRA. The addresses (in storage) of the MRA are the following

2042	Exponent
2043	Mantissa (major part)
2044	Mantissa (minor part)

A. The Interpreted Instructions

The interpreted instructions used by the (30 - j, j) interpreted subroutine are written in the form

i x y

where xy denotes a pair of letters which frequently correspond to an analogous WWI instruction.

1. The Instructions, ica al and ics al.

The interpreted instruction, ica al, unpackages the $(30 - j, j)$ number in registers al and al + 1 to a $(30, 15)$ number and copies the $(30, 15)$ number into the MRA.

The interpreted instruction, ics al, complements the mantissa of the $(30, 15)$ number before copying it into the MRA.

No alarm can occur on either of these instructions.

2. The Instruction, its al

The instruction, its al, forms a $(30 - j, j)$ approximation to the $(30, 15)$ number in the MRA as follows:

Let \bar{x} denote the number obtained by rounding off the 30 digit mantissa in the MRA to $30 - j$ digits. Define x^* by

$$x^* = \frac{1}{2} \bar{x} \quad \text{if } |\bar{x}| = 1$$

$$x^* = \bar{x} \quad \text{if } |\bar{x}| < 1$$

Let y denote the exponent in the MRA. Define y^* by

$$y^* = y + 1 \quad \text{if } |\bar{x}| = 1$$

$$y^* = y \quad \text{if } |\bar{x}| < 1$$

If

$$-2^j + 1 \leq y^* \leq 2^j - 1$$

the $(30 - j, j)$ approximation is defined to be (x^*, y^*) .

If

$$y^* \leq -2^j$$

the $(30 - j, j)$ approximation is defined to be $(x^*, -2^j + 1)$.

If

$$y^* \geq 2^j$$

the number has no $(30 - j, j)$ approximation and an alarm is generated (the deliberate generation of a computer alarm is referred to as mistake anticipation).

The resulting $(30 - j, j)$ approximation is then copied into registers al and al + 1.

The instruction, its, is an inverse to the instruction, ica, in the sense that the sequence

ica al

its al

leaves unchanged the contents of al and al + 1.

3. The Instruction, iex al

The instruction, iex al, forms the $(30 - j, j)$ approximation to the number in the MRA (exactly as described for the its operation), exchanges it with the $(30 - j, j)$ number in registers al and al + 1, un-packages the latter number to a $(30, 15)$ number and copies it into the MRA (exactly as described for the ica operation).

4. The Instructions, iad al and isu al

The instruction, iad al, first un-packages the $(30 - j, j)$ number in registers al and al + 1 to a $(30, 15)$ number. An alarm is generated (mistake anticipation) if the major part of its mantissa lies in the range, $0 < |x| < 1/2$. We denote the unpackaged number by

$$Y \cdot 2^y$$

and the number in the MRA by

$$X \cdot 2^x$$

The exponents, x and y, are compared and the number with the larger exponent is placed in the MRA.

If $|x - y| > 29$ the numbers are called incommensurable and the sum of the numbers is defined to be the number with the larger exponent (already in the MRA).

If $|x - y| \leq 29$, the following sum is formed

$$\bar{Z} = X + Y \cdot 2^{-|x-y|}$$

and scale factored to yield

$$\bar{Z} = Z \cdot 2^p$$

where either

$$1/2 \leq |\bar{Z}| < 1 \quad \text{and} \quad -32 \leq p \leq 1$$

(if $|\bar{Z}| \neq 0$) or

$$|\bar{Z}| = 0 \quad \text{and} \quad p = -33$$

(if $|\bar{Z}| = 0$). The sum of the two numbers

$$Z \cdot 2^{p+x}$$

is copied into the MRA.

The routine yields correct results even when the number in the MRA has a non-scale-factored mantissa. The sum of two numbers has a scale-factored mantissa except for the case where the summand in the MRA was

non-scale-factored and incommensurable (with larger exponent) with the summand from storage.

Difficulties can arise when one of the summands is a zero. Since the routine does not distinguish zeros the sum of a zero and a non-zero number can easily be the zero (if its exponent is much larger).

An overflow (which is not anticipated by the routine) can occur in forming the exponent of the sum, $p + x$, but the possibility is remote.

The instruction, `isu al`, behaves exactly like the instruction, `iad al`, except that the mantissa of the summand from storage is complemented before entering the routine.

The mantissa of the sum (or difference) is accurate to at least 28 binary digits.

5. The Instruction, `imr al`

The instruction, `imr al`, first unpacks the $(30 - j, j)$ number in registers `al` and `al + 1` to a $(30, 15)$ number. An alarm (mistake anticipation) is generated if the major part of the mantissa lies in the range, $0 < |x| < 1/2$. We denote the unpackaged number by

$$Y \cdot 2^Y$$

and the number in the MRA by

$$X \cdot 2^X$$

A two register approximation, \bar{Z} to the four register product, XY , is formed and scale-factored to yield

$$\bar{Z} = Z \cdot 2^P$$

where either

$$1/2 \leq |Z| < 1 \quad \text{and} \quad -32 \leq p \leq 0$$

(if $|\bar{Z}| \neq 0$) or

$$|Z| = 0 \quad \text{and} \quad p = -33$$

(if $|\bar{Z}| = 0$). The product of the two numbers

$$Z \cdot 2^{P + X + Y}$$

is copied into the MRA

An overflow alarm (which is not anticipated by the routine) can occur in forming the exponent of the product, $p + x + y$.

The mantissa of the product is accurate to at least 28 binary digits.

6. The Instruction, idv al

The instruction, idv al, first unpackages the $(30 - j, j)$ number in registers al and al + 1 to a $(30, 15)$ number. An alarm is generated (mistake anticipation) if the major part of the mantissa lies in the range, $0 < |X| < 1/2$. We denote the unpackaged number by

$$Y \cdot 2^Y$$

and the number in the MRA by

$$X \cdot 2^X$$

A two register inverse, $\frac{1}{4} Y^{-1}$, of $4Y$ is formed and the multiplication routine is used to multiply.

$$\frac{1}{4} Y^{-1} \cdot 2^{-Y+2} \quad \text{and} \quad X \cdot 2^X$$

An overflow alarm (which is not anticipated by the routine) can occur in forming the exponent of the quotient, $x - y + 2$. A divide error alarm (also not anticipated) will occur if register al contains a zero.

The mantissa of the quotient is accurate to at least 24 binary digits.

7. The Instructions, isp al, icp al and sp al

The execution of an interpreted instruction is in general initiated by the following WWI instructions

2018 | ao 2019

2019 | ca -

Register 2019 is called the interpreted program counter (IPC) since it contains the address of the interpreted instruction being performed.

The instruction

bl, isp al

begins by storing the WWI instruction, ca bl + 1, in a register (whose address is 2040) called the interpreted A-register (IAR). The previous contents of the IAR is stored in the initial register (whose address is 2025) of a four-register table called the jump table. Previous entries in the jump table are shifted downward. The IAR and jump table thus list the addresses of the last five isp instructions performed.

The WWI instruction, ca al, is then placed in the IPC and WWI computer control transferred to the IPC.

The instruction

bl, icp al

behaves exactly like the instruction, `isp al`, if the number in the MRA is negative, and is otherwise ignored.

The instruction

`bl, sp al`

causes WWI computer control to be transferred to register `bl` from which computer control passes to register `al`.

8. The Instructions, IN and OUT

The WWI instruction

`al, IN`

is translated into the instruction, `sp 2046`, and transfers computer control to a two register program

`2046 ta 2019`

`2047 sp 2019`

Since `2019` is the IPC this causes the interpretive subroutine to interpret the instruction contained in register, `al + 1`. An `IN` which is executed by the interpretive instruction is ignored (see the instruction, `sp al`).

The interpreted instruction

`al, OUT`

is translated into the instruction, `sp al + 1`, and is a special case of the instruction, `sp al`. An `OUT` which is executed as a WWI instruction is ignored.

B. The Cycle Counting Instructions

The single letter, `@` may be used as a suffix for any of the interpreted instructions

`its al, iex al, ica al, ics al,`

`iad al, isu al, imr al, idv al,`

`isp al.`

and has the effect of subtracting a fixed amount (`+20`) from whatever is the value of the corresponding interpreted instruction.

Let `ixy` denote any of the above instructions except `isp`.

In executing the instruction

`ixy al+c`

the interpretive instruction first forms the address

$W = (al + 2i) \text{ mod } 2048$

where i denotes the contents of the index register being used, and then executes the instruction, $ixy W$. In the case of the isp instruction the quantity, W , defined above is replaced by

$$\bar{W} = (a_1 + i) \text{ mod } 2048$$

Overflow alarms (not anticipated) can occur in forming W (or \bar{W}) and in executing the instructions ici , icd , ict or iat . In general these occur when the contents of an index register becomes excessive.

1. The Instruction, $isc n$

The instruction, $isc n$, makes it possible to use multiple pairs of index and criterion registers. The integer, n , can assume the values, $n = 0, 1, 2, \dots$, and the corresponding index and criterion registers (numbered accordingly) are assigned consecutive locations in a storage block which is allocated during the translation process.

c_1	i_0
c_1+1	n_0
c_1+2	i_1
c_1+3	n_1

The length of the block depends on the maximum value of n appearing as the address of an isc instruction in the printed manuscript. The location of the block will vary, but the address of the 0 index register (i.e. c_1) always appears in register 2045.

A check alarm is generated (mistake anticipation) if the interpretive subroutine attempts to execute an $isc n$ instruction for which no index and criterion register has been allocated in the above table.

C. The Buffer Registers

Some (30, 15) storage locations (called buffer registers) have been defined which can be used as operands for interpreted instructions. These are denoted by the symbolic addresses

$$b, 1b, 2b, \dots, 263b$$

and can be used as addresses for the instructions

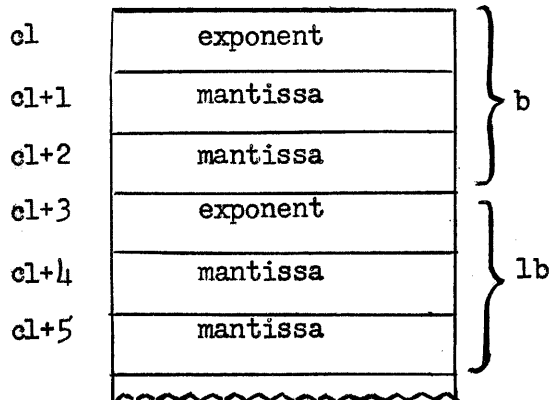
$$its, iex, ica, ics, iad, isu, imr, \text{ and } idv$$

When such an instruction is performed both operands are (30, 15) numbers.

The single letter, b , is assigned the value, 1784, during the

translation process so that the address, nb , becomes $n + 1784$. Addresses denoting buffers are larger than ordinary addresses in a program.

The buffer registers are exact images of the MRA and are assigned consecutive locations in a storage block which is allocated during the translation process.



The length of the block depends on the maximum value which the integer, n , assumes in addresses of the form, nb , appearing in the printed manuscript. The location of the block will vary, but the address of the exponent of the 0 buffer always appears in register 2039.

V. Automatic Compilation

A $(30 - j, j)$ interpretive subroutine is automatically compiled during the translation process in response to the appearance of certain words in the printed manuscript. The rules are explained in the following section:

A $(30 - j, j)$ interpretive subroutine will be compiled if and only if the following conditions are satisfied.

1. The last (m, n) control word in the manuscript is of the form, $(30 - j, j)$, where $j = 1, 2, \dots, 14$.

2. An interpreted instruction, $ixy a_1$, or an i START AT appears in the manuscript.

3. The word, NOTPA, does not appear in the manuscript.

The satisfying of these conditions guarantees the inclusion of a minimal (30 - j, j) interpretive subroutine which performs the ordinary arithmetic and logical instructions not involving cycle counters or buffers.

Further specialization of the (30 - j, j) interpretive subroutine can be accomplished. This has been made possible by breaking up the interpretive subroutine into four program blocks.

1. The Basic Program Block: is described above.
2. The Cycle Counting Program Block: executes instructions suffixed by c and instructions affecting the cycle counters. This block is included whenever an instruction of the form, ixyalc, or one of the instructions, isc, icr, ict, iat, or iti, appears in the manuscript.
3. The Buffer Program Block: executes instructions whose addresses refer to buffers. This block is included whenever an instruction of the form, ixynb, appears in the manuscript.
4. The isc Program Block: executes the isc instruction. This block is included whenever an isc instruction having a non-zero address appears in the manuscript.

The lengths of the various blocks are given in the following table.

Basic	266
Cycle Counting	40
Buffer	74
isc	23

In addition to the program blocks the following blocks of storage are allocated during translation:

1. The cycle counters
2. The buffers

The lengths of these blocks has already been discussed.

The total length of the (30 - j, j) interpretive subroutine compiled can be obtained by adding all of the block lengths listed above (noting that the presence of the cycle counting block without the isc block requires a single cycle counting register pair).

The spatial arrangement of the blocks in storage is the following:

Buffers
Cycle Counters
isc Block
Buffer Block
Cycle Counting Block
Basic Block

2047

VI. Mistake Anticipation

Mistake anticipation refers to the programmed detection of illegal situations. Those illegal situations anticipated by the (30 - j, j), interpretive subroutine have been described in the previous sections.

With anticipated alarms care has been taken not to disturb pertinent temporary registers (MRA, IAR, IPC) in the interpretive subroutine until it is certain that the operation can proceed correctly. This is not the case with non-anticipated alarms.

Thus a programmed arithmetic post-mortem (PAPM) taken after an anticipated alarm yields accurate results while a PAPM taken after a non-anticipated alarm may present a jumbled picture of the interpreted subroutine.

Anticipated alarms cause the computer to stop on a check alarm in register 2036.

CHAPTER XIV: THE CONVERSION PROGRAM

- I. Introduction
- II. The Operation of the Conversion Program
 - A. The Use of the Computer Memory
 - B. The Address Indicators
- III. The Vocabulary of the CS Coding Language
 - A. Characters
 - B. Words
 - C. Polysyllabic Words
 - 1. Syllables
 - a) Constant Syllables
 - b) Parametric Syllables
 - c) Special Syllables
 - 2. Control Words
 - a) Floating Address Tags
 - b) Current Address Assignments
 - c) Drum Address Assignments
 - d) Starting Address Assignments
 - e) Preset Parameter Assignments
 - f) Temporary Address Assignments
 - g) Relative Address Assignments
 - 3. Single Letters
 - a) Buffer Registers
 - b) Cycle Counters
 - c) Relative Addresses
 - d) Temporary Storage
 - D. Special Words
 - 1. Storage Words
 - a) Generalized Decimal Numbers
 - b) Input and Output Subroutine Requests
 - c) PA Entry and Exit Words
 - d) Stop Instructions
 - 2. Control Words
 - a) Titles

- b) Integer Base Indicators
- c) Cancellation of PA Requests
- d) Library Subroutine Indicators
- e) Number System Indicators
- f) Comment Words
- g) DITTO

IV. Error Detection During the Conversion Process

I. Introduction

The primary concern of a coder is to express a program in a language acceptable to the computer. The characters, or symbols, of this language must be those which are accepted by the computer input devices, and the syntax, or grammar, of the language must permit a simple translation of the coded program into the equivalent machine-coded statement of the problem in the computer memory. The simplest language from the translation point of view is the one which is essentially an exact portrayal of the eventual machine-code. In this case the coder writes machine-code. For example, with Whirlwind this would correspond to coding with binary representations of the machine words and, as a matter of fact, the earliest input translation programs for Whirlwind did no translating and merely accepted binary-coded, hand-prepared tapes.

The next stage beyond this is the use of octal or sexadecimal representations of groups of the binary digits, or decimal characters in the case of decimal machines, to reduce the number of characters that must be manipulated and written by the coder. However, languages using such characters still require knowledge by the coder of the exact representations in the computer memory of the instructions he wishes to write. An essentially different approach is taken, though, if the coder writes in a language whose elements are abbreviated functional descriptions of the operational facilities of the computer, i.e., the operations and operands used during computation. Such a language, which may bear very little relationship to the actual appearance of these words in the computer memory, is called an alphanumerical, mnemonic code.

The coding language of the Comprehensive System provides such a code. It has elements which denote the functional operations, etc., with which the coder would prefer to be concerned while initially coding a problem, but it also contains elements which allow the coder to denote exactly what is to be the contents of particular registers of the computer memory in case he feels the need for such coding. Abbreviations such as "ca" are used to designate the operation of clearing the Whirlwind accumulator and adding to it the contents of a register. The location of the register can be designated by a floating address tag which may also bear some mnemonic relation to the variable designated. Other

facilities, like cycle counting and buffer storage in the interpreted computers, again use mnemonic designators. In no case need the coder unnecessarily concern himself with other than the functional structure of the program he is coding. In addition he has at his disposal various pseudo-operations which effectively enlarge the number of facilities available.

The conversion program of the Comprehensive System accepts the sequence of alphanumeric characters comprising the code for a program and translates this into the equivalent machine-code in Whirlwind. In this chapter the CS language will be completely specified, and some of the operational aspects of the system will be described.

II. The Operation of the Comprehensive System

A. The Mechanics of the System

A program is introduced into the Whirlwind computer via the Comprehensive System by punching the sequence of characters corresponding to the coded program into seven-channel paper tape with a Flexowriter typewriter-punch. This tape is placed in the photo-electric paper tape input device of Whirlwind and the read-in button is pushed. The programs described in Chapter XV then transfer the contents of core memory to auxiliary drum group #0 and bring the conversion program into the core memory of Whirlwind. This program takes over the read-in and translation process.

The conversion program consists of several smaller programs which perform their functions in sequence. First, the title program reads the title punched on the Flexo tape and records this on various output devices for identification purposes (see Chapter XV on this also). In particular, the title is normally recorded on Magnetic Tape #3 to identify program results which might appear there. The next program, called the first pass program, then reads in the remainder of the paper tape, translates it partially, and records the partially-translated information as a sequence of blocks on Magnetic Tape #1. The information on MT#1 is called "logical" information to distinguish it from the untranslated form on paper tape and the completely-translated end product. Tables of information pertaining to the kinds of floating addresses and preset parameters used and the automatic interpretive routines desired

are also compiled during the first pass and are left in core memory.

The second pass program reads the information on MT#1 backwards and tabulates in core memory the values of all the floating address tags and all the automatic output requests. Control is then transferred to a set of programs which compile all the interpretive and output routines desired and store these at the proper locations on auxiliary drum group #0 corresponding to the eventual core memory addresses.

The third pass program then reads the information on MT#1 in the forward direction and completes the translation process, placing the completely translated words on the auxiliary drum. Any words written on drum group #0, of course, replace those initially copied there from core memory. The last program then records the values of all the floating address tags on MT#3 and transfers control back to a program which copies auxiliary drum group #0 into core memory and stops the computer in such a manner that pushing the Restart button will transfer control to the address indicated by the START AT at the end of the Flexo tape.

Several things need to be said about the conversion process. First, the computer stops after all of the Flexo tape in the reader is read in. If the Restart button is pushed, then the conversion proceeds as above. But if this Flexo tape does not contain all of the information that is to be translated at this time, as, for example, when the program tape is broken into two or more sections, then pushing the Start at 40 button will cause the first pass program to read in more Flexo tape from the tape reader until the next START AT word occurs. This process may be repeated as many times as desired until all tapes are read in. Pushing the Restart button will then initiate the second pass program. Several tapes can thus be converted together as if they were one tape.

The computer operator can also cause the conversion program to record on MT#3 a copy of the converted program as it would appear in computer memory in binary form. This process is controlled by pushing other buttons on the control console. The information on MT#3 can be punched out on paper tape to give a 556 copy of the program. This may subsequently be read back into the computer without making use of the conversion program (see Chapter XV for a description of the structure of this

556 tape.

The conversion process will terminate at any stage if the conversion program detects one of several kinds of error in the coded program. These errors are described in a later section. An indication of these errors is printed on the direct output typewriter and also on MT#3.

B. The Address Indicators

A program which has been coded and punched on tape has a sequential structure which reflects the sequential, single-address structure of the machine code in the Whirlwind computer. If this sequence is broken or otherwise disturbed, then the program in computer storage will not operate in the expected manner. However, not only must the sequence of words be preserved, but also the values used in the address sections of the instruction words must be in agreement with the actual location of the designated operands.

Coders who use numerical addresses throughout a program pose no problems in this respect for an input translation program, but there are two kinds of difficulty which do arise in other situations. Consider first the case where a program is coded wholly or in part with floating addresses. In this case the translation program must do some computation to find the value of the floating addresses used in the program and insert the correct numerical values where desired into the address sections of instructions. The mechanics of this computation is rather simple: the translation program merely notes the location assigned to the initial word of a block of words and sets a counter to this value. Then, as successive words which will occupy storage locations occur, the translation program merely indexes this counter. Whenever a floating address occurs tagging a word, the translation program assigns the current value of this counter to the floating address. This evaluation takes place during the first pass of the translation process, and the counter described is called the current address indicator. It should be noted that this indicator is set to a new value only when a current address assignment occurs, and is indexed once or twice whenever a word occupying storage registers occurs, depending upon the number of registers occupied by the word.

A second problem arises, though, when a coder decides that not

all of his program will be stored in the random access memory of the computer during the read-in and translation process. This, for example, occurs when the program consists of a large iterative loop, not all of which can fit into core memory at once but which must be entirely within the computer at all times if the solution is to be computed in a reasonable time. Here the problem is that those parts of the program which are to be stored in auxiliary memory must be assigned locations during the read-in process which may bear little or no relationship to the actual locations in core memory when being executed.

In Whirlwind this auxiliary memory is the auxiliary drum. The registers in the core memory are addressed from 32 to 2047. A natural manner of addressing the registers on the drum so that words can be assigned locations there is to consider the drum as a non-random access extension of core memory. The system used is to assign the addresses 0 to 24575 to the registers of the drum. Registers 32 to 2047 are used to store the contents of core memory during the read-in process and are normally identified with core memory. Registers 22528 to 24575 are used to store the utility control program and hence are also not available for use by programmers. However, the entire range is covered by another address counter, called the drum address indicator, which enables a coder to assign words to any location in either core memory or the drum. Programs or data stored on the drum during read-in can be brought into core memory during program operation by proper use of the drum in-out instructions (see Chapter XII).

The exact rules by which the conversion program determines the locations of words and the values of floating address tags using these two indicators are as follows: the value given to a floating address tag is based entirely upon the value of the current address indicator when the tag occurs, whereas the location assigned to words in storage is exactly the value of the drum address indicator when the words occur. Both indicators are given the initial value of 32 (decimal) by the conversion program before any Flexo tape is used. Whenever a current address assignment causes the current address indicator to be increased or decreased by a certain amount, then the drum address indicator will be increased or decreased by that same amount.

Hence, if a program fits entirely into core memory, there will be no need for the coder to be concerned with the drum address indicator. However, if a program is to be stored on the drum, then use must be made of drum address assignments. Whenever one of these assignments occurs, the conversion program gives the indicated value to the drum address indicator without affecting the current address indicator. Thus it is possible to have words stored in one location while associated floating address tags receive different values.

An exception will now be described: whenever a current address assignment immediately follows a drum address assignment, then the drum address indicator is not affected by the current address assignment. This permits a coder to assign explicit values to both indicators. Definitions and examples of the use of the two kinds of assignments are given in the section on polysyllabic words.

III. The Vocabulary of the Comprehensive System Coding Language

A complete listing of the elements of the CS coding language is given in Figures 1 and 2 at the end of this chapter. Constant reference should be made to them during the following description.

A. Characters

The coder has available the symbology on the keyboards of the Flexowriter typewriter-punches used at the Digital Computer Laboratory. Depressing a key causes a unique six binary digit character to be punched on paper tape and the tape to be advanced one position. This tape is the medium for inserting coded programs into the computer. Only those six bit characters corresponding to keyboard characters are accepted by the conversion program, and only meaningful sequences of characters should be written or punched. The conversion program detects and indicates all illegal characters (see Figure 1); however, not all meaningless sequences are detected. For a description of the detected ones, see Section IV.

All accepted characters have a unique meaning to the conversion program except for the letters o and l. These are considered to be the same as the digits 0 and 1. Upper case characters are considered distinct from lower case characters even though the same codes are punched on the paper tape. The shifts to upper or lower case correspond to actual characters punched on the paper tape and are used to distinguish upper and lower

case characters. Any number of the shift characters may occur at any place on the tape so long as the print of the resulting tape has the correct appearance. That is, superfluous shift characters are ignored. Tab and carriage return characters may also be inserted where desired between words to control the print format.

B. Words

In the CS language nearly every word in the coded program corresponds either to a computer instruction or to a data word and will occupy registers in the storage element of the computer after translation. There are other words, however, which do not in themselves occupy storage registers but are used instead to influence the form and location of the storage words. These two kinds of words are classified by calling them, respectively, storage words, and control words.

A different classification can be obtained by considering the structure of the words. Some, for example, like

ca47+h3-zb2

or

243+r |

are composed of more primitive elements which usually consist of more than one character. The sixteen bit value of the translated word is essentially the sum of the binary values of these elements. These elements are called syllables, and such words, polysyllabic words. On the other hand, there are control words like

DITTO

and storage words like

iMOA+1.234567s

which have a specific meaning or function in the translated program but not a polysyllabic structure. These are called special words.

Thus we have four classes of words: polysyllabic storage words, polysyllabic control words, special storage words, and special control words. All words in the CS language are classified in this manner, and the description of the language is built around this system of classification.

C. Polysyllabic Words

A polysyllabic word is a sequence of syllables followed by a terminating character. Each syllable is detected by the conversion program and, unless it is a special syllable (see l.c below), is given a sixteen bit value. These values are summed to give the value of the converted word. The summation does not take place in the sequence the syllables occur, but all the constant syllables are summed first and the parametric syllables are added later. However, the letter r is treated in the special manner described in the section on relative addresses. The value of a syllable is added or subtracted depending upon whether the syllable is preceded by a plus sign (explicit or implicit) or a minus sign.

The conversion program detects the end of a word by the occurrence of a terminating character. The terminating character determines the function of the word terminated, as well, perhaps, as that of the following word. There are only four such characters: the tab or carriage return, the comma, the equals sign, and the vertical bar. Of course, the syllables occurring in a word also influence the meaning of the word.

A polysyllabic word terminated by a tab or carriage return is always a storage word unless the preceding word is terminated by an equals sign or unless this word contains the special syllable START AT. On the other hand, any word terminated by a comma, equals sign, or vertical bar is a control word. The sequence of the syllables should have no effect upon either the meaning of the word or the binary value of the word. However, the manner of summation of the values of syllables occurring in the word can influence the binary value of the word. The "special add" operation is used in combining the syllables (see Chapter XI), and any overflow that may occur during the formation of the intermediate partial sums is forgotten. Since the constant syllables are summed before the parametric syllables, it is possible for such an overflow to occur without the coder being immediately aware of the fact. Fortunately, however, the only time that this situation will normally occur is when a coder uses negative syllables in a Whirlwind instruction containing the operation code "ca", as described in Chapter VIII.

1. The Syllables

The syllables are classified as constant, parametric, and special.

a) The Constant Syllables

The constant syllables are so called since their binary values are fixed and do not depend upon the other words in the program.

1) The Whirlwind Operations

These syllables are written as two or three lower case letters and are listed, with their binary values, at the end of Chapter XI. Usually only one of them appears in an instruction word, but actually as many as desired may be used.

2) The Interpreted Operations

These are always written as three lower case letters with the first letter an "i". They are given in Chapter XIII.

3) The Integers

These are any of the integers from 0 to $2^{15}-1$ and are written with or without sign, but never with a radix point. They are assumed to be decimal unless otherwise indicated by use of the appropriate special words (see the section on the integer-base indicators).

4) The Octal Fractions

Sometimes it is desirable to specify the exact binary value of a word, or part of a word; the octal fractions permit this. They are written as a one or a zero, followed by a point and exactly five octal digits. This amount of information is equivalent to a sixteen bit Whirlwind word. Octal fractions need not be the only syllable in a word, and the only precaution which must be observed by a coder is that an octal fraction appearing as the first syllable in a word must not be preceded by a plus or minus sign. If this rule is broken, then the word will be considered to be a generalized decimal number.

5) The Single Letters

Two syllables, b and c, consist of a single letter and have a fixed binary value. These are discussed further in the section on single letters.

b) The Parametric Syllables

The parametric syllables are given binary values which depend upon an implicit or explicit assignment given by the coder. The rules describing assignments are given in later sections of this chapter.

1) The Floating Addresses

A floating address syllable consists of a single letter followed by a decimal integer from 1 to 255. Any letters except o or l may be used. The binary value assigned to a floating address is determined by the conversion program and depends upon how the floating address is used as a tag by the coder. This value is substituted for the syllable whenever the conversion program detects the floating address in other words.

2) The Preset Parameters

A preset parameter syllable consists of two letters followed by a decimal integer from 1 to 40. The first of the two letters must be a p, u, or z, while the second can be any letter except o or l. The binary value assigned to a preset parameter is defined explicitly by the coder. This value is then substituted in all subsequent uses of the parameter until the parameter is redefined.

3) The Single Letters

The two syllables r and t are given binary values which depend upon their use in the program being converted. They are discussed at length in the section on single letters.

c) The Special Syllables

Two syllables are unusual in the sense that they have no corresponding binary values, but are instead used in words to determine their control functions. Both consist of upper case letters and are normally written as the first syllable of the words in which they appear.

1) The Drum Address Syllable

This syllable consists of the upper case letters DA, and is used to distinguish a drum address assignment from a current address assignment.

2) The Starting Address Syllable

This is written as START AT, though only the first three letters need actually be written. It must be preceded by a lower case i if the initial instruction executed in the program is an interpreted instruction.

2. The Polysyllabic Control Words

Any polysyllabic word terminated by a comma, equals sign, or vertical bar, or containing one of the special syllables is a polysyllabic control word. The seven different kinds of such words are described below. These words are usually restricted in the kind of syllables which they may contain.

a) Floating Address Tags

A floating address is assigned a value by writing it as a syllable in a polysyllabic word which is terminated by a comma and which appears just before the word being tagged. Only integer syllables may appear in the tag word in addition to the floating address itself. Thus, a coder may write

$$al, -79$$

thereby tagging the location of the word -79 by the floating address al. The conversion program actually gives the value of the current address indicator to al when the al, occurs. On the other hand, he may also write

$$5+al, -79$$

In this case the conversion program subtracts 5 from the current address indicator and gives the resulting value to the floating address al. In other words, the conversion program always subtracts the positive or negative integers which may be present in the tag word from the current address indicator in order to find the value of the floating address.

The number of floating addresses which may appear in a program is limited. If no input-output special words are used in the program, then the sum of the maximum integers appearing for each of the letters used for floating addresses must be less than 256. Hence, if only the floating addresses w1, w10, and r245 are used, then we have the inequality

$$10+245 < 256$$

satisfied. On the other hand, if w11 or r246 or b1 were also used, then

we would not be able to satisfy the inequality. Each input-output special word which appears reduces the number of floating addresses which may be used by one, so, for example, if ten such words appear, then the above sum can be at most 245. It is important that all floating addresses appearing in a program should be considered in determining the maximum integer for each letter and not just those floating addresses which appear in tag words.

The current address indicator may at times be in the indefinite status described below. No floating addresses may be defined then. Violation of either this rule or the one above will cause an error indication by the conversion program.

b) Current Address Assignments

A coder may wish to specify the initial address of a program in core memory, or the end of a block of registers in which a word is being DITTO'd (see section III, D). In these cases he must make a current address assignment. This is done by writing a polysyllabic word, terminated by a vertical bar, in which no special syllables appear. Such a word will alter the current address indicator and drum address indicator as described in section II, B.

Hence, if a coder writes

$$132|$$

or

$$a1+10|$$

then the current address indicator will be given the values, respectively, of 132 or a1+10. Suppose, now, that the previous word is not a drum address assignment and that the current address indicator had the value 32 when the current address assignment 132| occurred. Then the effect of the current address assignment is to increase both the drum address indicator and the current address indicator by 100. On the other hand, if the previous word is a drum address assignment, then only the current address indicator is altered.

If any parametric syllables other than r appear in a current address assignment, then the current address indicator is placed in an indefinite status. It will remain so until a current address assignment occurs which contains only constant syllables or r.

While the indicator is in this state, no floating address assignments may occur. However, if at most one parametric syllable was in the current address assignment, then the relative address may be assigned a value (see section III.C.3.c).

c) Drum Address Assignments

In order to specify the location of sections of a program which lie in auxiliary-drum storage, a coder must use drum address assignments. These have the same structure as current address assignments except that the first syllable is always the special syllable DA. The occurrence of one of these words always causes the drum address indicator to be set to the value of the word without altering the value of the current address indicator. The drum address indicator does not have an indefinite status corresponding to that described for the current address indicator.

d) Starting Address Assignments

The last word which must be punched on every Flexo tape is a starting address assignment. This word indicates to the conversion program the address of the first instruction to be executed when the converted program is operated. It has the structure of a storage word except that the first syllable must be the special syllable START AT or i START AT.

e) Preset Parameter Assignments

Frequently a coder will want to specify an arbitrary value for a syllable. This can be done by making a preset parameter assignment. The value specified by such an assignment for a preset parameter will apply in later uses of the parameter until a new assignment for that parameter occurs.

An assignment consists of a polysyllabic word terminated by an equals sign and containing no parametric syllables other than the preset parameter being assigned. A polysyllabic storage word or a special storage word other than an input-output request, or a tab or carriage return must immediately follow the assignment. The value given to the preset parameter is the value of this subsequent word, if one appears, less the value of all the syllables in the assignment other than the preset parameter itself. Hence, for example,

$$\text{ica3+pal} = \text{al-r} + 0.04777 + \text{pb2}$$

is equivalent to

$$\text{pal} = \text{al-r} + 0.04777 + \text{pb2} - \text{ica-3}.$$

In either case the effect would be to assign a value to the syllable pal which will then be used by the conversion program in later appearances of pal in other words, as for example in

10+pal

or

DA pal

One further rule applies to the use of preset parameters in a program. Each preset parameter syllable consists of two letters and an integer. The sum of the maximum integer for each of the two letter pairs used in a program must be less than 32. That is, if the parameters pal, uq22, zz7, and zz8 are used, the rule is satisfied, since $1+22+8 < 32$. However, this is not true if any parameters other than uq1 to uq21 or zz1 to zz6 are also used.

f) Temporary Address Assignments

The syllable t is given a value by exactly the same method used to specify a preset parameter, and if t is used in a program then the sum of the maximum integers for the preset parameters must be less than 31.

Usually only one assignment is made for t, but since it is treated as a preset parameter by the conversion program it can be respecified as often as desired.

g) Relative Address Assignments

A value is given to the relative address syllable r whenever a word terminated by a comma occurs. Hence, a floating address tag is also a relative address assignment, and the value given to r in such cases is the same as that given to the floating address. However, if a polysyllabic word containing only constant syllables and terminated by a comma occurs, then a value is given to r equal to that of the current address indicator less the value of this word. Regardless of how r is defined, the value specified is substituted in all other uses of r until a new value is defined.

3. The Single Letters

The single letters b, c, r, and t, when used in polysyllabic words, must always be preceded and followed by the correct punctuation characters. These are the plus and minus signs and the various terminating characters. In only one case can one of these characters be omitted; that is when the preceding character is a plus sign and its omission causes no ambiguity in meaning. For example, the word

ca+t

can be abbreviated to

cat

without ambiguity. However, the word

ca+r+t

can be written as

car+t

but not as

cart

since rt would be considered as an improper two letter Whirlwind operation code syllable. The syllable structure is emphasized here again since the conversion program detects syllables by reading the characters of the word in sequence from left to right. The syllables are actually the longest sequences of characters which have meaning. In the case of "cart", the conversion program starts a new syllable on the letter r and, the following letter t causes confusion. However, the ca is detected as a syllable since car is not a defined syllable (see Figure II).

a) Buffer Registers

The buffer registers are referred to as b, lb, 2b, 3b, ..., and as many buffer registers are provided as is indicated by the coder. This indication is implicit: whenever the syllable b occurs in any polysyllabic word, the conversion program observes the value of the address section of the word. The maximum of these values in all such words in the program, or group of programs being converted together, determines the length of the block of buffer registers. The value of the address section is actually the sum of the values of the integer and relative address syllables in the word reduced modulo 2048. In forming the value of the converted word, the integer value of 1784 (decimal) is

given to b. This value is not added to a word until after the address section value is determined.

Care must be taken when the number of buffer registers in a program depends upon some parametric quantity. For example, writing

```
pal=10
.
.
.
icapal+b
```

will not of itself cause eleven buffer registers to be set aside by the conversion program for the interpretive routine to use; it will instead cause only one buffer to be available. If, however, the coder writes

```
pal=10b
.
.
.
icapal
```

then eleven buffers will be available.

b) Cycle Counters

The address section of interpreted instructions may be modified by the currently selected cycle counter index register by simply appending the single letter syllable c to the word. The effect of this syllable upon the converted word is to change the binary value of the operation code. This is done by giving the syllable c the octal fraction value 0.57777. The c may thus be considered a fourth letter for the interpreted cycle count instruction code.

The presence of the letter c any place in the tape being converted results in the cycle count block of the interpretive routine being provided. If no more than one cycle counter is used and no isc instruction occurs with an address greater than 0, then only the zeroth counter registers appear. However, if an isc instruction occurs with an address greater than 0, then the highest-valued such address determines the highest-numbered cycle count line occurring in the converted program. Complete details are given in Chapter XIII about the use of these counters. The address value of the instruction is the sum of the integer and relative address syllables modulo 2048. Preset parameters

cannot be used directly to specify the highest-numbered cycle counter, and a subterfuge must be used. For example,

```
pal=10
```

```
•
```

```
•
```

```
•
```

```
•
```

```
iscpal
```

will not give 11 counter lines, but

```
pal=isc10
```

```
•
```

```
•
```

```
•
```

```
•
```

```
pal
```

will.

c) Relative Addresses

Absolute and floating addresses provide two extremely different systems of addressing words in programs. The first enables a coder to know exactly where in storage a given word is located but allows him no flexibility whatsoever in varying this location. On the other hand, floating addresses do allow him complete freedom in altering the location of words or whole blocks of words, but does not immediately provide him with information as to the exact location of a given word in storage. Relative addressing provides the coder with a somewhat intermediate facility by allowing him to specify the absolute location of a given word relative to some initial point whose absolute address may or may not be known. This initial point may be a location in storage or simply a word whose floating (or relative) address is known.

In essence, the relative address system depends upon the ability to save the value of the current address indicator at some time during the conversion process and being able to use it later. In CS, whenever any polysyllabic word terminated by a comma occurs, the value of the current address indicator less the stem value of the word is copied into the relative address indicator. This indicator differs from the current and drum address indicators in that its value is not indexed when storage words occur, nor is it altered when a current or

drum address assignment occurs. Whenever the letter r occurs in any polysyllabic word except those terminated by a comma, the value of the relative address indicator is added to or subtracted from the value of the word depending upon whether the r is preceded by an (implicit) plus or a minus sign.

Often the word "Or," is used to tag the word whose location is to be later used as a base point. Thereafter, the location tagged by "Or," will be used as the value of r in such words as "ca Or" or "20r ." Hence it becomes possible to refer to the 10th or 20th word or location after this base point without having to use either absolute or floating addresses.

The importance of this facility is evident. However, certain precautions should be observed in its use. If a coder wrote the sequence

Or, ta20r

lr, ts24r

3r, ad21r

·
·
·
·

then after the third storage word the value of r would be one less than the previous value. That is, the ability to change the value of r at any place in a program can work to the coder's disadvantage in a manner which he may not immediately observe. The basic difficulty here is that relative addressing provides the ability to code in an absolute form independent of the actual location of the program in storage, and mistakes in counting or labeling the words by their relative address in this block can be just as disastrous as miscounting the absolute address of words in an absolute address program.

Another difficulty that may trap the unwary coder is the following: the value of the relative address indicator is essentially a copy at some time of the value of the current address indicator. Now if the value of the current address indicator were indefinite at the time the relative address indicator was assigned a value, then the same status is transferred to the relative address indicator. Now, if this

status is maintained until some later time when the relative address is used to assign a value to the current address indicator, then whether or not the current address indicator is then definite, it will become indefinite. That is, it is possible for the relative address r to have the indefinite status normally associated with all the other parametric syllables. Also, if the relative address has an indefinite value, then it is actually the sum of a constant syllable and a parametric syllable, and in the process of determining the binary value of the words in which r is used, the constant syllable part is used in the first summations and the indefinite part in the second. This situation may lead to undetected overflows during the summation process which are not easy to discover when looking for errors.

d) Temporary Storage

During long computations coders frequently find the need to store intermediate results for later use. The location of such storage registers can be conveniently designated by floating, relative, or absolute addresses. However, such usage of memory may be wasteful. The coder cannot easily remember, at different places in the program, which of these locations are available for similar use unless he carefully keeps a record of the contents of all these registers at all times. One way of systematizing and simplifying this procedure is to use a single temporary storage block address designator; in CS the letter t is available for this purpose.

The letter is used by the coder as an address designating the initial address of a block (of arbitrary length) used for the storage of intermediate results. The addresses of the registers in the block are t , $1+t$, $2+t$, ..., and there is, of course, no restriction upon the manner in which these registers can be used by the coder at various places in the program. The use of t , instead of the other types of address available, considerably simplifies the problem of condensing the amount of storage needed for a program. All of the library subroutines which need such storage make use of this facility and hence t is usually one of the subroutine parameters which needs to be specified. The difficulty might arise in some special circumstance, however, that two subroutines making use of the same areas of the temporary storage block are

used at the same time, for example when an arctangent routine transfers control to a square root routine. In such cases it would be convenient to have several different storage blocks. In CS this problem is solved by allowing the value of t to be respecified whenever desired by the coder. A value is assigned to t in the same manner that a value is assigned to a preset parameter. Hence, a value assigned to t applies only until another assignment occurs. Normally only one value is assigned and for reasons of clarity is best assigned at the beginning of the program tape, but coders can reassign new values whenever they desire.

D. Special Words

1. Special Storage Words

Each of these words occupies registers in the core memory or auxiliary (not buffer) drum storage of the computer. The number of registers is determined by the particular kind of word. The location occupied is determined by the preceding sequence of storage, current address and drum address assignment words.

a) Generalized Decimal Numbers

These words provide the most flexible form for decimal data input. Various representations of a number can be obtained depending upon the number system currently selected and the manner of writing the number. These representations range from fifteen bit integer or fractional numbers to two register, thirty bit fixed or floating point numbers.

A generalized decimal number must always be written with a plus or minus sign and a decimal point in order that it be distinguishable by the conversion program from integer numbers and octal fractions. If no digits at all appear, as for example when a coder writes

+.

the value of the converted number is zero. As many as 18 digits may be written, however, although at most 9 of these can have a significant effect upon the converted number. The decimal point may be placed anywhere among these digits. Coders are also permitted to write as many factors of powers of two and ten as they please in order to simplify the process of writing the numbers. Thus, a number may appear as

$$-1234.567892345x2^3 \times 10^{-12} x2^4$$

if a coder so desires. The positive exponents are written without plus signs, and the resulting number must fit into the previously specified number system. Otherwise there are no limitations upon the manner of writing the number.

b) Input and Output Subroutine Automatic Requests

The input and output pseudo-instructions are described fully in the chapters on Input-Output. Only the treatment of these words by the conversion program proper will be described here.

An input-output word may or may not be an interpreted instruction, i.e., begin with the letter "i". However, the next three capitalized letters are essential in order that the conversion program be able to determine what the special word is. The complete word, up to the terminating tab or carriage return, is replaced by an sp instruction (whether the pseudo-instruction is interpreted or not) which transfers control to a subroutine near the end of core memory. The exact location of the subroutine is determined by the output adaptation programs. In addition, the conversion program tags the location of the pseudo-instruction by a floating-address tag of its own choice. These tags do not duplicate any of the tags used by the programmer, but do count against the total number allowed the coder.

Other limitations are placed on the use of these pseudo-instructions. At most, 55 such words may occur in a program or group of programs being converted at one time. The total number of Flexo characters which may occur in all these words is also limited. If there are n requests for automatic output in a group of tapes being converted together and there are k_i , $i=1, \dots, n$, characters typed in each pseudo-instruction as it appears in the fc program tape (not in the actual output requested!), then

$$10n + \sum_{i=1}^{i=n} k_i$$

must be less than 720. Since at least four characters must be typed per request, the above restriction of at most 55 requests is never realized.

c) PA Entry and Exit Words

The Comprehensive System offers the possibility of computing alternately in the Whirlwind computer code and any one of the

CS interpreted codes. The coder controls the alternation from one of these computers to the other by making use of the special words IN and OUT. The "IN" status implied by these is that of being in an interpreted system, as opposed to being "OUT" in the non-interpreted, or Whirlwind computer. The execution of one of these words implies an actual change in computer control. Two computer control elements are available, that of the interpreted system and that of the Whirlwind system, and one of these actually relinquishes control to the other in order to effect the transition from the one computational system to the other.

The special words are converted in the following manner: the OUT becomes an sp instruction which transfers control to the register immediately following the OUT. (This location is computed on the basis of the current address indicator so that the result is consistent even in drum addressed programs.) The IN is converted to an sp to a fixed location in the PA routine.

d) The Stop Instructions

The two special words STOP and iSTOP allow the programmer to stop operation of the computer by transferring control to the utility control program (see Chapter XV). Both the STOP and the iSTOP are converted to the instruction sp25.

2. Special Control Words

These words do not occupy storage registers in the converted program. Their use affects only the form and structure of the converted program. These words differ from polysyllabic control words in that each has a unique form and a rather special significance.

a) Titles

The title is punched at the beginning of a Flexo-coded program tape in order to identify the program when the tape is printed, the results when the program is run, and the binary 556 tape if produced. A title must occur at the beginning of every CS Flexo tape and must have the initial characters "fc". Filing conventions require that a tape number follow the initial characters and be in the following form: ddd-dddd-ddddd, where the d's indicate decimal digits. Any descriptive phrases can occur following the identifying number. These may, for example, be the problem name, the coder's name, the date, information

about the program on the tape, or some of the conditions in which the tape is to be used. However, the first carriage return or tab character terminates the title and all of these phrases must be in upper case.

b) The Integer-Base Indicators

Two special words are available which alter the base in which some integers are converted. These words apply only to the integers appearing in polysyllabic storage or control words, but not to all of them. The distinction is between address-type integer syllables and numerical-type integer syllables. The special words OCTAL and DECIMAL apply only to the address integers, and even then with the exceptions stated in the discussion of the library subroutine indicators.

All integers appearing in floating address assignments, current address assignments, drum address assignments, relative address assignments, temporary address assignments (e.g., $10+t=$), and starting address assignments are considered to be address-type integers. However, the case of polysyllabic storage words, preset parameter assignments, and preset parameter and temporary address values is somewhat more complicated. Here, the integers are considered to be numerical integers if no syllables other than integers, preset parameters, or octal fractions occur in the word. Otherwise, the integers are address-type integers.

In all cases, numerical-type integers are always considered to be decimal, regardless of the presence or absence of the indicators OCTAL or DECIMAL. In the absence of these indicators the address-type integers are also converted decimally. However, if the word OCTAL occurs, then subsequent address-type integers are deemed to be octal until a DECIMAL occurs, following which they are again converted decimally, and so on. The last indicator which occurs specifies the base of the addresses printed in conversion post-mortems and in the floating address value table (decimal if no indicators).

This rather complex set of rules governing the conversion process is due to the past necessity of combining two slightly different sets of conventions used in two previous conversion programs in the one program used in the Comprehensive System. The rationale used is that all integers appearing in words where the integers are

obviously meant to be addresses would be converted as octal integers in an OCTAL program. If this meaning were not obvious, then the integers were treated as data or numerical-type integers. Hence, the distinction based on the presence of only integers, preset parameters, and octal fractions. These definitions might cause confusion to the coder who wishes to use an integer to reset an address modification counter, but the only real difficulty would arise in the case when, in an octal program, the coder found that writing, for example,

$$10+pal=10+b2$$

does not imply that $pal=b2$.

c) Cancellation of PA Requests

Frequently a coder who has written an interpreted program in the Whirlwind computer desires to modify it or change some parameters by reading in a new tape. In such cases, the coder must be careful that the new tape does not unintentionally cause a new PA routine, which might differ from the one already there, to be read into the Whirlwind memory. Since PA requests are implicit, depending upon the presence of interpreted instructions, number system designators, etc., this might easily happen. The effects of different PA routines sometimes differ only subtly and would not be the first cause of error sought by the coder. To prevent such difficulties from arising, the special word NOTPA is available. The presence of this word in a tape, or in any of a group of tapes being converted together, prevents any PA routine at all from being read into storage during the conversion process. Thus, it is possible to preserve the previous PA routine in storage and modify only the desired parts of the main program or data.

d) The Library Subroutine Indicators

Subroutines in the Library of Subroutines are identified by two special words: at the beginning of the subroutine the word

LSR

is placed, where the dots usually indicate the identifying title and number of the subroutine, and at the end appears the word

END OF SUBROUTINE

However, these special words serve more than an identification purpose. **All** of the subroutines in the library are closed routines. They are

normally coded in a relative address form with decimal addresses. In addition, some subroutines make use of preset parameters which are assigned values at the beginning of the subroutine but after the title LSR..... This requires a special integer-base convention for subroutines.

The convention followed is this: between the LSR and the END OF SUBROUTINE all integers are converted decimally except those occurring in preset parameter value specifications. These words are converted according to the conventions which exist in the main program when the subroutine is encountered by the conversion program. This is consistent with the idea that the subroutine itself is, functionally speaking, a closed box, and that the coder need be concerned only with the preset parameters he himself specifies.

e) The Number System Indicators

The only number systems available are the (15,0), i.e., the basic Whirlwind number system, and the (30-j,j) systems, where $0 \leq j \leq 15$, i.e., the interpretive systems. Three control words are available to indicate to which number system subsequent generalized-decimal numbers are to be converted. A specified number system applies until another indicator occurs.

In the absence of any other specification, the (15,0) system is assumed. Thus, in a program which consists entirely of Whirlwind code, all generalized decimal numbers will appropriately enough become single-register numbers. On the other hand, in programs written entirely in an interpretive code the word (24,6), for example, at the beginning of the tape, after the title, will fix the number system of the desired PA routine automatically provided.

A greater degree of complexity arises when both single-register and multiple-register generalized-decimal numbers are desired in a single program. In such a case the coder would have to isolate sequences of single-register numbers and precede them by a (15,0), and similarly precede sequences of multiple register numbers by, for example, (24,6)'s. In the course of designing the conversion system it was felt that some coders might desire to change the number system specified for the multiple-register numbers after having once run the program. He would then have to go through the entire tape changing all the (24,6)'s

to, for example, (23,7)'s. To avoid this problem, the special words SINGLE and MULTIPLE are provided. SINGLE may be used in place of the (15,0)'s, MULTIPLE in place of the (24,6)'s (or (23,7)'s). The desired multiple-register number system need, therefore, be placed once only, just after the title at the beginning of the tape. The effect on the system in which the multiple-register numbers are converted is exactly the same as before. Then, if it seemed necessary to change the number system of the multiple-register numbers, only one change in the tape need be made. The words SINGLE and MULTIPLE are also useful within library subroutines, which may be written without assuming the use of any particular multiple-register number system.

f) Comment Words

The comment-word facility permits a coder to preserve on the Flexo tape as many of the annotations of his coded program as he desires. These comments, which must start with a vertical bar and end with a tab or carriage return, will be printed with the code itself whenever the tape is printed, but they will be ignored by the conversion program and have no influence whatsoever on the form or operation of the code in the computer.

g) DITTO

The special word "DITTO" is intended to simplify the problem of repeating the same word in storage many times. Any storage word, special or polysyllabic, can be repeated, and the block of registers over which the repetition is extended can be either in core memory or on the drum. The word "DITTO" must be preceded by the word which is to be repeated and followed by a current address assignment which specifies the (current) address of the next register available after the end of the block. Hence coders must consider not only the number of times the word is to be repeated but also the number of registers occupied by the word itself (this is analogous to the considerations in using the corresponding ditto facility available with 556 binary tape as described in Chapter XV).

The current address assignment terminating the "DITTO" block may have the general polysyllabic structure, and the usual cautions and rules apply. However, the use of "DITTO" can be facilitated

by the technique of tagging the word which is to be repeated by the relative address tag "Or," and specifying the terminating current address assignment in terms of "r". For example, suppose it is desired to repeat the word "+1" 40 times, i.e., place the word "+1" in storage 41 times. This block is to begin at the current location in storage. Since this word is a single register word, we would write

Or,+1

DITTO

41r |

If another storage word is written following the current address assignment "41r ", then this word will immediately follow the end of the block of forty-one, +1's. This technique permits a block to be formed at any time during the course of writing a code without forcing consideration of the actual location of the block in storage and does not affect the definiteness or indefiniteness of the current address indicator.

At most, 511 registers can be occupied by the repeated word. Thus, in the above example, if we instead desired to fill 1000 registers with +1's, we would subdivide the block and perhaps write

Or,+1

DITTO

512r | Or,+1

DITTO

498r |

The current address assignment at the end of the block cannot have a value less than that of the current address at the beginning of the block; however, a word can be repeated zero times by writing

Or,+1

DITTO

1r |

or the equivalent; the effect is that the word appears once in storage.

If a double-register number is repeated, the coder must remember that the block of registers occupied will be twice as long as in the corresponding case with single-register words.

IV. Error Detection During the Conversion Process

The coding language of the Comprehensive System has a syntax whose rules must be obeyed in order to have a coded program properly converted. Since some of the effects of violating these rules are subtle, an effort was made to have the conversion program detect some of the more common errors during the conversion process. For example, one of the common errors that may get by even a careful checking of a code is the failure to tag a word by a floating address used in the address section of an instruction elsewhere in the program. This kind of error is detected and indicated by the conversion program. On the other hand, the use of undefined two or three letter operation or pseudo-operation codes is not detected by the conversion program. Usually this error is immediately obvious to a coder who knows the operation codes.

The conversion program prints an indication of the error committed on the direct output typewriter and also on the delayed-output tape unit. A complete list of all the kinds of detected errors follows. In all cases except where explicitly mentioned, the conversion process is carried no further when an error is detected.

Unassigned Flads

This results when a floating address is used in a word and that floating address has not actually been assigned a value. The locations of the first thirty such errors are recorded on the delayed output unit only, and the complete table of all the assigned floating addresses in the usual form is recorded immediately thereafter. The conversion of the program is completed with the value zero given to all the unassigned floating addresses, so the program can be run if this still seems desirable.

Duplicate Flad is --

Sometimes a coder tags two or more different locations or words with the same floating address tag. Under certain conditions this is an error, when it is not clear which value is to be used. The convention in CS is that such reassignments will be permitted and the last value assigned will be the one used, provided each reassignment is separated from the preceding assignment of the same tag by a current address assignment. However, if two inconsistent assignments of the same tag

occur without an intervening current address assignment, then an error indication will be given and the conversion process will stop. The error can then be found by scanning the floating address tag assignments on the print of the program tape.

This convention was established with the intention that coders should be permitted to make "patch" modifications at the end of a tape and to reassign tags in them. Such modifications are always initiated by a current address assignment. Coders should note, though, that duplicate tag assignments which occur within the main body of the program tape and which are separated by current address assignments terminating a DITTO block, for example, will prevent an error indication from being given.

Too Many Flads

A programmer may use at most 255 distinct floating addresses in a program. Actually this limit is only rarely attainable in practice because of other limitations arising during the conversion process. These are the following: each automatic input-output request special word makes use of an implicitly-defined floating address tag over which the coder has no control and which hence causes one to be subtracted from the maximum number of usable floating addresses. The coder can count the number of such requests in his program and reduce 255 by that quantity. If the number of available tags is exceeded, then the above error indication is given.

The conversion program determines the number of floating addresses used by counting not only those which are used to tag words, but also those which occur in any kind of polysyllabic word. Hence if a coder, uses, for example, both a255 and b255 in a program and does not tag any word with either of these, then he will still receive an error indication.

Indefinite Flad

If a coder tags a word with a floating address while the current address indicator is indefinite, then he will receive this post-mortem. For example, this will happen if he writes

```

pal | -
      -
      -
      -
      -
      b2,-

```

This error is detected during the first pass of the conversion program, and the computer is stopped while the part of the Flexo tape containing the error is still in the tape reader.

Too Many Characters in Output Requests

At most 55 automatic input-output request special words can occur in a program being converted. Actually this limit can be attained only if all these words consist at most of the three upper case letters defining the request, for the number of additional characters which can be written is limited. The exact formula is given in Section III.C.1.b).

Program Too Long at --

The conversion program detects whether or not the part of the program being converted which will lie in core memory will occupy any of the space already occupied by the desired interpretive or input-output routines. If every block of consecutive words which does overlap is initiated by a current address assignment which has a value greater than that of the lowest addressed register of the interpretive and input-output routines, then no error indication results. However, if any such block only partially overlaps these routines, then there will be an error indication. The intention of this rather fine distinction is to permit coders to modify particular sections of the interpretive routines if they so desire. Unfortunately, some unintentional modifications will not be detected.

gd Number at --

This indication occurs when a generalized-decimal number is too large to fit into the number system selected at the time the number occurs.

t Unassigned

If a reference to the temporary storage registers occurs in any word at a time when no value has previously been assigned to t, then this indication will be given. The request can be found by examining the Flexo tape at the point where it stopped in the tape reader during the conversion process.

Illegal Characters

This indication is given when one of the illegal characters listed in Figure 1 occurs on the Flexo tape being converted. The illegal character will be located in the reader when the computer stops.

Treatment of Flexowriter Coded Characters
by The Conversion Program

Binary Numerical Sequence

Character 123456	Lower Case	Upper Case	CS Treatment	Character 123456	Lower Case	Upper Case	CS Treatment
000000	not used		ignored	100000	t	T	accepted
000001	not used		illegal	100001	not used		illegal
000010	e	E	accepted	100010	z	Z	accepted
000011	8	8	accepted	100011	back space		illegal
000100	not used		illegal	100100	l	L	same as 1
000101			accepted	100101	tabulation		accepted
000110	a	A	accepted	100110	w	W	accepted
000111	3	3	accepted	100111	not used		illegal
001000	space	bar	ignored	101000	h	H	accepted
001001	=	:	accepted	101001	car. return		same as tab
001010	s	S	accepted	101010	y	Y	accepted
001011	4	4	accepted	101011	not used		illegal
001100	i	I	accepted	101100	p	P	accepted
001101	+	/	accepted	101101	not used		illegal
001110	u	U	accepted	101110	q	Q	accepted
001111	2	2	accepted	101111	not used		illegal
010000	col. change		ignored	110000	o	O	same as zero
010001	.)	accepted	110001	stop		ignored
010010	d	D	accepted	110010	b	B	accepted
010011	5	5	accepted	110011	not used		illegal
010100	r	R	accepted	110100	g	G	accepted
010101	1	1	accepted	110101	not used		illegal
010110	j	J	accepted	110110	9	9	accepted
010111	7	7	accepted	110111	not used		illegal
011000	n	N	accepted	111000	m	M	accepted
011001	,	(accepted	111001	upper case		accepted
011010	f	F	accepted	111010	x	X	accepted
011011	6	6	accepted	111011	not used		illegal
011100	c	C	accepted	111100	v	V	accepted
011101	-	-	accepted	111101	lower case		accepted
011110	k	K	accepted	111110	0	0	accepted
011111	not used		illegal	111111	nullify		ignored

FIGURE 2

Vocabulary of Syllables, Words, and Special CharactersI. Syllables

A. Constant syllables

1. WW operations: si, rs, bi,, md
2. Interpreted operations: isc, icr,, isp
3. Integers: 0, 1, 2,, $2^{15}-1$
4. Octal fractions: 1.00000, 1.00001,, 0.77777
5. Single letters: a) buffer register "b"
b) cycle counter "c"

B. Parametric syllables

1. Floating address tags: al, ..., a255, ..., z255
(l and o excluded as letters)
2. Preset parameter tags: pal, ..., pz40, ual, ..., uz40,
zal, ..., zz40 (l and o excluded as letters)
3. Single letters: a) relative address "r"
b) temporary address "t"

C. Special syllables

1. Drum address: DA
2. Starting address: START AT, iSTART AT

II. Word terminating characters

- A. Tab and carriage return: \rightarrow | \curvearrowright
- B. Vertical bar: |
- C. Comma: ,
- D. Equal sign: =

III. Words

A. Polysyllabic words

1. Storage words: The sum of any constant and/or parametric syllables terminated by a tab or carriage return, e.g.,
caf83, -50+r, ics al+pa2-5r
2. Control words: The sum of certain constant, parametric and/or special syllables terminated by a suitable terminating character:
 - a) Floating address assignment: al,
 - b) Current address assignment: 50+al |
 - c) Drum address assignment: DA 50+al |

- d) Starting address assignment: START AT a1
- e) Preset parameter assignment: pal=q4
- f) Temporary address assignment: t=q7
- g) Relative address assignment: lOr,

B. Special words

1. Storage words

- a) Generalized decimal numbers: $-123.45678 \times 2^{-3} \times 10^5$
- b) Input-Output requests: iMOA+1.234s
- c) PA entry and exit words: IN and OUT
- d) Stop instructions: STOP, iSTOP

2. Control words

- a) Titles: fc.....
- b) Integer base indicators: OCTAL, DECIMAL
- c) Library subroutine indicators: LSR....., END OF SUBROUTINE
- d) Cancellation of implicit PA request: NOTPA
- e) Number system indicators: SINGLE, MULTIPLE, (m,n)
- f) Comment word: |.....
- g) Ditto indicator: DITTO

CHAPTER IV: THE UTILITY CONTROL PROGRAM

- I. The Problem Organization Process
 - A. Automatic Coding Systems
 - B. The Computation Center
 - C. Computer Operation

- II. The Utility Control Program
 - A. The Automatic Mode
 - B. The Manual Mode

- III. The Drum Utility Programs
 - A. The Binary Input Program
 - 1. The Block Length Control Word
 - 2. The Block Address Control Word
 - 3. The Check Sum Control Word
 - 4. The Ditto Control Word
 - 5. The Starting Address Control Word
 - B. The CS Translation Program
 - C. The Generalized Post-Mortem Program

- IV. The Performance Request
 - A. The Standard Abbreviations
 - B. Examples of Operating Instructions

- V. Director Tapes
 - A. Restrictions on Director Tapes
 - B. Post-Mortem Request Tapes in Director Tape Runs
 - C. Additional Director Tape Vocabulary

I. The Problem Organization Process

It is convenient to distinguish the following four subdivisions in the process of solving a problem on a digital computer.

- (1.) Programming
- (2.) Coding
- (3.) Transcribing
- (4.) Operating

Programming consists of preparing a general plan for the solution of a problem and includes such decisions as

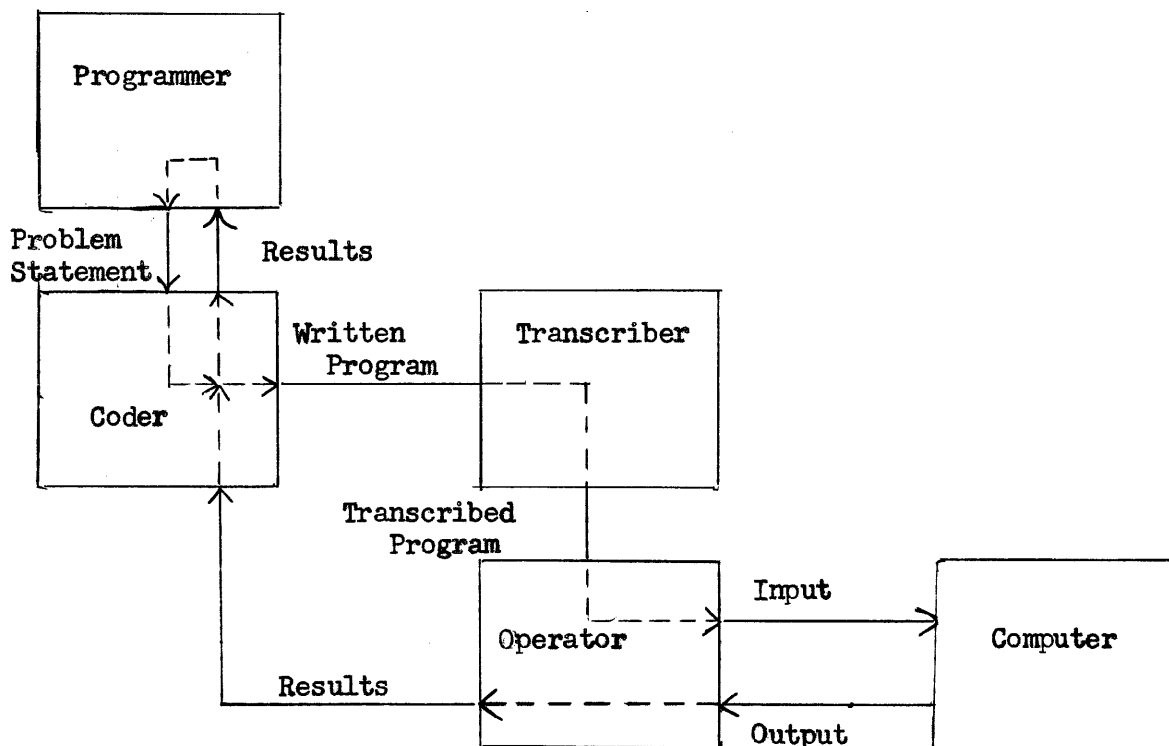
- (1.) Choice of mathematical model
- (2.) Choice of numerical method
- (3.) Parameter specifications
- (4.) Evaluation of results.

The language of programming is varied. Generally it consists of English words and mathematical symbols imbedded in a flow diagram.

Coding consists of reducing the flow chart to a program expressed in terms of machine code.

The coded program must then be transcribed onto suitable input media, introduced into the computer storage and operated.

The problem organization process is illustrated schematically below.



A. Automatic Coding Systems

The programmer has almost complete freedom in his choice of language and will choose the language best adapted to stating the problem. The coder, on the other hand, is restricted to using the machine code for a language. A machine code is, in general, convenient only to the builder of the machine.

The automatic coding system attempts to ease the coder's task by allowing him to state the problem in new languages (called input languages) which are richer than the machine code. The system provides a device (called a translation program) for translating input language into machine code.



B. The Computation Center

The problem organization process is realized in a computation center. The basic activities involved have already been described in section 1. Superimposed on these activities will be a clerical function, since the center must provide for the flow of information between the various activities and the keeping of records.

Work on the WWI computer generally is done on an open shop basis, i.e. the person with the problem to solve does the programming and coding. Transcription (in this case the punched paper tape) and computer operation are performed by trained specialists. Communications between the programmer and the typist proceeds via a standard form called a tape requisition. Communication between the programmer and the computer operator proceeds via a standard form called a performance request.

C. Computer Operation

Computer operation consists of organizing input to and output from the computer, and the keeping of suitable records of machine operation. Operation will become more complicated as the computation center begins to accumulate a library of utility programs (i.e., programs which perform certain more or less routine tasks, e.g., automatic coding systems and

post-mortem routines). The need for speed and accuracy in computer operation is obvious.

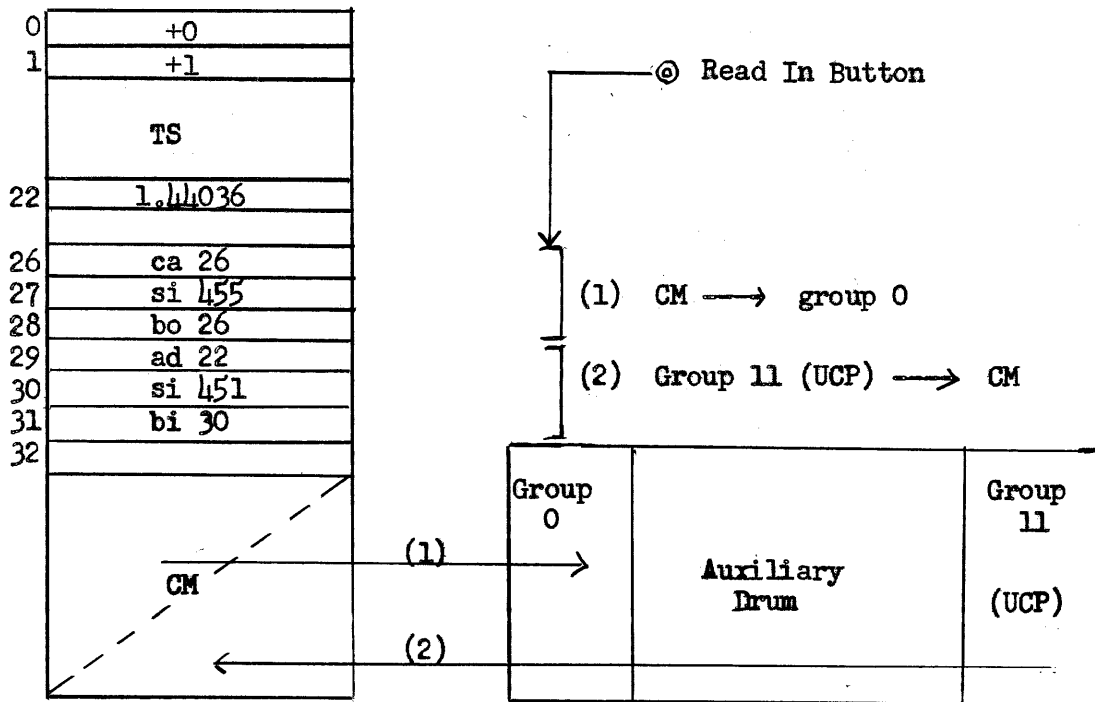
II. The Utility Control Program

Operation of the WWI computer has been partially automatized. The system is centered in the use of a program called the utility control program (UCP) which has been semi-permanently recorded on group 11 of the auxiliary drum (AD). (1.)

Input to the WWI computer is initiated by pressing a push button on the control panel which is called the read-in button. This starts the computer at the instruction contained in register 26 of toggle switch storage (TS) which is the normal entry point of a short program called the TS input program. The TS input program does the following:

- (1) Copies the contents of magnetic core memory (MCM) onto group 0 of the auxiliary drum.
- (2) Copies the contents of group 11 of the AD (i.e. the UCP) onto CM.

Computer control at this point passes from register 31 of TS to register 32 of CM (and hence to the UCP). The process is illustrated below.



(1) The recording heads on this particular group have been disabled so that information can be read from the group but not recorded on the group.

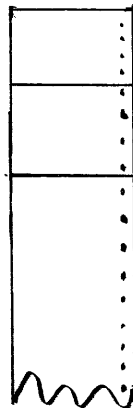
At this point the UCP takes charge. The modes of operation of this program can be described best by splitting them into the following two categories:

(1) An automatic mode - in which the UCP assumes that a punched paper tape will be read in on the photoelectric tape reader (PETR).

(2) A manual mode - in which the UCP assumes that the operator has manually inserted information describing what he wants the UCP to do.

A. The Automatic Mode

The punched paper tapes which are encountered by the UCP have the following general form:



(1) Identifying characters

(2) Title

(3) Main body of tape

(1) The identifying characters describe the kind of tape which follows. For example

- (a) fc denotes CS flexo tapes
- (b) fb denotes binary tapes
- (c) fp denotes post-mortem request tapes.

Other kinds of punched paper tape exist but will normally not be encountered by the user of the CS system.

(2) The tape title contains the following information in the following order

- (a) Tape number
- (b) An optional tape description
- (c) A terminating carriage return.

The tape number consists of three integers separated by dashes,

$$x - y - z.$$

The first integer (x) is the problem number and is fixed by the problem. The second number (y) is the programmer's number which never changes

(for the particular programmer). The third integer (z) may be arbitrarily chosen by the programmer and is used to identify the particular tape.

The tape description may consist of any collection of upper case characters that can be typed on a flexowriter and should at least include the programmer's name.

A sample tape title is the following:

fc	165 - 17 - 302	SINE ROUTINE SMITH
	└──────────┘	└──────────┘
	Tape number	Tape Description

The UCP reads in the tape title, identifies the kind of tape following and logs the tape title.

The following logs are kept by the program:

- (1) a film log
- (2) a paper tape log.

(1) The digits of the tape number are displayed on the output oscillograph and photographed. Each dash in the title terminates a line on the scope. For example, the above tape number appears as:

165

17

302

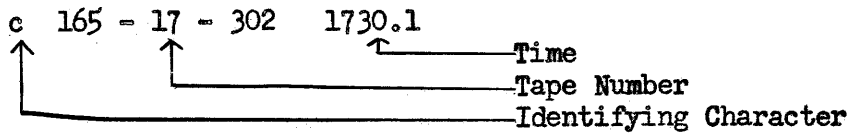
The photographed titles serve to identify any information recorded on the scope by the program.

(2) Logging information concerning the tape is also recorded on a punched paper tape (produced by a flexowriter punch which is directly connected to the computer). The following information is entered in the log:

- (a) An identifying character
 - c for fc tapes
 - b for fb tapes
 - p for fp tapes
- (b) The tape number (except for fp tapes)
- (c) The time of read-in.

For example, the preceding tape title could record the following entry in

the paper tape log:

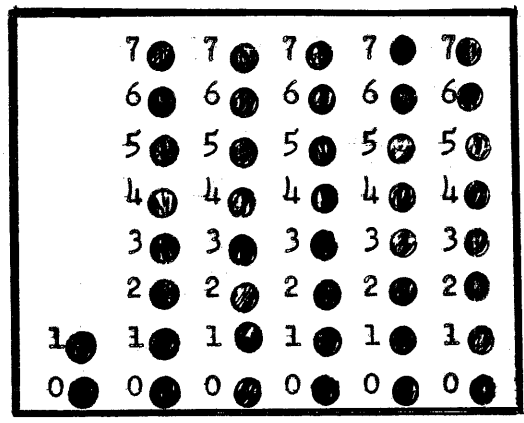


The paper tape log gives a fairly complete picture of the sequence of operation and is used as a basis for charging for computer time.

The utility control program finally selects the utility program required to translate the main body of the tape (on the basis of the identifying information), brings the required utility program into CM and transfers computer control to it.

B. The Manual Mode

The computer operator specifies the manual mode by pressing a push button on the console called the examine selector panel button. The mode required is specified by placing a characteristic number in a set of octal push buttons called the selector panel. This is illustrated below.



SELECTOR PANEL

Sign Numerical Digits
Digit

For example, if the number 0.00001 is placed in the selector panel, then selection of the manual mode by the operator will cause the CS translation program to produce binary (fb) tapes whenever CS flexo (fc) tapes are translated.

A second set of octal push buttons called the insertion panel also exists on the console and is used (in conjunction with the contents of the selector panel) to specify manual modes requiring a parameter.

For example, the operator can examine the contents of any register on the drums by placing the octal number, 0.00002, in the selector panel, the address of the register in the insertion panel, and selecting the manual mode.

A large number of manual modes exists, but their use by the CS coder is rare.

III. The Drum Utility Programs

In order to speed up the operational process, the utility programs in daily use on the WWI computer have been semi-permanently stored either on group 11 of the AD (along with the UCP) or as numbered (identifiable) blocks on a magnetic tape unit (number 0).^(1.)

If the required utility program is permanently stored on group 11, then the utility section process is trivial (since the utility program has already been copied into CM by the TS input program). If the required utility program is permanently stored on magnetic tape unit 0, then the UCP searches tape for the required program and copies it onto CM.

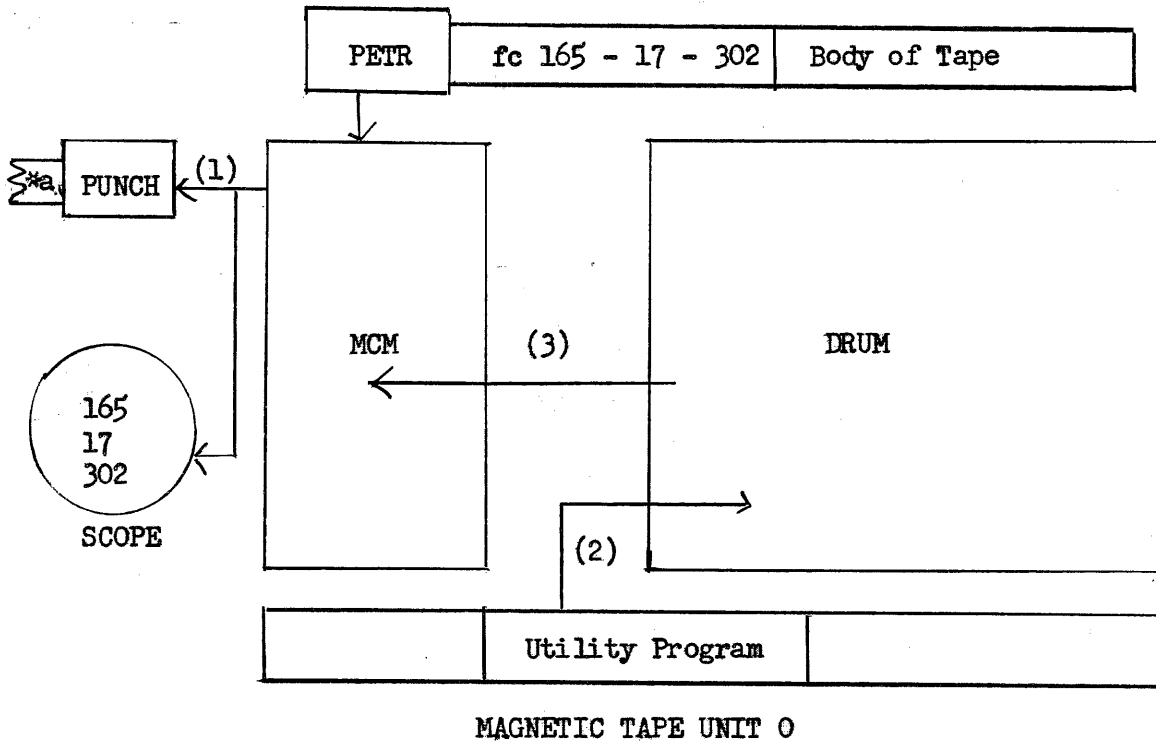
A further refinement of the latter process has been made for programs which are used most frequently and which are too long to fit on group 11. These programs, called drum utility programs, are permanently recorded on magnetic tape but operate from only one of the magnetic drums.

Whenever a drum utility program is required, the UCP first checks a specified range of registers on a drum to see if the required program is on the drum or not. This is done by summing the contents of all of the registers in the range and comparing it with a standard sum. If the utility program is already on the drum it is brought into CM and operated. If the utility program is not on the drum it is brought from magnetic tape unit 0 to the drum and a new standard sum is formed.

(1.) The recording heads on magnetic tape unit 0 have been disabled.

Logging and utility program selection are schematically illustrated below:

*a c165-17-302 1730.1



The utility programs of interest to the CS programmer are the following:

- A. The binary input program
- B. The CS translation program
- C. The generalized post-mortem program.

A. The Binary Input Program

An input language must perform at least the following two functions:

(1) Provide a vocabulary of program words which translate into machine words and appear in the computer storage.

(2) Provide a vocabulary of control words which specify the locations in computer storage at which program words appear.

The binary input program translates binary (fb) tapes and is stored on group 11 of the AD (along with UCP). The vocabulary of program words and control words handled by the binary input program is relatively simple.

There is only one kind of program word, namely the 16 digit binary number which is stored on paper tape in the 556 form described in Chapter XII. There are several control words to be described. These are also stored on paper tape as 16 digit binary numbers in 556 form.

A distinction between control words and program words on tape is made by the sequence with which they occur on the tape. This can be done, for example, if the first word on tape is always a control word and if each control word contains implicitly in its definition the number of program words following on tape before the appearance of the next control word. In practice this number can be zero, a fixed number, or be a function of a previous control word.

1. The Block Length Control Word

The block length control word is read as a negative integer having the form, $-n + 1$, where $n = 1, 2, \dots, 2048$. This control word informs the binary input program that the next block of program words appearing on tape is n words long. The next word appearing on tape, however, is assumed to be a control word by the binary input program. In particular, this could be another block length control word which would supercede the previous one.

2. The Block Address Control Word

Program words can be stored on groups 0 to 10 of the AD and groups 2 to 7 of the buffer drum (BD) by the binary input program. The block address control word is used to specify the location of the initial word in a block of program words.

Addresses on the drum are specified as follows:

(a) The sign digit of the block address control word is used to specify the magnetic drum desired. A zero in this digit denotes the AD, a one in this digit, the BD.

(b) The 15 numerical digits of the block address control word are used to specify the numerical address of the drum register desired. The numerical address of the word located on group n register m is

defined to be

$$n \cdot 2048 + m$$

Note that this follows the convention described in Chapter XII for reading from and recording on the drum.

The next n words on tape (where n is specified by the block length control word preceding) are assumed to be program words by the binary input program and are placed on the drums. The word on tape following the block of program words is assumed to be a control word.

Two examples follow:

- | | | |
|-----|---------|--|
| (1) | -2 | Block length control word |
| | 0.40040 | Block address control word (AD 8 - 32) |
| | ca 0 | } Three program words |
| | ca 1 | |
| | ca 2 | |
| (2) | -0 | Block length control word |
| | 1.20040 | Block address control word (BD 4 - 32) |
| | ca 0 | One program word. |

The block length control word may be omitted before program blocks consisting of a single word. Thus example (2) above could have been written as

1.20040
ca 0

3. The Check Sum Control Word

The check sum control word is best read as the WWI instruction, ck 5. It must always be followed by another control word on tape called the check sum.

The occurrence of a check sum control word causes the binary input program to compare the check sum with a sort of sum-mod-one of the preceding block of program words (which was formed while the words were being read into the computer by the binary input program). An alarm is generated if the two do not agree.

Example (1) with a check sum is shown below:

-2	Block length control word
0.40040	Block address control word
ca 0	} Program Words
ca 1	
ca 2	
ck 5	Check sum control word
1.40045	Check sum.

The check sum control word must immediately follow the block of program words being checked. The word on tape following the check sum is assumed to be a control word.

The occurrence of check sums on a binary tape is optional.

4. The Ditto Control Word

The ditto control word is read as the WWI instruction, ck 512 + m, where m = 0, 1, 2, ... If a ditto control word appears on tape the binary input program records the next block of program words appearing on tape on the drum m times in a block of consecutive registers. The initial address of the expanded block is specified by the drum address control word appearing with the block of program words on tape.

Two examples follow:

(1)	ck 512 + 2	} becomes	0.00500 + 0	(1)		
	-1				ca 0	
	0.00500		} becomes	0.00500 + 0	(2)	
	+0					ca 0
	ca 0					
	ck 5					
	ca 500					
(2)	ck 512 + 3	} becomes	0.00500 + 0	(1)		
	0.00500				+ 0	(2)
	+0				+ 0	(3)

If m = 0, i.e., ck 512, the next block of program words appearing on tape is ignored.

The first word appearing on tape after the block of program words is assumed to be a control word.

5. The Starting Address Control Word

The starting address control word is best read as a WWI instruction, spm. It must always be followed on tape by another control word of the form, spx, which is called the starting address. The starting address control word normally serves to terminate the read-in of the paper tape.

Two cases can arise:

(1) $m \geq 3$: If the address, m , of the starting address control word satisfies the inequality, $m \geq 3$, then the starting address control word (and the starting address) are ignored. The next word appearing on tape (after the starting address) is assumed to be a control word.

(2) $m = 1$ or $m = 2$: In this case the binary input program stores the starting address (spx) in TS register 2 (which is a flip-flop register), copies the contents of AD group 0 into CM and transfers computer control to register m (i.e. 1 or 2). A partial picture of the contents of TSS at this moment is shown below:

0	+0
1	+1
2	spx

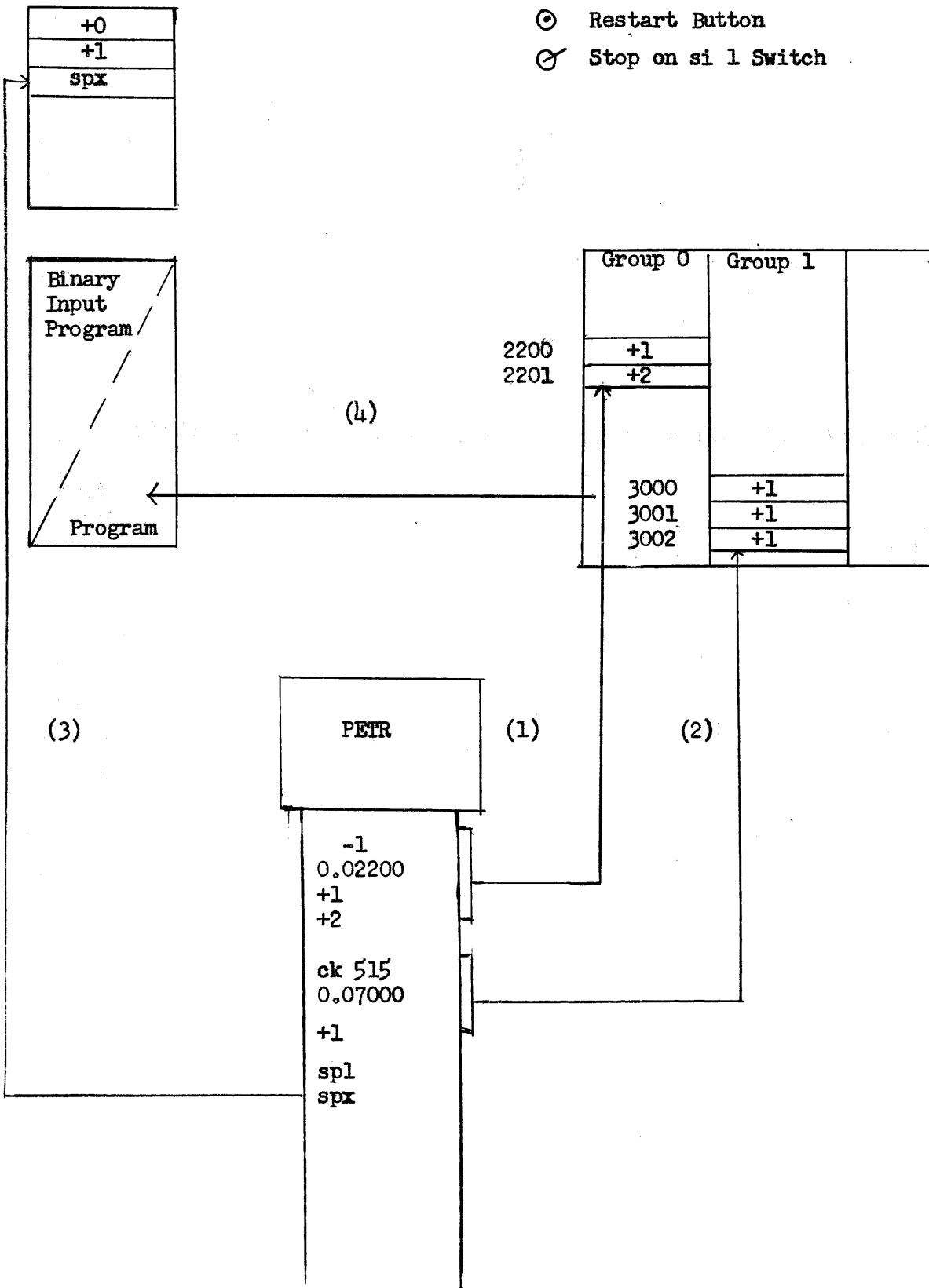
If $m = 1$, the computer is stopped (on an si 1 instruction). The operator may now operate the program (brought from AD group 0 to MCM) by pressing the restart button. This transfers computer control to the starting address (x) of the program as obtained from paper tape.

If $m = 2$ (or if $m = 1$ and the stop on si 1 switch is off), then the program immediately operates (beginning at the starting address).

A copy of the program before operation will remain on AD group 0 unless it is distributed by the program or unless another read-in occurs.

The operation of the binary input program is illustrated schematically on the next page.

- ⊙ Read In Button
- ⊙ Restart Button
- ⊗ Stop on si l Switch



B. The CS Translation Program

The CS translation program translates CS Flexo tapes and stores the translated words on the auxiliary drum. Translation takes place in two stages:

(1) The paper tape is read in and partially translated results are recorded on a magnetic tape unit (number 1). Partial translation is terminated by the appearance of a START AT control word on the paper tape which causes the computer to stop on an si l instruction (provided the stop on si l switch is on). The operator then has two choices. If other fc tapes are to be translated with the previous fc tape as a dependent set, then the operator places these in the PETR and presses a push button on the control console called the start at 40 button.

(2) If no further dependent tapes exist, then the operator completes the translation by pressing the restart button.

The effect of a START AT block on an fc tape is identical to that of the spl starting address control word on an fb tape. The starting address (spx) of the program is stored in TS register 2, the contents of AD group 0 is copied into CM and computer control is transferred to TS register 1 (containing an si l instruction).

The CS translation program is a drum utility program and is stored on BD groups 4, 5 and 6. The structure of fc tapes is described in Chapter XIV.

Read-in of an fb or an fc tape as previously described consists of reading selected blocks of words from paper tape to selected blocks of registers on the drum. Words appearing on tape replace those previously stored on the drums. Other registers on the drums are left undisturbed.

A useful variation of this process is produced when the operator pushes a button on the console called the erase button before reading in the tape. If this is done, the utility control program records the word, +0, on all registers of AD group 0 before translating the tape. Thus, after read-in has taken place all registers in CM whose contents are not specified on the tape will contain +0.

Erasure before read-in is recommended if possible since the retention of meaningless information in storage registers serves to confuse the situation.

C. The Generalized Post-Mortem Program

The generalized post-mortem program translates post-mortem request (fp) tapes and records the required requests on specified output units. When this is completed, the program copies AD group 0 into CM (thus restoring CM to its contents prior to translating the fp tape), and stops the computer (on the si 0 instruction in TS register 0).

The generalized post-mortem program is a drum utility program and is stored on BD groups 6 and 7. The structure of the fp tape is described in Chapter XVII.

IV. The Performance Request

Communication between the programmer and the operator takes place on a standard form called performance request which conveys the following information to the operator:

- (1) Type and amount of expected output
- (2) Alarm procedure to be followed
- (3) The operating instructions for the run.

The type and amount of expected output must be specified so that the operator can organize runs on the computer. It is all too easy to tie up the computer because a particular piece of output equipment is not available.

The alarm procedure tells the operator what steps to follow if the program does not work.

The operating instructions describe to the operator the sequence in which the tapes are to be translated and operated.

In the final analysis, this implies stating the exact sequence of button pushings to the operator. In order that the operating instructions can be briefly, accurately, and uniformly stated, a set of standard abbreviations has been developed for use by the programmer.

A. The Standard Abbreviations

- (1) e, Press the erase button
- (2) ri, Press the read-in button
- (3) rs, Press the restart button
- (4) sah0, Press the start at 40 button

- (5) fc 100-0-0, Place the corresponding tape in the PETR
 fb 100-0-0,
 fp 100-0-0,
- (6) si l switch on, Turn the stop on si l switch on
- (7) si l switch off, Turn the stop on si l switch off
- (8) rm x Set the selector panel to x (octal) and press
 the examine selector panel button.
- (9) lm x Set the insertion panel to x (octal).

B. Examples of Operating Instructions

- (1) fb, fc or fp tapes can be translated by the following sequence:

tape no., ri,

e.g. fb100-0-0, ri,

MCM can be erased prior to translation by the following sequence:

e, tape no, ri,

- (2) fb and fc tapes can be both translated and operated by the following sequence:

tape no., ri, rs,

Operation begins at the starting address contained on the tape.

If the stop on si l switch is off, then operation occurs automatically after translation. Thus translation and operation (beginning at the starting address on tape) can also be specified by the sequence:

si l switch off

tape no., ri,

Automatic operation also occurs if an fb tape ends with an sp 2 starting address control word. Translation and operation for such a binary tape can be specified by:

fb 100-0-0, ri,

- (3) Multiple (independent) translations before operation can be specified by the sequence:

tape no., ri,	tape no., ri, ...,	rs,
translation	translation	operation

e.g.

fb 100-0-0, ri, fb 100-0-1, ri, rs

Multiple translation can occur only if the stop on si l switch is on.

(4) Dependent fc tapes (having overlapping flads or parameters) must be translated as if they were a single tape. Dependence of a set of tapes is indicated on the performance request by writing the tape numbers in the proper order and following each tape number with a right-hand parenthesis.

tape no.), tape no.), ..., ri,

e.g.

fc 100-0-0), fc 100-0-1), ri,

Dependent tapes can be translated only if the stop on si 1 switch is on.

(5) A binary (fb) tape can be produced during translation of an fc tape by writing the standard abbreviation, ric, in place of the standard abbreviation, ri.

tape no.), tape no.), ..., ric,

Since this is done by the manual mode when 0.00001 is placed in the selector panel, the following sequence could also be used:

rm 1, tape no.), tape no.), ..., ri,

(6) Complex runs can be obtained by sequentially requesting simple runs. For example:

e, fc 100-0-0, ri, rs,	fc 100-0-1, ri, rs,
└──────────────────┘	└──────────────────┘
simple run	simple run

In the above example the assumption is made that the operation of the program is terminated by stopping the computer. This can be accomplished in many ways (e.g., by having the computer execute the WWI instructions si 0 or si 1, or equivalently the instructions sp 0 or sp 1), however, for several reasons it is best for programmers to terminate operation by means of the special instructions, STOP or iSTOP. Both of these stop the computer on the WWI instruction, si 0, which is contained in TS register 0.

(7) Program operation in a complex run can also be terminated so as not to stop the computer. This is done by executing the WWI instruction, sp 26, which calls in the utility control program and translates the next tape in the run.

For example, if (in the preceding example) tape fc 100-0-0 used the

sp 26 instruction for termination, the operating instructions should be:

e, fc 100-0-0, ri, fc 100-0-1, rs, rs
translate operate operate
 and
 translate

It should be noticed that the operator has been requested to place tape fc 100-0-1 in the PETR before operating tape fc 100-0-0 since translation of fc 100-0-1 will occur automatically after the operation of fc 100-0-0.

Complications of this sort can be avoided by physically splicing together the two tapes which can be identified by the single number, fc 100-0-0.

The operating instructions could then read:

e, fc 100-0-0, ri, rs, rs
 | operate
 operate
 and translate

(8) The operation of the preceding complex run can be made still more automatic by requesting that the operator turn off the stop on si 1 switch. In this case operation occurs automatically after translation, and the operator is not required to push the restart button to initiate operation. The above operating instruction could then read:

si 1 switch off,
e, fc 100-0-0, ri,

Translation and operation of both tapes occur automatically after pushing the read-in button.

(9) A program may be operated beginning at an arbitrary address, say x, by means of the operating instruction

sa x,

For example:

e, fc 100-0-0, ri, rs, sa x
 | | | |
 Translation Operate
 Operate

In this example the program has been operated twice: once beginning at the starting address (rs) and once beginning at the instruction contained in register x.

V. Director Tapes

A director tape for a particular run on the computer is a punched paper tape obtained by typing the operating instructions for the run on a flexowriter.

The objective of the director tape is to duplicate automatically the sequence of button-pushings that would be performed by the operator in manually executing the run. The operator places the director tape in the mechanical tape reader and the run tapes (spliced together in the proper sequence) in the PETR. The run is then executed by pushing the read-in button once. This objective is almost realized. There are, however, certain restrictions on the use of director tapes which will be described.

The interpretation of the director tape is performed by a program (called the director tape program) which is part of the utility control program. The director tape program is a drum utility program and is stored on BD group 2.

There are two advantages in using director tapes which are especially important in complex runs.

(1) Automatic operation is more efficient since computer down time required for operator action is eliminated.

(2) Automatic operation insures that the operating instructions will be carried out exactly as indicated.

A. Restrictions on Director Tapes

(1) Only standard abbreviations (as described in previous sections) can be used in the operating instructions.

(2) The operation of programs must be terminated by either the sp 26 instruction or the STOP (iSTOP) instruction.

The reason for the latter restriction is that program operation must not be terminated by stopping the computer. The STOP (iSTOP) instruction detects whether or not a director tape is being used. If a director tape is not being used, the execution of the STOP instruction will stop the computer. If a director tape is being used, the execution of a STOP instruction will call in the director tape program which then obtains the next operating instruction from the director tape and continues operation.

B. Post-mortem Request Tapes in Director Tape Runs

During manual operation the operator is frequently requested to perform a post-mortem only if an alarm occurs. Since post-mortem request tapes must be spliced with the program tapes some means must be provided for ignoring an fp tape if no trouble occurs. The following procedure has been adopted.

If a director tape is being used, then post-mortem request tapes are not executed during translation. Instead, they are stored on the buffer drum and remain there undisturbed until another post-mortem request tape is translated. If an alarm occurs during a program the operator can execute the post-mortem requests tape from the buffer drum by a simple manual procedure. If no alarm occurs, the requests can be completely ignored.

Request tapes which have been translated and stored on the buffer drum can also be executed by the following sequence of operating instructions in the director tape:

rm 33, ri,

The following example illustrates the use of post-mortem request tapes in a director tape drum:

fp 100-0-0, ri,	fp 100-0-0 stored on BD
e, fc 100-0-0, ri, rs,	Translate and operate
fp 100-0-1, ri,	fp 100-0-1 stored on BD
rm 33, ri,	Operate fp 100-0-1

If an alarm occurs during the operation of fc 100-0-0 the operator can manually obtain fp 100-0-0. If no alarm occurs fp 100-0-1 operates automatically.

C. Additional Director Tape Vocabulary

The following standard abbreviations can be used as operating instructions if and only if a director tape is to be prepared for the run:

- (1) isa x, operate the program in CM by interpreting the instruction contained in register x (octal).
- (2) eg x, record +0 on all registers of AD group x (octal).
- (3) bo x, copy the contents of TS and CM onto AD group x (octal).

(4) bi x, copy the contents of AD group x (octal) into TS and
 CM.

The standard abbreviation, rs, may be used after the execution of a STOP (iSTOP) instruction when a director tape is being used and has the following effect:

If the STOP (iSTOP) instruction was executed as a WWI instruction, then the program is re-operated by executing the word following the STOP instruction in CM as a WWI instruction.

If the STOP (iSTOP) instruction was interpreted, then the program is re-operated by executing the word following the STOP instruction as an interpreted instruction.

All director tapes must be introduced by a title of the form:

fd x - y - z

where x - y - z is a tape number in the usual sense.

Director tapes should be terminated by the special operating instruction:

sa 0,

which stops the computer.

CHAPTER XVI: OUTPUT

I. General Procedures

The following procedures are the most common ones used for getting information out of the Whirlwind computer:

A. The contents of individual registers in the computer can be examined by making use of the switches and push-buttons on the control console. The information is displayed as binary numbers and, since the process is slow and awkward compared to the following ones, this method is seldom used.

B. Direct programming makes use of the select (si) instructions discussed in Chapters XI and XII. This procedure is potentially the most efficient in time and storage. However, it requires a greater proficiency in Whirlwind coding than is required by procedures C, D, and E. (See Chapter XII).

C. The library of subroutines contains a number of output routines. By specifying appropriate preset parameters and by entering the routines at different points, a great deal of flexibility can be obtained. A list and description of the output subroutines available may be obtained in the tape preparation room at the Digital Computer Laboratory. In general, subroutines are shorter but not as flexible or as easy to use as the pseudo-codes described under procedure E.

D. Post-mortem (fp) tapes have the advantage of not consuming any of the storage space occupied by the main program. Their principal disadvantages are that absolute addresses must be specified and that they are not practical to use when output is interspersed with computation unless the program run is controlled by a director tape (see Chapter XV). A more complete discussion of how fp tapes are specified and of the forms of output available is given in Chapters VI and XVII.

E. Pseudo-codes are fictitious instructions that have been invented to allow the programmer to request output through the use of a notation that may be easily learned. No knowledge of Whirlwind coding or of Whirlwind terminal equipment (beyond its existence) is required. The output request is translated during read-in and the necessary subroutines are compiled.

The output requests that are available for recording the contents of the MRA in printed form were described in Chapter V. It will be remembered

that these requests are characterized by three upper-case letters (MOA for the delayed printer, TOA for the direct printer, and SOA for the scope) following the lower-case letter i. They also contain a "sample number" which describes the form in which the output is to be recorded. The request, iFORMAT, which may be used to specify the layout or format to be followed when executing these requests, was also described in Chapter V. This request differs in form from the others in that it does not contain a sample number, but instead requires that additional information be provided by means of "parameters" occupying successive storage registers immediately following the output request.

There are available many other output requests, both for the CS computer and for Whirlwind I. Like the ones which have already been described, each of these is specified by typing a distinguishing combination of upper-case letters. Many of them, like iFORMAT, require also that suitable parameters be stored in the following registers. This chapter is devoted to a description of the remaining output requests.

II. Non-Interpreted Output Requests

All of the output requests described in Chapter V may be used in non-interpreted Whirlwind-coded routines simply by omitting the letter "i" which prefixes the request. When the prefix "i" is omitted, any accumulator reference is to the accumulator (AC) of Whirlwind instead of the MRA and, upon completion of the request, control will be transferred in the WWI (non-interpreted) mode.

Thus the request MOA + i.1234c will record on magnetic tape for delayed typing the number in the WWI accumulator (AC) as a decimal fraction with four digits to the right of the decimal point (no round-off). The special symbols (+, -, s, c, etc.) may be used in the same way as described in Chapter V.

Observe that since the number in the AC is always less than one in magnitude, then an MOA, TOA, or SOA instruction with no factor indicated to scale the contents of the AC before print out will always give a decimal fraction print-out. To print out the contents of the AC, C(AC), as an integer one need only insert the scale factor 2^{15} in his request. For example,

$$\text{MOA} + 12345 \times 2^{15} \text{c}$$

will record on magnetic tape for delayed printing the C(AC) multiplied first

by 2^{15} - that is, considered as an integer. Since $|C(AC) \times 2^{15}| < 32768$, the resulting integer can never contain more than five digits. On the other hand, if the programmer asks for fewer than five digits to the left of the decimal point, ambiguity may sometimes result if the request is followed precisely. This happens whenever the number to be printed out unexpectedly exceeds the requested range.

For example, if

$$C(AC) = +5178 \times 2^{-15} \quad (1)$$

and if the request were

$$TOA + 123 \times 2^{15} \text{ c} \quad (2)$$

then if the print-out followed the request exactly, the result would be $+517\downarrow$ with no indication of the error. To avoid this situation, the integer print-out (compiled in response to an MOA, TOA, or SOA request containing the single scale factor 2^{15}) always gives five digits to the left. Thus for case (1) above, the request (2) would produce the following printout:

$$+05178\downarrow$$

To avoid printing non-significant initial zeros the programmer need only write (see Chapter V):

$$TOA + i12345 \times 2^{15} \text{ c}$$

This request (irrespective of the number of digits requested) would produce the result

$$+5178\downarrow$$

where the initial zero has been suppressed.

Any "uninterpreted" output request with a scale factor other than 2^{15} , 2^0 , or 10^0 will be treated in the same manner as a generalized decimal number.

III. Automatic Scope Output Requests

As was indicated in Chapter V, the instructions SOA and iSOA are used in exactly the same way as the MOA (iMOA) and TOA (iTOA) instructions. SOA causes the contents of AC to be displayed on the scope, while iSOA is used to display the contents of MRA. SOA must be used while in the WWI mode of operation; iSOA while in the interpreted mode. The form of the display is specified with a sample number typed immediately following the capital letters SOA (with no intervening tabs or carriage returns). The conventions for forming the sample number are exactly the same as for the

sample numbers used with the typewriter output instructions. Any form obtainable with MOA(iMOA) or TOA(iTOA) is available with SOA(iSOA).

The terminating symbols "space", "tab", and "carriage return" have been given meanings for the oscilloscope which are similar in effect to the typewriter machine functions.

When a space is recorded on the scope the horizontal deflection is indexed to the right by the width of one character. Nothing is displayed in the area passed over. If during the execution of "space" or during the actual display of a number, the horizontal deflection should run off the right edge of the scope face, an arithmetic-overflow alarm will be generated. The scope output routines consider the scope to be 63 characters wide, so the programmer must arrange the display to keep it within this range.

For the purposes of the tab symbol, the scope face has been divided into four columns of equal width. The tab causes the horizontal deflection to be moved to the right to the beginning of the next available column. If a tab should be given while characters are being displayed in the last (fourth) column, a check-register alarm is generated.

The carriage return causes the vertical deflection to be indexed downward to the next line and the horizontal deflection to be reset, in general, to the left edge of the scope.* If a carriage return is given after the last available line has been used, its effect is to cause the camera to be indexed and to reset both the horizontal and vertical deflections to the upper left of the scope face.

The vertical bar, used as part of some numbers on the typewriter, is not displayed on the scope.

Single Characters

The following instructions may be used to obtain the display of a single character on the scope:

SOA c	iSOA c
SOA s	iSOA s
SOA t	iSOA t

* See COLUMN instruction for exception to this rule.

SOA .	iSOA .
SOA +	iSOA +
SOA -	iSOA -

FORMAT

The instructions FOR and iFOR may be used unchanged with SOA and iSOA.

FRAME

The instruction FRAME(iFRAME)* causes the camera to be indexed one frame, and if SOA or iSOA is also in use, it causes the deflections to be reset to the upper left-hand corner of the scope. If FRAME or iFRAME is used without an SOA instruction, it simply causes the camera to be indexed.

COLUMN

The instruction COLUMN (iCOLUMN)** may be used to facilitate displaying data in columns. This instruction causes the horizontal deflection to be moved to the right to the beginning of the next available column and the vertical deflection to be restored to the top of the scope. It also causes subsequent carriage returns to reset the horizontal deflection to the left edge of this column instead of to the left edge of the scope face. This behavior of carriage return will continue until either

- a) another COLUMN instruction is executed;
- b) a FRAME instruction is executed; or
- c) a carriage return causes the camera to be indexed.

Should a COLUMN instruction be given while characters are being displayed in the last column, that COLUMN instruction will behave exactly like a FRAME instruction.

Repeated Output Requests

It is common practice to use an output request without the sample number when exactly the same request as the preceding one on the manuscript is desired. For instance, in the program

```
iSOA+nl.2345c
-
-           no intervening
-           output requests
iSOA
```

the word iSOA will be converted to iSOA+nl.2345c. Remember, however, that

-
- * These may be shortened to FRA and iFRA
 - ** These may be shortened to COL and iCOL

this always gives exactly the same request as the immediately preceding one.
In the program

```

iSOA+n1.2345c
-
-
-
icp al
iCOL
al, -
-
iSOA

```

the word iSOA will have the effect of iCOL regardless of the sequence of instructions executed during program operation.

Summary

SOA	require sample number, cause display
iSOA	of C(AC) or C(MRA) (as appropriate) on scope.
FRA	Cause camera to index, and if SOA or iSOA is
iFRA	also in use, cause deflections to be reset to
	left top of scope face.
COL	Set deflection to top of next available column
iCOL	and cause all succeeding carriage returns to reset
	horizontal deflection to beginning of this column
	instead of to left of scope face.

No. of available columns (when using COL instruction or tab character) = 4.

No. of characters per line = 63 max.

No. of lines per frame = 36 max.

IV. Curve Plotting

A. Pseudo codes are available to facilitate the plotting of curves on the oscilloscope. The instruction SOC (Scope Output Curve) displays one point on the scope. The vertical coordinate of the point must be in AC ((for iSOC, in the MRA) at the time the instruction is given. The location in which the horizontal coordinate has been stored is specified as a parameter following the SOC instruction. Thus, the sequence

```

iSOC      or      SOC
al                al

```

will plot one point. Note that a_1 is an address* in CM showing where the x-deflection will be found; it is not the x-deflection itself.

B. Normally, the range of deflections on the scope is $-1 < x < +1$ and $-1 < y < +1$. In the absence of another specification, numbers will be plotted to this scale. When the numbers to be plotted fall outside this range or are non-uniformly located, we may wish to use another format. The instruction FOC (FOrmat Curves) permits us to do this.

Two variations of FOC must be distinguished. In the first of these, the origin (0,0) remains at the center of the scope but the scale is to be expanded (or contracted) to give a different maximum deflection. This is called the symmetrical case, and is specified using the pseudo-instruction FOC 1 (iFOC 1), followed by two parameters:

iFOC 1	FOC 1
x_{\max} y_{\max}	x_{\max} y_{\max}
}	}
double-length numbers	single-length WWI numbers

The execution of FOC 1 (iFOC 1) causes subsequent SOC (iSOC) instructions to plot points to the scale $-x_{\max} < x < x_{\max}$, $-y_{\max} < y < y_{\max}$ instead of the normal scale $x_{\max} = y_{\max} = 1$.

If it is desired to have the point (0,0) at some position other than the center of the scope (the asymmetric case), the instruction FOC 2 (iFOC 2) must be used. This instruction requires four parameters:

iFOC 2	FOC 2
x_{\min} x_{\max} y_{\min} y_{\max}	x_{\min} x_{\max} y_{\min} y_{\max}
}	}
double-length numbers	single-length WWI numbers

and causes subsequent SOC (iSOC) instructions to plot numbers to the scale $x_{\min} < x < x_{\max}$, $y_{\min} < y < y_{\max}$.

C. For some applications, it may be desired to plot x- and y-axes on the scope, in order to produce a more effective display. Two pseudo-instructions are provided which do this automatically. The instruction

* a_1 may not be a buffer, and the notation a_1+c may not be used here.

SUX (Scope Uncalibrated axes) requires two parameters:

iSUX	SUX
x_0 } double-length numbers	x_0 } single-length WWI numbers
y_0 }	y_0 }

and plots simply a set of rectangular axes centered about the point (x_0, y_0) .

The instruction SCX (Scope Calibrated axes) requires four parameters:

iSCX	SCX
x_0 } double-length numbers	x_0 } single-length WWI numbers
y_0 }	y_0 }
α } single-length WWI integers	α } single-length WWI integers
β }	β }

and plots a set of rectangular axes centered about (x_0, y_0) , with short calibration marks so placed that the x-axis is divided into α equal increments and the y-axis into β equal increments. The calibration marks always start from the origin of the coordinates. Note that α and β are always single-length WWI integers, even with the iSCX instruction. Neither α nor β may be less than 3.

For both the calibrated or uncalibrated axes, the position of (x_0, y_0) is determined in accordance with the last FOC (iFOC) specification that was executed.

V. DIB and DOB

The auxiliary drum is frequently used by programmers as an auxiliary storage medium. To facilitate the transfer of information to and from the drum, automatically assembled requests similar to the output requests are available. The drum is useful, of course, only for temporary storage of information which (later in the course of the program) will be read back into the computer. These are, therefore, not "output" requests in the same sense as the others that have been described; the information transferred to the drum is accessible only to the computer and not to the coder.

For this reason, the drum requests deal solely with computer words in their binary form. The transferred words may be WWI or CS computer numbers or instructions (or, indeed, any combination of these intermixed in any sequence). This is true because anything that is stored in the computer memory, regardless of the form in which it was originally obtained, appears simply as an array of binary digits occupying one or more storage

registers.

The request **DOB** (Drum Output Binary) or **iDOB** is used to effect the transfer of words from the computer memory to the drum. This request is followed by three parameters in the following order:

- a) Initial address in core memory
- b) Initial address of drum memory
- c) Number of words to be transferred

As an example, let us suppose that the contents of registers **a1** to **b1** (inclusive) in core memory are to be transferred to the drum, starting at drum register 12765. This will be accomplished by

⋮		⋮
iDOB		DOB
a1	or	a1
12765		12765
b1-a1+1		b1-a1+1
⋮		⋮

The contents of the core memory registers is not changed by **DOB** or **iDOB**.

Similarly, words may be transferred from the drum to the computer memory by using the request **DIB** (Drum Input Binary) or **iDIB**. The same three parameters, in the same order, must follow the input request. The contents of the drum registers is not changed by **DIB** or **iDIB**.

Note that, although **DOB** and **iDOB** do exactly the same thing (this is true also of **DIB** and **iDIB**), the uninterpreted form must be used in a **WWI** program (i.e., while **OUT**) and the interpreted form (including the **i**) must be used in a **CS** computer program (i.e., while **IN**).

VI Automatic Assembly of Output Requests

Each output request occupies one register at the point where it is written in the program. Of course, many of the output requests are followed by parameters which are separated from the request and from one another by tabs or carriage returns.* These parameters are no different from other storage words, and are stored in sequence in the registers following the one which contains the output request itself.

* Sample numbers are part of the output request, not separate parameters.

Besides occupying a register in the program each output request causes during read-in the automatic assembly of the routines necessary to execute the request. These routines are stored in the higher-numbered registers of core memory, outside the region used by the programmer. The request itself is translated into a Whirlwind I sp operation whose address part is determined by the location of the corresponding assembled routine.

The words of the program are stored in the locations specified by the programmer (normally, in successive core-memory registers starting at register 32). It is possible, therefore, that a long program may "overlap" the automatically-assembled routines, indicating that there is not enough room in the memory for both the program and the auxiliary routines that it requests. Such an overlap will be detected during read-in and a conversion post-mortem will result; there is no danger of operating a program in which this mistake has occurred. However, the programmer may wish to know before reading in his tape whether or not his program is too long. In order to determine this, he must know exactly how many registers will be occupied by the auxiliary routines requested by his program.

In determining this, two important facts must be taken into account. First of all, no matter how many times the same request appears in a program, the routines needed to execute that request will be included only once. Thus, if the request IMOA+1.23456c is used at five different places in the program, the routine for IMOA+1.23456c will be included once, not five times. Each of the occurrences of IMOA+1.23456c will be translated into an sp instruction to the same output routine.

Secondly, it is obvious that many requests differ only in detail, but otherwise are quite similar. For example, IMOA+1.23456c and ITOA+1.23456c request exactly the same form of output, and differ only in the output device selected. The routines which execute these requests are almost identical. If a program contains both of them, it would be wasteful to repeat those sections of coding which are common to the two routines. Consequently, the assembly procedure permits such common sections of coding to be shared by more than one routine, and includes the common sections only once. Thus, if IMOA+1.23456c and ITOA+1.23456c are both used in the same program, the number of registers required for the two routines is not obtained simply by adding together the number required for each of the routines alone.

The actual combined routine is shorter than this, because the common sections are not repeated.

The procedure for calculating the number of registers occupied by the automatically-assembled output routines is given below. Each different output request used must be taken into account, but repetitions of identical requests must be ignored.

Each request containing a sample number requires an "interlude" which occupies registers in storage. Each interlude consists of four registers if the sample number contains a scale factor (e.g., $x2^2 \times 10^{-3}$) and otherwise consists of three registers. If the request contains no sample number, then no interlude is required.

The various output requests which can be written have been divided into categories. Each category used in the program requires the appearance in core memory of an "entry block". The output request categories and the lengths of the associated entry blocks have been listed in columns 1 and 2 of Table 1. The third column of table 1 contains the numbers of the various "auxiliary blocks" required by each entry block. The length of each auxiliary block is tabulated in table 2.

The total number of registers in core memory occupied by the automatic output subroutines equals the sum of the number of registers required for interludes, entry blocks and auxiliary blocks. Each entry block and each auxiliary block must be counted only once in this sum.

For example, let the following output requests occur in a program.

- a) iTOA 12.34c
- b) MOA 12.34 x 2^2 c
- c) iTOA 34.56c
- d) MOA end
- e) iTOA 4.56c
- f) iFOR

Request c) can be eliminated because it is identical to request a).

Of the remaining requests the ones requiring interludes are a), b) and e). The interludes for requests a) and e) are 3 registers long and the interlude for request b) is 4 registers long. The total length of the interludes is then

$$I = 3 + 4 + 3 = 10$$

Requests a) and e) all fall in the same category: namely iTOA, no scale factor, not 2^{15} . Request b) falls in the category: MOA, scale factor. Requests d) and f) constitute separate categories. Table 1 gives the lengths of the entry blocks:

<u>Category</u>	<u>Entry Block Length</u>
iTOA, no scale factor	19
MOA, scale factor, not 2^{15}	24
MOA, end	15
iFOR	12

The total length of the entry blocks is:

$$E = 19 + 24 + 15 + 12 = 70$$

Table 1 also gives the numbers of the auxiliary blocks required for these entry blocks.

<u>Category</u>	<u>Auxiliary Block Numbers</u>
iTOA, no scale factor	0,1,6,25,52
MOA, scale factor, not 2^{15}	0,1,6,25,52
MOA, end	-----
iFOR	0,25,28

Eliminating duplicated auxiliary block numbers gives the following list of blocks required:

0, 1, 6, 25, 28, 52.

Table 2 gives the length of these blocks

<u>Auxiliary Block Number</u>	<u>Length</u>
0	30
1	42
6	17
25	16
28	32
52	218

The total length of the auxiliary blocks is

$$A = 30 + 42 + 17 + 16 + 32 + 218 = 355$$

The total number of registers required is

$$I + E + A = 10 + 70 + 355 = 435$$

If the CS computer is also used the number of registers required for the programmed arithmetic subroutine (see Chapter XIII) must be added to the above number.

Table 1

Category Description	Entry Block Length	Auxiliary Block Numbers
SOA, no scale factor	9	0, 1, 8, 15, 25
SOA, scale factor, not 2^{15}	26	0, 1, 6, 15, 25, 52
iSOA, no scale factor	22	0, 1, 6, 15, 25, 52
iSOA, scale factor	30	0, 1, 6, 15, 25, 52
SOA, scale factor, 2^{15}	9	0, 1, 15, 18, 25
SOA .	3	15
iSOA .	4	15
SOA -	3	15
iSOA -	4	15
SOA +	3	15
iSOA +	4	15
SOA s	2	15
iSOA s	4	15
SOA t	2	15
iSOA t	4	15
SOA c	2	15
iSOA c	4	15
MOA, no scale factor	7	0, 1, 8, 25
MOA, scale factor, not 2^{15}	24	0, 1, 6, 25, 52
iMOA, no scale factor	20	0, 1, 6, 25, 52
iMOA, scale factor	28	0, 1, 6, 25, 52
MOA, scale factor, 2^{15}	7	0, 1, 18, 25
MOA .	7	--

Table I continued

Category Description	Entry Block Length	Auxiliary Block Numbers
iMOA .	8	--
MOA -	7	--
iMOA -	8	--
MOA +	7	--
iMOA +	8	--
MOA s	7	--
iMOA s	8	--
MOA t	7	--
iMOA t	8	--
MOA c	7	--
iMOA c	8	--
MOA wend	15	--
iMOA end	16	--
TOA, no scale factor	6	0, 1, 8, 25
TOA, scale factor, not 2^{15}	23	0, 1, 6, 25, 52
iTOA, no scale factor	19	0, 1, 6, 25, 52
iTOA, scale factor	27	0, 1, 6, 25, 52
TOA, scale factor, 2^{15}	6	0, 1, 18, 25
TOA .	6	--
iTOA .	7	--
TOA -	6	--
iTOA -	7	--
TOA +	6	--
iTOA +	7	--

Table I continued

Category Description	Entry Block Length	Auxiliary Block Numbers
TOA s	6	--
iTOA s	7	--
TOA t	6	--
iTOA t	7	--
TOA c	6	--
iTOA c	7	--
FOR	11	0, 25, 28
iFOR	12	0, 25, 28
DOB	4	46
iDOB	6	46
DIB	4	46
iDIB	6	46
FOC 1	32	13, 14, 21, 22
iFOC 1	26	2, 11, 12, 23, 24
FOC 2	48	13, 20, 21, 22
iFOC 2	40	3, 11, 12, 19, 23
SOC	3	14, 21, 22
iSOC	3	2, 12, 23
SUX	6	13, 14, 21
iSUX	5	11, 12, 24
SCX	77	13, 14, 22
iSCX	76	11, 12, 24

Table I continued

<u>Category Description</u>	<u>Entry Block Length</u>	<u>Auxiliary Block Numbers</u>
FRA	3	--
iFRA	4	--
COL	6	--
iCOL	8	--

Table IIAuxiliary Block NumberLength

0	30
1	42
2	8
3	12
6	17
8	19
111	68
12	54
13	71
14	21
15	104
18	44
19	10
20	26
21	15
22	20
23	19
24	6
25	16
28	32
46	43
52	218

CHAPTER XVII: GENERALIZED POST-MORTEMS

I. Introduction

- A. Non-Selective Programs
- B. Selective Programs
- C. Trace Programs

II. Post-Mortems of Type A

- A. Lights
- B. Programmed Arithmetic Post-Mortems
- C. Scope Post-Mortems

III. Post-Mortems of Type B

- A. fp tapes

IV. General Operation of Post-Mortem Programs

I. Introduction

One of the most difficult, time-consuming aspects of the solution of a complex problem on a high-speed computer is the process of detecting and removing errors from a coded program. This is due to the lack of a flexible and rapid communication link between the computer and user. In many installations the only link provided is a facility for operating the computer so that it stops after each instruction. The operator can then manually inspect the contents of any register in the computer before proceeding to the next instruction. This is probably the most flexible and, in terms of computer utilization, least efficient method of detecting mistakes.

A typical approach to trouble-shooting is to hypothesize the cause of a mistake and check the hypothesis. Often the difference between a good trouble shooter and a poor one is the ability to imagine a number of ways a mistake could have been caused. Forming these hypotheses often requires a considerable amount of thought and checking them may take considerable time. Since random access to the computer is not generally possible in many installations, this procedure often results in the computer standing idle while the user thinks (or worse yet, while he does not think!), or in a considerable delay in trouble shooting while waiting for free time on the computer.

For these reasons, it is more desirable to provide the coder with a printed record which he can study at his leisure without sacrificing computer time. A record obtained after a program has stopped is called a post-mortem (since the program has effectively "died"). Post-mortems are best obtained by the computer itself if at all possible, since the computer is best equipped to interrogate its own memory. Its output equipment, however slow, is faster than human transcription.

Programs which collect and record post-mortem information will be called post-mortem programs, or, briefly, PM programs. There are three general types of such programs which will be described below.

A. Non-selective Programs

Programs of this type print a small amount of predetermined information which will presumably be of use in detecting most errors. The programmer has no control over what information will be recorded and

hence, in the interests of efficiency only, the most commonly used data are generally recorded. It is very desirable that the computer be designed so that all such information can be made automatically available to the programmer. Since this is not the case with Whirlwind I, a certain amount of this data is routinely recorded manually by the computer operator.

B. Selective Programs

Programs of this type allow the programmer to specify the information desired and the form of output. These are most useful if the programmer can anticipate the information he will need for error detection.

C. Trace Programs

Programs of this type essentially simulate, instruction-by-instruction, the operation of the computer and come closest to attaining the flexibility of actual manual operation. Normal operation of the computer is simulated by interpretive routines which, in addition, record the contents of certain registers, specified by the programmer, after each instruction, or after each instruction of a certain class. In sophisticated programs of this type the programmer can switch out of the tracing routine into normal operation and back, so that only specified sections of the program are traced. This mode of operation is often called "trapping".

In addition to these three types of post-mortem programs, it is possible to detect certain purely clerical errors in the process of reading in and translating a tape. This is called a "conversion post-mortem", and the one used in the Comprehensive System is described in Chapter XIV. Only those post-mortems which give information about the actual operation of a program will be described here.

The great majority of programming and coding mistakes can be detected fairly quickly with the aid of information obtained from post-mortem programs of type A. Large quantities of data are rarely required and only a very obscure mistake can warrant large memory printouts or extensive tracing. The desirability of the different types of post-mortem facilities hinges on the relative worth of computer time as compared to human time. This worth should not always be based on obvious

economic factors. Post-mortems of type A plus enough painfully detailed thinking should in theory enable every programming or coding mistake to be detected, just as enough checking of any program should eliminate all mistakes before it is ever run. However, there are valid reasons, both economic and psychological, for placing part of the burden on the computer. For this reason it may be desirable to have available programs which can produce large quantities of data on request or which can trace a lengthy program even with a considerable sacrifice in speed. The advisability of programs of this type also depends upon the characteristics of the computer and input-output equipment with which they are used.

The Comprehensive System has available only programs of types A and B, which will be described in more detail in the remainder of this chapter.

II. Post-Mortems of Type A

A. Lights

Whenever the computer stops, certain information is displayed by lights on the computer console. The computer can be stopped by any one of 5 alarms, by an automatic stop instruction, or by manually pressing the stop button. On the performance request, provision is made for indicating which of these occurred. Blanks are also provided for the recording of the contents of 9 registers which are available from lights on the console. These blanks are preceded by the abbreviations listed in Table I with their meanings.

Table I

PC	Program Counter
AC (partial)	Accumulator
AR	"A" Register
PAR	Parity Register
CS	Control Switch
IOS	In-Out Switch
AC (carry)	Accumulator Carry Bits
Aux. Drum	Auxiliary Drum
GSR	Group Selection Register
SAR	Storage Address Register
Buf. Drum	Buffer Drum
GSR	Same as Aux. Drum
SAR	

In general, unless the programmer specifically requests otherwise, only those blanks will be filled which are immediately pertinent to the reason for the stop. All numbers recorded are octal, since they occur on the console in binary form and it is undesirable to spend the time required for conversion to decimal form. The number in the PC is always noted and usually specifies the location of the next instruction. The following examples will serve to clarify some common cases.

Example:

Arithmetic check alarm

PC	42	} all octal
AC	0.44572	
AR	0.57123	
PAR	ca 55	

This information indicates that an overflow occurred on the instruction in register 41, which by inspection of the program proved to be

su 60

where the number 0.57123 was stored at location 60. The original contents of the accumulator can be recovered by adding 0.57123 (which is in the A register) to 0.44572 giving

$$\begin{array}{r} 0.57123 \\ 0.44572 \\ \hline 1.23715 \end{array}$$

If the overflow was caused by an add instruction, the AR must be subtracted from the AC assuming the sign position to contain a significant binary digit.

Example:

Divide error alarm

PC	44
AC (partial)	0.65040
AR	0.11315
PAR	0.11315
CS	dv
AC (carry)	0.05045

The divide error occurred during the performance of the instruction at location 43. The AR and PAR contain the divisor. The

initial contents of the AC can be recovered by adding the contents of the AC (partial), AC (carry) and the AR, dividing the result by 2 and adding the AR again. In this case

$$\begin{array}{r}
 0.65040 \\
 0.05045 \\
 \hline
 0.11315 \\
 2 \overline{) 1.03422} \\
 \hline
 0.41611 \\
 0.11315 \\
 \hline
 0.53126
 \end{array}$$

which is the original dividend. The divide error was justified since

$$0.11315 \leq 0.53126$$

The in-out switch in general contains the address part of the last si instruction which was executed before the stop. If magnetic tape or PEIR was last selected, then 400 octal is added to the IOS when the computer stops. This is to insure deselection of the free-running equipment at the time of the stop.

The control switch is, in general, either clear or holds the binary code for the Whirlwind operation on which the computer stopped.

The AC (carry) contains unperformed carries (if any) and is generally useful only for the divide-error alarm.

The drums are divided into groups, each group capable of storing 2048 words (addresses 0000-3777 octal) There are 12 groups on the auxiliary drum (00-13 octal) and 7 groups on the buffer drum. For both drums the GSR denotes the group selected and the SAR the register within the group.

A program alarm most often occurs when information from either the photoelectric reader or a magnetic tape unit is arriving at the in-out register faster than read instructions are removing it.

Inactivity alarms usually occur when IN-OUT equipment is unable to respond to a read or record instruction, because it has not been selected by a proper si instruction.

Example:

Inactivity alarm

PC 52

CS rd

IOS 707

Since the auxiliary drum had been selected to record, the rd instruction was impossible. The alarm will occur on the second rd instruction (or second word of a bi instruction) after the improper si.

B. Programmed Arithmetic Post-Mortem (PAPM)

If the computer stops while performing an interpreted instruction, the console lights will in general have little meaning. Illegalities in this case usually lead to a programmed check alarm with the PC = 3764. In this case the cause of the alarm can be deduced from the contents of the AR. Table 2 lists some contents of the AR which can occur.

Whenever interpreted instructions are used, a programmed arithmetic post-mortem (PAPM) should be requested. A sample PAPM is given below with explanation of its various sections.

Example:

- 1) 191-25-62 Jones 0622.1 11-10-55
(24,6) PAPM
- 2) Stopped at 279 279|iex493+c 499|-.12345678|+7 MRA|.12345678|+22
- 3) 1626 | b|.123456789|+32 1b|-.987654321|-2 2b|.135798642|-0
- 4) 1635 | 0|0,10 1|||3,12 2|0,0 3|0,7 4|6,6
- 5) 509|icp606 615|isp285 320|isp221 246|icp255 274|icp278

Section 1)

This section contains the last title read into the computer followed by the time to a tenth of a minute in 24-hour notation and the date. The next gives the last number system used by the programmer.

Section 2)

This section gives the location of the register at which the computer stopped, the contents of this register and the contents of the register referred by the instruction at this location. The contents of the MRA at the time of the stop is recorded at the end of this line.

In this example, the computer stopped while performing the instruction in register 279, which was iex 493+c. The index of the

counter most recently selected was 3 so that the effective address referred to was 499. Register 499 (and 500) held the generalized decimal number $-.12345678 +7$.

The number in the MRA had an exponent which was too large for the number system used ($10^{+22} > 2^{63}$), which caused the stop.

Section 3)

The contents of all buffers provided by the PA routine are listed as 9 digit numbers. The number at the beginning of the section is the decimal address of the first register of the zeroth buffer. If buffers are not called for by the program, this section will not be printed.

Section 4)

The contents of the index and criterion register of all counters called for by the program are printed in this section. The number at the beginning of the section is the decimal address of the index register of the zeroth counter. The number of the counter most recently used is followed by two extra vertical bars. In the example, counter 1 was most recently used, which has index 3, which checks with the information in section 2. If no counters are called for by the program, this section will not be printed.

Section 5)

The PA routine keeps a record of the locations of the registers containing the 5 most recent isp or icp instructions which resulted in changes of control or jumps. The contents of these registers at the time of stop are listed in this section. The contents of the register which contained the most recent jump instruction is listed last, the contents of register which contained the previous jump instruction is listed next to last, etc. If the contents of these registers has been changed since the jump instruction occurred, the new contents will be printed. If less than 5 jumps have occurred, only those which have occurred will be printed. If no jumps have been executed by the interpreted routine, the phrase "no jumps" will be printed. An interpreted automatic output routine will return to the main program by an isp instruction which will be listed in this table. Such jumps can usually be identified by a large address preceding the vertical bar. Table II will help in the interpretation of some cases which arise.

Table II

WWI Lights (Octal)	Operations Which Cause Alarm	Reason for Stopping	Comments on PAPM
check alarm PC = 3764 AR = 3430	di, ck(dh), md	These are not legal interpreted instructions	All instructions printed as WWI instructions
check alarm PC = 3764 AR = 3400	Any of 17 operations which refer to a counter	Counter instruction cannot be executed if counter block has not been requested	Register referred to is not printed; instruction is printed with WWI operation code
check alarm PC = 3764 AR = 3453	iad, isu, imr, idv iadc, isuc, imrc, idvc	The instruction refers to a non-zero unscalfactored number	Number referred to is printed as two octal fractions
check alarm PC = 3764 AR = 3503	its, iex, itsc iexc	MRA contains a number whose exponent is too large	MRA is printed as decimal number; largest exponent allowable is 0.3×2^j where $(30-j, j)$ floating point system is used
check alarm PC = 3764 AR = 3276 (without buffer block)	isc	Address section of isc instruction is too large	
check alarm PC = 3764 AR = 3167 (with buffer block)	isc	Address section of isc instruction is too large	
check alarm PC = 3764 AR = 3555	isp, icp, ict, ispc	Program was stopped manually	If an illegal interpreted instruction is referred to, it is printed as Whirlwind instruction
divide error alarm PC = 3612	idv, idvc	A division by zero was attempted	
	sp	Computer stopped in non-interpreted part of program. See WWI lights	Computer exited from PA routine by this sp instruction

C. Scope Post-Mortem

Because of the availability of very high-speed output in the form of film records of oscilloscope displays, there is available a special post-mortem of type A which records all of the contents of core memory in the form of octal fractions on film. This record is called a scope post-mortem and can be obtained by checking the appropriate box on the performance request.

This post-mortem has the disadvantage that all transcription into instruction or decimal numbers must be done by the programmer. Table III giving the octal equivalents of the Whirlwind and interpreted operation codes is included here. Reference may be made to Chapters X and XIII for instructions on conversion of single and double length octal numbers into decimal.

Table III

si	0.00	not used	cs	1.04	icd
not used	0.04	itsc	ad	1.10	icx
bi	0.10	iexc	su	1.14	ita
rd	0.14	icac	cm	1.20	icp
bo	0.20	icsc	sa	1.24	its
rc	0.24	iadc	ao	1.30	iex
sd	0.30	isuc	dm	1.34	ica
not used	0.34	imre	mr	1.40	ics
ts	0.40	idvc	mh	1.44	iad
td	0.44	ispc	dv	1.50	isu
ta	0.50	isc	slr	1.54	imr
ck	0.54	icr	slh	1.55	imr
ab	0.60	ict	srr	1.60	idv
ex	0.64	iat	srh	1.61	idv
cp	0.70	iti	sf	1.64	isp
sp	0.74	sp	clc	1.70	} not used
ca	1.00	ici	clh	1.71	
			md	1.74	not used

III. Post-Mortems of Type B

Post-mortems of type B may be obtained in the Comprehensive System by means of post-mortem request tapes (fp tapes). This facility allows the programmer to request that the contents of several continuous ranges of registers be recorded in any of several forms on one or more output units.

It is essential that post-mortem request tapes begin with the characters, fp. No other Flexowriter characters (visible or invisible on the printed manuscript) can occur before or between them. The identifying characters (fp) can be followed by a tape number and a tape description (see section II,A in Chapter XV). Requests for post-mortem information may follow and are written in the following form

```
5-346 wi 5-512
      639 of 679
```

The abbreviations, wi and of, specify the form in which results will be recorded. The possible forms and their abbreviations are listed in Table IV.

Registers on the auxiliary drum can be requested as indicated above, i.e., by giving the group followed by a dash and the location within the group or by a decimal integer obtained by multiplying the group number by 2048 and adding the location within the group. Thus, the first request above could also be written in the equivalent form

```
10586 wi 10752
```

The contents of core memory are copied onto auxiliary drum group 0 before the operation of an fp tape, so that requests for the contents of registers in the range 1 to 2047 (inclusive) are actually taken from the corresponding registers of drum group 0 and are hence equivalent to requests for the contents of registers in the range 0-1 to 0-2047. Register 0 (or 0-0) cannot be requested.

Requests can also be written in the form

```
361 wi 469 gd 480 ii 619.
```

This is taken to be equivalent to

```
361 wi 469
469 gd 480
480 ii 619
```

Table IV

- ii interpreted instructions. The three-letter abbreviation for the interpreted operation corresponding to the left five bits is printed followed by the address referred to by the instruction. If there is no legal interpreted instruction corresponding to the left five bits, the abbreviation for the corresponding Whirlwind instruction is printed.
- gd generalized decimal numbers. The contents of consecutive pairs of registers within the range specified are printed as (30-j,j) numbers. If an odd number of registers is requested, the last register is disregarded.
- of octal fractions. The contents of the registers requested are recorded as a binary digit followed by a decimal point and five octal digits representing consecutive groups of three binary digits, giving a representation of the 16 binary digits of a register.
- wi Whirlwind instructions. The left five bits of the register contents are recorded as a two-letter combination according to the mnemonic form of the Whirlwind operation code.
- di decimal integers. The contents of the registers requested are recorded as a plus or minus sign (according as the sign digit of the register is 0 or 1) followed by at most five decimal digits representing the magnitude of the binary integer contained in the register.
- df decimal fractions. The contents of the registers requested are recorded as a plus or minus sign (according as the sign digit of the register is 0 or 1) followed by a decimal point and five decimal digits representing the magnitude of the binary fraction contained in the register.

Note that the contents of registers 469 and 480 are recorded twice in different forms. An exception would arise if a request for generalized decimal (30-j,j) numbers covered an odd number of registers, in which case the final register would not be recorded as part of the gd request. If each request begins with the proper register, the proper results will be obtained.

The output unit desired is specified by one of the three-letter abbreviations del, dir, sco, which cause succeeding requests to be recorded on a delayed typewriter (via magnetic tape) the direct typewriter or on film (via the oscilloscope). If these are placed within a request, they are considered to have preceded the request. For example, the request

741 ii 839 wi 841 gd sco 858

is equivalent to

741 ii 839

839 wi 841

sco

841 gd 858

Use of the direct typewriter should be avoided since computer time is required for typing. In the absence of any specifically designated output unit, del is assumed.

The three letter abbreviations dec and oct control the way in which the addresses in succeeding requests are interpreted and the form in which the contents of the registers requested are recorded. Requests following dec or oct are assumed to be written in decimal or octal representation, respectively, and the output addresses are recorded in the corresponding number system. If no specific designation is made, dec is assumed.

Subject to the limitations given in the following paragraph, an fp tape may consist of as many requests as desired on any output unit or units. The last request must be followed by two vertical bars,

An fp tape may not contain more than 169 requests or 1427 characters (not including title). The title or identifying information may not consist of more than 192 characters (not including the letters fp). Since no check is made by the fp program to determine if these limits are

exceeded, normal operation will be disrupted in apparently illogical fashion in these cases. The limits are sufficiently high so that this is very unlikely and seldom need be considered. In case of doubt, two or more fp tapes may be used in succession.

Extra characters or spaces are ignored provided they do not form any of the above recognizable abbreviations or intervene between characters forming an abbreviation or number. For example, the requests

DEL Foursscore 741 and Gd 894
seven Ye Ars 861 ago II 862

are incorrect only because of the underlined sco which will cause the results to appear on the scope instead of the delayed typewriter and because of the lower case between the G and d of the abbreviation gd.

Exceptions to the above are the backspace, which will cause read-in of an fp tape to stop, and color shift and delete, which may occur anywhere on post-mortem tapes.

Post-mortem requests for a single block of registers on one drum group can be made by entering appropriate information in the selector and insertion panels (see Chapter XV). In particular, the requested information may be punched by this means on Flexowriter tape for direct read-in to the computer. This should be done only with the cooperation of someone familiar with such punch-outs since a START AT block must be manually punched at the end of these tapes before they can be read into the computer.

IV. General Operation of Post-Mortem Programs

All of the functions of the CS system and in particular the post-mortem routines are controlled by the utility control program. When the read-in button is pressed, the contents of core memory are transferred unaltered to auxiliary drum group 0 and the utility control program is copied into core memory from auxiliary drum group 11. If the utility control program determines that post-mortem request information is present, (either in the manual intervention registers or in the form of an fp tape) buffer drum group 7 is sum checked and brought into core memory if correct (otherwise the conversion and post-mortem routines are located on tape unit 0 and read into the appropriate groups of the buffer drum). Control is then transferred to the appropriate entry point in

the post-mortem program which proceeds with the execution of the request. If an fp tape is used and the following registers of drum group 0 contain the indicated information, a PAFM is automatically given without request.

2046, ta 2019

2018, ~~ao~~ 2019

1877, srh

If these conditions are not met and the fp tape contains requests for either interpreted instructions or generalized decimal numbers, the line

NO PA

is printed. A +0 in a certain register of the PA dispatcher will cause the line

PA unused

to be printed.

After the post-mortem information has been recorded, the contents of drum group 0 is restored to core memory.

WHIRLWIND I INSTRUCTION CODE

The complete WWI instruction code is given below in tabular form. The notations used in the table, together with their meanings, are as follows:

- | | | |
|---|---|---|
| <p>---- = remains unchanged
 IOR = In-Out Register
 PC = Program Counter
 AC = Accumulator, $AC(i) = \text{digit } i \text{ of } AC, 0 \leq i \leq 15$
 BR = B-Register, $BR(i) = \text{digit } i \text{ of } BR, 0 \leq i \leq 15$
 AR = A-Register, $AR(i) = \text{digit } i \text{ of } AR, 0 \leq i \leq 15$
 ρ = round-off from BR; if $BR(0) = 1, \rho = 2^{-15}$
 CM = Core memory</p> | <p>x = address of a storage register $0 \leq x \leq 2047$
 n = positive integer $0 \leq n \leq 511 \text{ mod } 32$
 SAM = Special Add Memory
 $C(i)$ = original contents of register i
 $C(i)$ = original contents of digit i of register i
 $F(\cdot)$ = fractional part of the quantity in $\{\cdot\}$
 $\{ \cdot \}$ = integral part of the quantity in $\{\cdot\}$
 --- = becomes</p> | <p>$x(i)$ = digit i of register $x, 0 \leq i \leq 15$
 $k-l$ = digits k thru l of register $x, 0 \leq k < l \leq 15$
 $AC+BR$ = the composite 32 digit register (including sign) composed of the AC and BR taken in that order
 $(AC+BR)(i)$ = digit i of $(AC+BR), 0 \leq i \leq 31$
 \oplus = Boolean "exclusive or" operator
 \otimes = Boolean "and" operator</p> |
|---|---|---|

Instruction	Function	Binary Code	Dec. Equiv.	AC	Final Contents of	AR	SAM	x	Comments	Time
si pqr	select in-out unit or stop the computer	00000	0	----	----	----	----	----	The unit selected is designated by the digits pqr, and is started. <u>si</u> will stop the computer. <u>si</u> is a "conditional" stop. Program alarm possible if <u>si</u> selects Magnetic or Photo Electric Tape Reader without necessary <u>rd's</u> .	31 μ sec.
bi x	block transfer in (n words to CM)	00010	2	x+n	----	x	----	first word of block	$n \cdot 2^{-15}$ must be stored in AC at the time the computer executes the <u>bi</u> . If no one word will be read but not transferred. It is simply discarded.	For drum 8 msec. average and 16 msec. max. for first word. 32 μ sec. for ea. additional word.
rd x	read	00011	3	C(IOR)	----	C(IOR)	----	----	After word is transferred from IOR to AC, the IOR is cleared. The address of <u>rd</u> has no significance.	16 μ sec.
bo x	block transfer out (n words from CM)	00100	4	x+n	----	x	----	----	$n \cdot 2^{-15}$ must be stored in AC. If $n=0$, no recording will take place.	same as for <u>bi</u> .
rc x	record	00101	5	----	----	----	----	----	If <u>rc</u> is used as a display instruction, the IOR is cleared.	20 μ sec.
sd x	sum digits	00110	6	$C_1(AC) \oplus C_1(x)$ ---- AC(i) $i = 0, \dots, 15$	----	C(x)	clear	----	Adds digits without carry \oplus 0 1 0 0 1 1 1 0	21 μ sec.
ts x	transfer to storage	01000	8	----	----	----	----	C(AC)		24 μ sec.
td x	transfer digits	01001	9	----	----	----	----	x(0-4) unchanged x(5-15) = AC(5-15)		32 μ sec.
ta x	transfer address	01010	10	----	----	----	----	x(0-4) unchanged x(5-15) = AR(5-15)	<u>ta</u> normally follows an <u>sp</u> , <u>cp</u> , <u>sf</u> , or <u>so</u>	32 μ sec.
ck x	check	01011	11	----	----	----	----	----	Computer stops on "check-register alarm" if $C(AC) \neq C(x)$. (Note that $0 \neq -0$.)	24 μ sec.
ab x	add BR	01010	12	C(BR)+C(x)	----	C(x)	clear	C(BR)+C(x)	possible Arith. Overflow Alarm if $ C(x)+C(BR) \geq 1$ <u>am</u> <u>x</u> or <u>ah</u> 16 puts C(AC) into BR	32 μ sec.
ex x	exchange	01101	13	C(x)	----	C(x)	----	C(AC)	<u>ex</u> 0 will clear AC without clearing BR	32 μ sec.
cp x	conditional transfer control (conditional program)	01110	14	----	----	y+1 (digits 5-16)	----	----	If $C(AC) \geq +0$ proceed to next instruction. If $C(AC) \neq -0$, execute <u>cp</u> <u>x</u> . <u>y</u> is location of <u>cp</u> <u>x</u> .	16 μ sec.
sp x	transfer control (sub-program)	01111	15	----	----	y+1 (digits 5-16)	----	----	'Take next instruction from register <u>x</u> . PC \rightarrow <u>x</u> ' <u>y</u> is location of <u>sp</u> <u>x</u>	16 μ sec.
ca x	clear and add	10000	16	$C(x)+C(SAM)2^{-15}$	clear	C(x)	clear	----	possible Arith. Overflow Alarm if $ C(x)+C(SAM)2^{-15} = 1$	24 μ sec.
cs x	clear and subtract	10001	17	$-C(x)+C(SAM)2^{-15}$	clear	C(x)	clear	----	possible Arith. Overflow Alarm if $ -C(x)+C(SAM)2^{-15} = 1$	24 μ sec.
ad x	add	10010	18	C(AC)+C(x)	----	C(x)	clear	----	possible Arith. Overflow Alarm if $ C(AC)+C(x) \geq 1$	24 μ sec.
su x	subtract	10011	19	C(AC)-C(x)	----	C(x)	clear	----	possible Arith. Overflow Alarm if $ C(AC)-C(x) \geq 1$	24 μ sec.
cm x	clear and add magnitude	10100	20	$ C(x)+C(SAM)2^{-15} $	clear	C(x)	clear	----	possible Arith. Overflow Alarm if $ C(x)+C(SAM)2^{-15} = 1$	24 μ sec.
sa x	special add	10101	21	$F\{C(AC)+C(x)\}$	----	C(x)	----	if $ C(AC)+C(x) \geq 1$ or 0	Sign of SAM determined by sign of overflow. Previous contents of SAM cleared without alarm.	24 μ sec.
ao x	add one	10110	22	$C(x)+(1x2^{-15})$	----	C(x)	clear	$C(x) + (1x2^{-15})$	possible Arith. Overflow Alarm if $ C(x)+(1x2^{-15}) = 1$	32 μ sec.
dm x	difference of magnitude	10111	23	$ C(AC)-C(x) $	C(AC)	C(x)	clear	----	if $ C(AC) = C(x) $ result is -0	24 μ sec.
mr x	multiply and round-off	11000	24	$C(AC) \cdot C(x) + \rho$	clear	C(x)	clear	----	Sign of AC is determined by sign of product.	36-43 μ sec.
mh x	multiply and hold	11001	25	$C(AC) \cdot C(x)$ (digits 1-15)	$C(AC) \cdot C(x)$ (digits 16-30) BR(0) \rightarrow 0 BR(15) \rightarrow 0	C(x)	clear	----	Sign obtained same as for <u>mr</u> . Result in (AC+BR) is a double register product.	Same as for <u>mr</u>
dv x	divide	11010	26	± 0 (sign of quotient)	$\frac{ C(AC) }{ C(x) }$ (16 digits)	C(x)	clear	----	Divide Error Alarm if $ C(AC) > C(x) $. Arith. Overflow Alarm if $ C(AC) = C(x) \neq 0$ and <u>dv</u> <u>x</u> is followed by <u>slr</u> 15. If $C(AC) = C(x) = 0$, the quotient is 0.	73 μ sec.
slr n	shift left and round-off (n places)	11011 0	27	$F\{C(AC+BR)2^n\} + \rho$ (n taken mod 32)	clear	----	clear	----	possible Arith. Overflow Alarm if $ F\{C(AC+BR)2^n\} + \rho = 1$. The sign digit is not shifted. Digits shifted out of AC are lost. Negative numbers are complemented in AC before shifting and after rounding off. Digit 6 of <u>slr</u> n must be zero.	16-41 μ sec.
slh n	shift left and hold (n places)	11011 1	27	$F\{C(AC+BR)2^n\}$ (n taken mod 32)	$F\{C(AC+BR)2^n\}$ (digits 16-(31-n)) BR(j) \rightarrow 0 $j=16-n, \dots, 15, n>0$	----	clear	----	The sign digit is not shifted. Negative numbers are complemented in AC before and after the shift. Digit 6 of <u>slh</u> n must be a one.	Same as for <u>slr</u>
srr n	shift right and round-off	11100 0	28	$C(AC)2^{-n} + \rho$ (n taken mod 32)	clear	----	clear	----	possible Arith. Overflow Alarm on <u>srr</u> 0 (this instruction simply causes roundoff and clears BR). The sign digit is not shifted. Negative numbers are complemented in AC before shifting and after rounding off. Digit 6 of <u>srr</u> n must be a zero.	Same as for <u>slr</u>
srh n	shift right and hold	11100 1	28	$C(AC)2^{-n}$ (n taken mod 32)	$ C(AC+BR)2^{-n} $ (digits 16-31)	----	clear	----	The sign digit is not shifted. Negative numbers are complemented in AC before and after the shift. Digit 6 of <u>srh</u> n must be a one.	Same as for <u>slr</u>
sf x	scale factor	11101	29	$C(AC+BR)2^n$ (digits 1-15)	$C(AC+BR)2^n$ (digits 16-(31-n)) BR(j) \rightarrow 0 $j=16-n, \dots, 15, n>0$	$n \cdot 2^{-15}$	clear	$n \cdot 2^{-15}$ $x(5-15)$	X(0-4) unaffected. n is such that $\frac{1}{2} \leq C(AC+BR)2^n < 1$; if $C(AC+BR) = 0$, then $n = 33$. Negative numbers are complemented in AC before and after the multiplication.	33-81 μ sec.
clc n	cycle left and clear	11110 0	30	$(AC+BR)(n+1)_{32}$ \rightarrow AC(i) $i = 0, \dots, 15$	clear	----	----	----	Sign digit is shifted with all other digits. Digits shifted left out of AC 0 are carried around into BR 15. No roundoff. No complementing of AC either before or after the shift. Digit 6 of <u>clc</u> n must be a <u>msz</u> . Instruction <u>clc</u> 0 clears BR without affecting AC.	Same as for <u>slr</u>
clh n	cycle left and hold	11110 1	30	$(AC+BR)(n+1)_{32}$ \rightarrow AC(i) $i = 0, \dots, 15$	$(AC+BR)(n+1)_{32}$ \rightarrow BR(i) $i = 0, \dots, 15$	----	----	----	Sign digit is shifted with all other digits. Digits shifted left out of AC 0 are carried around into BR 15. No complementing of AC either before or after the shift. Digit 6 of <u>clh</u> n must be a <u>msz</u> . Instruction <u>clh</u> 0 does nothing.	Same as for <u>slr</u>
md x	multiply digits	11111	31	$C_1(AC) \otimes C_1(x)$ \rightarrow AC(i) $i = 0, \dots, 15$	----	-(Final AC)	----	----	Multiplies digits with no carry \otimes 0 0 1 0 0 0 1 1 1	24 μ sec.

* In operations mr, mh, dv, slr, srr, sf, the C(BR) is assumed to be the magnitude of the least significant part of AC + BR. For the ab and dm operations, the BR is treated just as any storage register.