Massachusetts Institute of Technology
Summer Session 1954

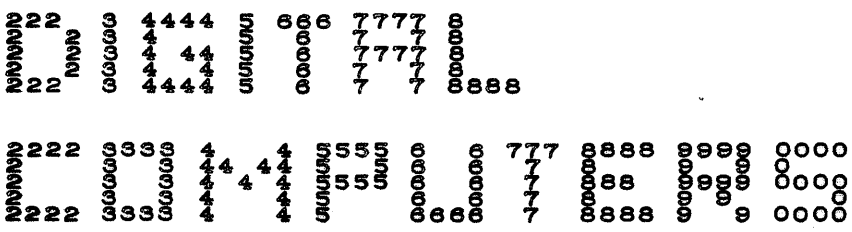
DIGITAL COMPUTERS

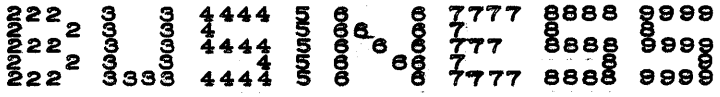
BUSINESS APPLICATIONS

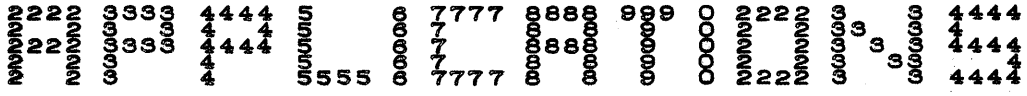
Massachusetts Institute of Technology

Summer Session 1954

Notes from a special summer program in







by
Charles W. Adams
Stanley Gill
and many others

with the cooperation of the
Digital Computer Laboratory of the
Department of Electrical Engineering
and the
Office of Naval Research

1. INTRODUCTION

These notes discuss a new kind of equipment for the rapid, reliable and inexpensive processing of large volumes of information -- systems built around general-purpose, automatically-sequenced, stored-program electronic digital computers. These devices, typified by Remington-Rand's UNIVAC and by IBM's types 701, 702, 704, and 650 have been the subject of much recent popular discussion, often being referred to as tape-processing machines, as electronic data-processors, or, unhappily, as "giant brains."

For convenience in what follows, the long and somewhat redundant modifiers -- general-purpose, automatically-sequenced, stored-program, electronic, and digital -- will be dropped, and the term "computer" alone will be used to refer to these systems. Before dropping the modifiers, one might reasonably ask what each of them means and what kinds of computing equipment are not being talked about here, as well as what kinds are.

Digital versus Analog

A digital computer is one which operates numerically on digits -- on quantities expressed in discrete, quantized, digital form. Digital contrasts with analog, which describes any computing device which operates on continuous variables. It is, in other words, possible to compute in two quite different ways -- by measuring to determine how much, or by counting to determine how many.

Man learned to count by enumerating things one by one in correspondence with his fingers and toes (his digits), just as he learned to measure lengths, say, by comparing with the length of his hand or his foot. Little wonder that he learned to count by fives and tens. Stones and then beads replaced the fingers, and the first real computing device, the abacus, was born. Adding machines, desk calculators, cash registers, parimutuel machines, score boards, and punched card machines are modern-day digital devices.

Slide rules, rulers, planimeters, scales, wind tunnels, towing tanks, and hour-glasses are analog devices. However, just as the term digital computer is frequently restricted to mean only a stored-program device, an analog computer is usually taken to mean not merely a simple computing aid or a model but rather an electronic, electrical, or mechanical device whose behavior is analogous to that of a given physical system. Analog computers are mainly useful in giving quick, inexpensive, relatively imprecise solutions to fairly complex engineering and control problems. The precision of an analog computer is limited to the precision of mechanical or electrical parts, usually one part in a hundred or a

thousand, or at most one part in ten thousand. The cost of increased precision in analog devices goes up exponentially, whereas the precision of digital devices can be increased by adding digits at less than linear increase in cost (and often by proper coding without even altering the equipment, although at a sacrifice of speed).

Getting back to the adjectives -- electronic, of course, implies that free electrons are used somewhere in the system, usually in a vacuum tube. Actually, the term electronic is usually reserved for those devices whose essential functions are performed entirely by electronics and not at all by mechanical processes. When mechanical actions control the flow of electrons and vice-versa, as in an electrical relay, the term electro-mechanical is used.

General- versus Special-Purpose; Applications

A general-purpose computer is one which can, if necessary, be used in any problem from any area of digital computer applications, unless its speed or its storage capacity is inadequate. This does not mean that any general-purpose computer is equally useful in any problem. Any computer is designed to turn in its best performance on some given type of problem. Often, however, the flexibility and complexity required for one given problem is such that a computer designed for the purpose turns out to be adaptable to almost any other problem -- though perhaps with much less efficiency dollar-for-dollar compared to a computer built with the other purpose in mind.

On the other hand, there are truly special-purpose computers -- such as desk calculators, accounting machines, IBM 407's, and magnetic drum inventory machines such as the American Airlines reservisor -- in which the lack of some facilities (notably decision-making) prevent the device from being adaptable to other kinds of problems.

Digital computers have application in five different areas:

1. SCIENTIFIC RESEARCH - testing a theory or method of solution by carrying out a solution and comparing the results with empirical data (when the method is successful, it often is used in an engineering analysis or synthesis approach to give new data).

2. ENGINEERING ANALYSIS AND SYNTHESIS - applying known techniques of analysis to the parameters of some system design (usually one chosen in an attempt to synthesize by finding parameters which will give the desired result); for example, computation of aircraft design, assembly-line balance, insurance rate tables, operations research, etc.

3. REDUCTION OF EXPERIMENTAL DATA - Processing engineering tests, e.g., rocket firings, into more usable form.

4. PROCESS CONTROL - determining, from all available measurements, the corrections needed at all available control points to adjust a physical system to give the desired output. For example, control of aircraft in flight, of steel mills, of refineries, of assembly plants, etc.

5. BUSINESS DATA-HANDLING - processing the day-to-day work of inventory records, accounting, payroll, etc., and from this obtaining data to be reduced for control and management purposes.

Each of these kinds of applications puts special demands on the computing system. For example, ease of programming is especially desirable to scientific research; dependability is especially essential in business data-handling and usually in process control; special input facilities are needed for all of the last three named; special output facilities are needed for the last two; and a large storage system or automatic file is usually needed for business data-handling.

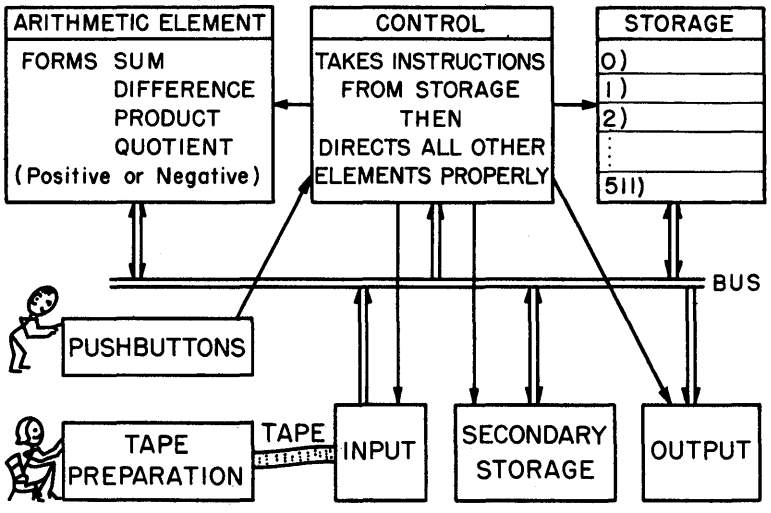
Automatic Sequencing

According to our definitions, an adding machine or a desk calculator can truly be called a general-purpose digital computer. Such machines today perform a ten-digit multiplication in less than ten seconds. Modern electronic techniques can speed this up a millionfold -- but to what avail? A competent person operating a modern desk calculator performs about 500 operations a day. By building a calculator a million times as fast, one can reduce the maximum of 5000 seconds of machine operating time per day to a twentieth of a second, but speed up the overall process by at most 10% or 20%. The bottleneck is, of course, the human operator.

The automatically-sequenced digital computer is simply a mechanization not only of the arithmetic operations but of the operator who determines the sequence in which the operations are performed. All of the important logical principles of the automatically-sequenced digital computer were outlined by Charles Babbage over a hundred and twenty years ago, but mechanical and electrical techniques could not satisfy his needs at that time and his Analytical Engine was never built.

The arithmetic element of an automatically-sequenced digital computer, corresponding to the desk calculator of a manual system, can advantageously be made to work very fast, performing arithmetical operations in a few millionths of a second, for the rest of the system can now keep up with it. The control element, the counterpart of the human operator, can readily be made far faster, more reliable, and somewhat less demanding of wages and fringe benefits than the man. Unfortunately, however, there is need for automatic memory or storage of various degrees of accessibility, corresponding to the memory of the operator, the notebook, and the reference library. There is also need for input and output - the means of communication with the outside world. It is the memory and the input-output that causes the greatest engineering

SEMI-AUTOMATIC DIGITAL COMPUTATION



AUTOMATIC DIGITAL COMPUTATION

difficulties in the physical realization, and places the greatest limitations on the speed and reliability of existing computers.

Stored-Programs versus Card-, Tape-, or Plugboard-Programmed

The flexibility of the sequencing can be improved if the computer can

1. move back and forth within the sequence without manual operation or other excessive delay, and

2. make modifications on its own instructions.

These two facilities are gained if the program of instructions as well as the data are stored in the internal, random-access memory. This is not what is done if plugboards or sequences of cards or punched tape are used to store the instructions. The second facility listed is achieved automatically if the instructions go into the same storage element as the numbers, although there are stored-program computers in which this is not done. Since all of the following discussions are based on stored-program computers, it is to be hoped that the virtues of such a system will become more apparent as the discussion progresses.

Programming and Coding

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

1. planning the program, or sequence of elementary steps, by which the problem may be solved

2. coding the sequence of steps into a coded program - a sequence of computer instructions. There are a number of other steps concerned with coding, as shown in the figure entitled Digital Computation Center, but except for debugging (removing mistakes) these are largely clerical.

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.

Since it is virtually impossible to discuss programming without referring to a specific computer, two hypothetical computers, called TAC and SAC (for Three-Address Computer and Single-Address Computer, respectively), will be assumed throughout the ensuing discussions and examples. These computers do not in fact exist, but their characteristics are to be simulated on the Whirlwind I computer to permit programs written for TAC or SAC to be performed and the results or lack of them to be indicated realistically.

The simulation of these idealized, pedagogical computers by Whirlwind I has actually been accomplished by means of special programs written for the Whirlwind I computer by members of the summer school staff. Such programming represents a considerable effort (several man-months) by experienced programmers and is not unlike the job of constructing a digital computer from tubes and wires, except that a pencil is used in place of a soldering iron. The techniques for accomplishing the simulation of one computer by another, and the motivations for doing so, will be described briefly in Chapter . For the present purposes, TAC and then SAC will be described as if they existed in the hardware and no further mention will be made of the role played by the Whirlwind I computer.

Both TAC and SAC have, of course, the basic computer elements: arithmetic element, control, storage, secondary storage, input and output. Between them they are intended to typify most of the characteristics of most of the other currently-available digital computers, combining many of the best features, omitting some of the special features and peculiarities of each. Naturally, both important concepts and innumerable details make up a complete description of the computer. Rather than attempt to describe the TAC and SAC computers completely at the outset, we have attached a complete description as an appendix to these notes and will build up first TAC and then SAC gradually, embellishing them with more details and more new concepts as we progress.

2. THE RUDIMENTS OF TAC CODING

The best way to learn to code for a given computer is to work out a few examples. First, of course, one must become acquainted with the general characteristics of the machine at hand. A complete description of TAC, such as is appended as Chapter 24, is hardly necessary at this point. It seems more reasonable to build the description a little at a time; but even so, the first bite, as presented in this chapter, is a big one.

General Characteristics of TAC

The storage element of TAC consists of 110 different registers. A storage register is a location, like a pigeon hole, in which a single computer "word" may be stored by the computer control element and recovered by it when the word is needed.

A word in TAC is a series of 9 digits, letters or other characters representing a number, an instruction, or some other data. Since with numbers the first character always is a sign, any one storage register can contain any integer (whole number) between -99999999 and +99999999 --anything up to a hundred million.*

Each register is identified by an address, just as the houses on a street are identified by addresses. The addresses of TAC's 110 different registers are 00, 01, 02,..., 99, x0, x1,...,x9. (These addresses are all listed on the "TAC Program Form", along with space for filling in the contents of each of the corresponding registers. One character goes into the first space and two into each of the others merely for convenience in reading and writing. There is also a blank space for comments or notes.) The x registers differ from the others by being, in effect, faster than the rest, as will be seen in more detail later. Register x0 is very special; it contains a very useful number, 000000000, and its contents can not be altered in any way by any instruction. Registers x1 and x2 will also be seen later to have an important property---TAC can automatically deal with them in tandem as a single "register" (called xx) which stores a 16-digit number made up of the two 8-digit numbers which they individually store.

---

*Alternatively, since a character of a word can be any of the symbols on the MIT Flexowriter keyboard (similar to a standard typewriter), it is possible to store any 9-character combination such as Joe E Doe or 16-8-1954 (spaces, dashes, etc., count as characters). However, certain of the instructions about to be described will deal only with strictly numerical quantities. Numbers larger than a hundred million or smaller than one may be handled by special techniques, but only signed 8-digit integers will be discussed here at the beginning.

An instruction is one kind of word. It specifies both an operation to be performed, such as add or subtract, and the addresses which designate where the words to be operated upon are to be found. For example, the word A27131609 (this and the many instructions which follow are easier to read if punctuated by spaces which would not actually appear in the word in TAC--thus, A 27 13 16 09) is an instruction which specifies that: the integer contained in register 27 is to be added (because the letter A represents addition) to the integer contained in register 13 (without changing what is contained in registers 27 or 13), the sum is to be stored in register 16 (erasing whatever was previously in register 16), and the next instruction to be obeyed is to be taken from register 09 (without changing what is in register 09). The fact that TAC is called a three-address computer, yet deals with instructions that clearly have four addresses in them, is merely a question of terminology. The fourth address, specifying the location of the next instruction, is not counted in the name because it does not serve an important logical function, for, as the Single-Address Computer SAC will demonstrate, the instruction could nearly as well be taken from consecutively-numbered locations.

Symbols Used

The results of the various operations which TAC can perform are easier to describe if a few symbols are used for commonly-needed phrases. Two in particular are useful enough to have become somewhat standard in the computer field generally, although many variations still exist. The symbols are:

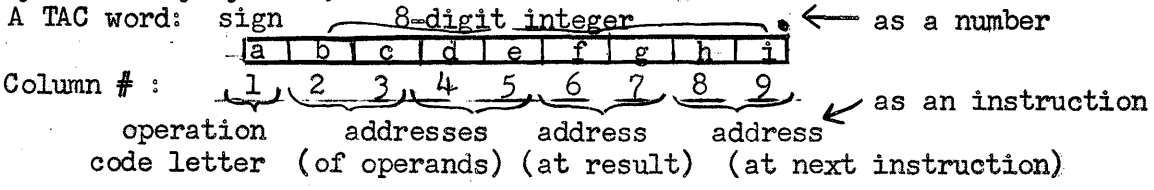
C( ) represents the word contained in the register whose address is enclosed by the parentheses, usually being read as "the contents of --" or sometimes merely as "C of --"

$\longrightarrow$ represents the phrase "becomes the new contents of" or simply "goes into".

For example:

C(27) + C(13) $\longrightarrow$ 16 should be read as follows: "the contents of register 27 plus the contents of register 13 becomes the new contents of register 16" or, for brevity, "C of 27 plus C of 13 goes into 16".

Another group of convenient symbols refers specifically to TAC alone. They make it possible to identify and distinguish between the 9 different characters (whether they be digits, signs, letters, symbols, etc.) which make up a TAC word. This is done in two different ways. First, the character positions or columns of a word are numbered 1, 2, 3, 4, 5, 6, 7, 8, 9 from left to right. Second, any word is sometimes represented symbolically by the 9 letters a b c d e f g h i.



Thus the letter d is used to represent the 4th character of the word, whatever that character happens to be. Similarly, the pairs of letters bc, de, fg, and hi are used to represent the four different addresses specified in any instruction word.

For example, in the add instruction A 27 13 16 09 described earlier, a = A (since the first character of the instruction was a capital A) bc = 27, dc = 13, fg = 16, and hi = 09. The result of this particular A instruction can be described symbolically, as above, by writing C(27) + C(13) $\longrightarrow$ 16. Similarly, the result of any A instruction can be written symbolically as C(bc) + C(dc) $\longrightarrow$ fg, which simply means that "the contents of the register designated by the first address (second and third characters) of the given instruction plus the contents of the register designated by the second address (fourth and fifth characters) of the given instruction becomes the new contents of the register designated by the third address (sixth and seventh characters) of the given instruction" and, implicitly, the next instruction is to be taken from hi, the register designated by the fourth address (eighth and ninth characters) of the given instruction.

Arithmetic Operations A, S, M and D

The four most obvious operations for TAC to perform are addition, subtraction, multiplication, and division. These are comparatively simple to describe and to use, especially if no attention is given to the questions of what happens if the results are too large to fit in a register or if non-numerical quantities are involved. Suffice it here to say that if the result of an arithmetic operation exceeds eight digits, TAC will print out certain symptomatic information, called a post mortem, and then stop. Non-numerical quantities can in certain cases be added or subtracted but not multiplied or divided, and results up to 16 digits may be handled by using the special xl-x2 tandem register xx. Detailed discussion of these complications will be deferred.

The four basic arithmetic operations in TAC, representing the four addresses by bc, de, fg, and hi as described above, are:

Name	Code	Function
Add	A	C(bc) + C(de) $\longrightarrow$ fg
Subtract	S	C(bc) - C(de) $\longrightarrow$ fg
Multiply	M	C(bc) x C(de) $\longrightarrow$ fg
Divide	D	C(bc) $\div$ C(de) $\longrightarrow$ fg

(next instruction from hi in all cases)

Thus, to repeat a previous example, the instruction A27131609 adds C(27) to C(13), places the result in register 16, and causes the next instruction to be taken from register 09. TAC treats all numbers as integers. Adding, subtracting, or multiplying two integers gives an integer as the exact result, and this is the result stored in fg by the A, S, or M operations. Division, however, is not an exact process. In this case TAC rounds off in a conventional way, equivalent to adding $\frac{1}{2}$ to the unsigned full quotient and then throwing away the entire fractional part. Thus TAC forms, as the rounded quotient, an integer which is at least as nearly equal to the exact quotient as any other integer would be.

One other instruction will be necessary in even the simplest cases, namely an instruction which stops the computer. This is called Halt rather than stop since the code symbol H was available whereas the symbol S is used for Subtract. Halting does not involve any operation on the contents of any registers, so the first three address sections are of no significance and may be filled in with zeros or any other characters. The fourth address of Halt specifies, as usual, where the next instruction is to be found, but this is of no significance unless it is anticipated that one might want to press a button and have the program carry on or repeat, in which case the halt address should be properly filled in. Since one occasionally wants to repeat a calculation just to make sure the machine was not behaving badly the first time through, it is good practise always to set the fourth address of a Halt instruction so that the program can be repeated even if no repeat is anticipated.

Name	Code	Function
Halt	H	stop the computation
(take next instruction from hi if restarted manually)		

Use of the Arithmetic Instructions

For example, if register 00 contains the integer +00000007 and register 01 the integer +00000095, then a program for placing 7 + 95 in register 02 would be

```
03| A00010204
04| H00000003
```

which, assuming that the computer is somehow gotten to start with the instruction in 03, adds the 7 in 00 to the 95 in 01 and places the result, +00000102, in 02. The H operation then stops the computer, ready to repeat if necessary. To place 95÷7 in register 2, the program could be

```
03| D01000204
04| H00000003
```

which divides C(01) = +00000095 by C(00) = +00000007, forming the result +00000014 which is then placed in 02.

As a first real example, which must of course still not be very hard, we give a nod to THE COMPLEAT STRATEGYST, which gives social standing to this type of problem, and take up the case of the vending-machine operators. Consider the plight of John, Mal, and Arnie, whose vending machines bring them each day a number of coins which we will represent as x nickels, y dimes, and z quarters. Somehow they must total the take and divide the profits, with Arnie giving or taking the odd cent if the total does not divide evenly. Their business being large (but less than a million dollars a day), they decide to use TAC to take the output of their electronic coin counters and work out the split. Our task is to write a program which will make TAC do the job.

First, let us express the problem as concisely as possible. The gross take, in cents can obviously be found by taking 5 times the number of nickels (which we decided to call x) plus 10 times the number of dimes (y) plus 25 times the number of quarters (z). Clearly, also, a share is a third of the gross, except that in rounding off to the nearest cent we may make the shares a third of a cent too large or too small. In this case, Arnie's share is to be a cent larger or a cent smaller than John's and Mal's. Thus we can find John's share and Mal's share by simply dividing the gross by 3, and then find Arnie's share by subtracting the other two from the total. Our problem can be summarized by three equations

Gross = 5x + 10y + 25z
John's share or Mal's share = Gross/3, rounded off
Arnie's share = Gross - John's share - Mal's share

Next, let us express the calculation in terms of a sequence of arithmetic operations

1. multiply: 5 times x = 5x
2. multiply: 10 times y = 10y
3. multiply: 25 times z = 25z
4. add: 5x plus 10y = 5x + 10y
5. add: 25z plus (5x+10y) = Gross
6. divide: Gross divided by 3 = John's or Mal's share
7. add: J's share plus J's share = 2 J's share
8. subtract: Gross minus 2 J's share = Arnie's share

Now we have a program -- a sequence of elementary operations -- for solving the problem. The final step is to code this program into basic TAC language. To do this, we must first assign all the necessary numerical information to specific registers in TAC's storage element. Let us list the necessary numbers

1. 5	constants needed	8. 5x	unknowns to be computed	
2. 10		9. 10y		
3. 25		10. 25z		
4. 3		11. 5x+10y		
5. x	the number of coins, which is data we are to be given	12. Gross		
6. y		13. John's or Mal's share		
7. z		14. twice John's share		
		15. Arnie's share		

Each of these quantities must be put into some storage register before TAC can carry out the calculation. Similarly, the coded program or list of instructions that we intend to write must also be stored, one instruction to a register. It makes no difference which register is used for which instruction, constant, or unknown (actually we will find later that, in so-called "serial" computers like TAC, the computation may go faster if the numbers and instructions are judiciously placed). Since it does not matter, we may as well assign the various constants, input data, and unknowns to registers 01 through 15 in the order in which they are listed above. Suppose we are given specific values for x, y, and z: 13,976 nickels; 9,433 dimes; 2,747 quarters.

01	+00000005	constants		08	registers reserved for quantities to be computed	5x
02	+00000010			09		10y
03	+00000025			10		25z
04	+00000003			11		5x+10y
05	+00013976	x	given data	12		Gross
06	+00009433	y		13		J or M share
07	+00002747	z		14		2 J's share
				15		A's share

To get the constants and the given data into registers 01 through 07 where we want them, we will simply type the addresses and the quantities, exactly in the form shown, on the Flexowriter, preparing a punched tape which can be read into the computer by putting it into TAC's tape reader and pressing the "READ IN" button. To reserve registers 08 through 15 we need do nothing at all except not to use these registers for anything else in our own program. The instructions, once we get them written, can go on the same tape with the data, typed in the same form.

In TAC, any of the instructions, including the first one, can go into any of the registers. At the end of the punched program tape we will type the location of the first instruction, followed by the word "start" and TAC will then stop reading in and start computing. Consequently, we can place our instructions anywhere in storage, except of course in the registers we have already assigned to other purposes. Register 16 seems as good a place to start as any. Since each of the operations called for in our previous list is an operation in TAC's vocabulary, and since all of the numbers involved are integers that will fit into TAC's registers, the coded instructions correspond directly to the operations of our previous list.

The list of instructions which we might use to describe our problem is as follows:

instructions		comments
16	M 01 05 08 17	5x→8
17	M 02 06 09 18	10y→9
18	M 03 07 10 19	25z→10
19	A 08 09 11 20	5x+10y→11
20	A 10 11 12 21	25z + (5x+10y)→12
21	D 12 04 13 22	Gross/3→13
22	A 13 13 14 23	2 J's share→14
23	S 13 14 15 24	A's share→15
24	H 00 00 00 16	stop

	numerical quantities (relisted for reference)	
1	+00000005	
2	+00000010	
3	+00000025	
4	+00000003	
5	+00013976	x
6	+00009433	y
7	+00002747	z
8	5x	
9	10y	
10	25z	
11	5x+10y	
12	5x+10y+25z = Gross	
13	Gross ÷ 3 = J or M share	
14	J + J = 2J	
15	Gross - 2J = A's share	

The instruction in register 16 says: multiply what is in register 01 (namely 5) by what is in register 05 (namely 13976), place the product (namely 69880) in register 8, and then take the next instruction from register 17. The instruction in 17 then places 10 × 9433 = +00094330 in 9; the instruction in 18 places 25 × 2747 = +00068675 in 10; C(19) places +00164210 in 11; C(20) places +00232885 in 12; C(21) places +00077628 (John's share or Mal's share, in which the extra third of a cent was rounded off) in 13; C(22) places +00155256 in 14; C(23) places +00077629 (Arnie's share, which in this case is one cent larger than John's or Mal's) in 15; and C(24) stops the computer -- ready to repeat if necessary. The complete program as typed in preparing the punched tape for TAC might look as follows, where the top line contains some conventionalized identification which will be described later on.

```
f2t 198-400-1
01 +00000005
02 +00000010
03 +00000025
04 +00000003
05 +00013976
06 +00009433
07 +00002747
16 M01050817
17 M02060918
18 M03071019
19 A08091120
20 A10111221
21 D12041322
22 A13131423
23 S13141524
24 H00000016
16 start
```

Program for the Vending Machine Operators

Output

A program to be useful must print the results of the calculation so that they can be read, not simply keep the whole thing to itself as in the preceding example. It is consequently necessary to have an instruction which prints. It would on the whole be satisfactory for this instruction simply to print the contents of a specified register. This would leave two addresses unused (unless one printed three words at once, which might not always be convenient), which does not make computer designers feel very bright.

In addition, it is very necessary to print the information intelligibly. In the preceding example, John's and Mal's shares and Arnie's share could be printed merely as +00077628 and +00077629, but it would be much better to print the shares in dollars, with some identification as well, for example:

John	$776.28
Mal	$776.28
Arnie	$776.29
total	$2328.85

The print instruction therefore permits printing parts of words, and permits putting one character, such as a space, a dollar sign, a comma, a period, a carriage return, etc., both before and after each printed group. (The complete list of Flexowriter characters includes digits, superscripts, capital letters, small letters, symbols, and machine functions, as listed at the top of page 24-1. The machine functions are represented in writing by underlined capitals: R = carriage return, S = space, T = tab, B = back space, C = color change, H = halt--printer only, , I = ignore). In the print instruction, only de is an address in the normal sense. The digits b and c are used to designate what part of the word is to be printed, b indicating which column to start with and c indicating which to end with. Characters f and g may be any Flexowriter characters at all. TAC simply prints f itself, whatever it is, before printing the designated part of C(de) and then prints g itself at the end. If nothing is wanted either before or after, f and/or g may be designated as I, representing "ignore."

Name	Code	Function
Print	P	Print the characters in the columns numbered b through c in C(de), preceded by the character f and followed by the character g

For example, in the vending-machine operator problem again, we could print John's share and Arnie's share in the form

+00077628
+00077629

by replacing the halt instruction in register 24 by the following:

24	P 19 13 I R 25	print characters 1 to 9 of C(13) preceded by nothing and followed by a carriage return (with line feed)
25	P 19 15 I I 26	print characters 1 to 9 of C(15) and nothing else
26	H 00 00 00 16	stop, ready to repeat if necessary

Alternatively, we could print their shares in the form

$776.28
$776.29

by writing

24	P 57 13 $. 25	print $ 776.
25	P 89 13 I R 26	print 28 and return carriage to next line
26	P 57 15 $. 27	print $ 776.
27	P 89 15 I I 28	print 29
28	H 00 00 00 16	stop

Suppose we really want to be really elegant -- we might arrange to print something like the earlier example.

John, Mal: $776.28
Arnie: $776.29
Total: $2,328.85

This requires a long string of print instructions and the insertion of some special words to contain the names "John, Mal:" and "Arnie:", the backspacing and underlining and the word "total:". Thus

24	P 19 25 J S 26	Prints John, Mal: followed by a space
25	ohn, Mal:	
26	P 57 13 $. 27	Prints $776.
27	P 89 13 I R 28	Prints 28 followed by a carriage return
28	P 19 29 A S 30	Prints Arnie: followed by 5 spaces
29	rnie: S S S S	

(continued on next page)

30	P 57 15 ¢ I 31	Prints ¢776
31	P 18 32 I I 33	Prints 4 backspaces, 4 underlines (i.e. underlines the four preceding digits)
32	B B B B _ _ _ _ I	
33	P 89 15 . I 34	Prints .29
34	P 27 32 I R 35	Underlines the 3 preceding digits and returns carriage
35	P 29 36 I S 37	Prints 2 spaces, then Total: , then another space
36	I S S Total:	
37	P 44 12 ¢ , 38	Prints ¢ 2, (total is in reg. 12)
38	P 57 12 I I 39	Prints 328
39	P 89 12 . H 40	Prints .85 and a stop character
40	H 00 00 00 16	Stops the computer.

The example given does a neat job of printing, but it is predicated on a beforehand knowledge of how many digits there are in each of the results. On a bad day, the boys might find their share printed in the form

John, Mal: ¢087.13
Arnie: ¢087.12
Total: ¢0,261.38

which would be esthetically displeasing, while on the good days, they might lose a thousand dollars apiece (although the discrepancy would show up in the total). The catastrophe of losing digits can be precluded by arranging always to print more digits, but this means that the results would almost always lack esthetic appeal. The proper solution, usually, is to write an elaborate program for determining how many significant digits there are and then printing only those. Such a zero-suppression program involves more coding features and tricks than have been covered thus far.

Input

We have seen that inserting a program of instructions and initial values into TAC is done by simply preparing a punched tape, putting it in the reader, and pressing the button. There are times, however, when the same program is to be used over and over again on different data where it is not efficient to have the computer stop after each calculation and wait for someone to press the button to read more data in. Also there are times when it is not convenient to prepare data directly

in the form of TAC words. Both of these situations are provided for by means of the read instruction, R, described below.

Name	Code	Function
Read	R	Read enough characters from punched tape to fill the positions numbered b through c in register de without changing the other digits of C(de)

Unlike program input, the Read operation reads explicitly each character which appears on tape except the code delete (all holes punched) which is ignored, and the upper and lower case symbols which are not counted (although they are used automatically to differentiate capital letters from small ones, etc.) Consequently underlined characters and ß and ¢ are each read as three separate characters.

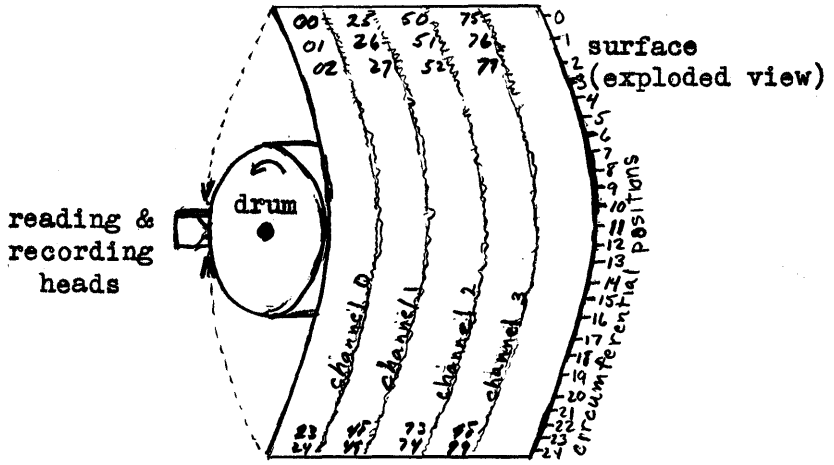
For example, in the vending-machine problem it would be more practical to leave the number of nickels, dimes, and quarters unspecified in the program but to arrange for the program to start by reading the necessary data from a separate tape. We might for instance arrange to have the coin counts typed out in the form "013976 nickels, 009433 dimes, 002747 quarters." The program for reading these quantities into registers 5, 6, and 7 could then be

25	R 49 05 00 26	reads 013976 into the right hand end of register 04
26	R 19 27 00 28	reads Snickles, into 27 to discard it
27	+00 00 00 00	acts as a waste basket to receive unwanted information
28	R 49 06 00 29	reads 009433 into the right hand end of register 05
29	R 17 27 00 30	discards Sdimes,
30	R 49 07 00 31	reads 002747 into 06
31	R 18 27 00 16	discards Squarters, goes to start of original program
25	start	new starting address

The above program will (as long as the contents of registers 04, 05, and 06 start out with +00 as they do in the original program) put the same data into 04, 05, and 06 as we did earlier by writing the values into the program. The revised program, however, can be used without change from day to day for different numbers of coins, and the number of coins can be tabulated in a convenient, readable form rather than having to be incorporated into the program in an artificial way.

3. Drum Storage, x-Registers, Scaling, Precision, and Conditional Sequences

TAC's primary storage element is imagined to be a magnetic drum, a rotating cyclinder with a magnetizable surface, on which the 9 characters comprising each of the 100 words stored in locations 00 through 99 are recorded. Each character occupies a sequence of seven positions in each of which there either is a pulse or no pulse, roughly corresponding to the system of holes or no holes used on punched tape or punched cards.



TAC's Hypothetical Drum

Access to a particular storage location requires electronically selecting one of the four channels on the drum, then waiting until the proper one of the 25 words on the drum begins to pass the reading heads. The channel selection and the circumferential position are found in effect by an exact division of the desired address by 25, the quotient being the channel, and the remainder being the desired circumferential position around the drum. The time between calling for a word and receiving it from storage is called the "access time". Access time is made up of the waiting time, or "latency", and the time taken for all of the string of 9x7=63 pulses to appear and be read, which is called the "word time".

Since TAC's drum is imagined as rotating 40 times a second, and since there are 25 words on each track, words go by the reading-recording heads at a rate of 1000 per second. A word time in TAC is therefore one thousandth of a second, usually called one milli-second. The "latency" varies from 0 to 24 word times, since the best that can happen is for the desired register to turn up just as it is wanted, and the worst is for it just to have passed. In executing most instructions, TAC must make four references to storage, one to read each of the operands, one to store the result, and one to read the next instruction. It should be noted here that "read", just as in its common usage, means to sense intelligently without destroying the information recorded.

The time taken to execute an instruction depends largely on the latency -- the time spent waiting for the right information to show up in the drum. TAC looks for the two operands specified by the bc and de addresses simultaneously if both are wanted. Thus TAC takes first whatever comes first, then waits for the other, or takes both together if they appear together. This has the effect of reducing the latency appreciably. TAC does not try to obtain the next instruction while it is still trying to store the result of the previous one, however, because this would be a possible source of mistakes by programmers if they attempted to alter the instruction that is to be performed next. The fact that latency can be reduced by judicious arrangement of words in storage is a matter of considerable interest in some computer designs and a matter of some consternation to the programmers. Work is progressing on producing computer programs which will in turn undertake to produce a "minimal latency routine". In the meantime we will merely note the existence of this complication and not concern ourselves here with the details.

The x-Registers

It is possible to reduce the latency a great deal if there is a small amount of storage which has a random access with no latency. Such parallel storage systems are commonplace for the full primary storage in present day computers, but they are more expensive than equivalent serial storage capacity. By a small amount is meant anywhere from one to sixteen registers, an amount which does not add as much percent-wise to the cost of the computer as it does to the effective speed. The high-speed or zero-latency registers in TAC are imagined to be either magnetic core stepping registers or drum revolvers, devices which will be discussed briefly in chapter 13. They are labelled x0, x1, x2, x3, x4, x5, x6, x7, x8, and x9, and are ordinarily used to hold frequently-needed data and intermediate results, although they may be used just as any of the drum registers are used.

In any computer, the number zero is needed surprisingly often. In a three-or four-address computer, zero is especially useful when a word is simply to be copied from one location into another. Register x0 has consequently been built to contain zero, in the form 000000000, and can not be made to hold anything else. Even if a non-zero word is sent to x0 by an instruction, the zero will be unchanged, the non-zero word lost, and no harm will be done unless the lost word was wanted. This feature makes x0 useful as a waste basket, in the way (for example) in which register 27 was used on page 2-11 to permit reading and discarding unwanted information. (Since x0 in effect contains nothing, there would, if TAC were to be built, be no need to use any hardware at all for register x0.

Tandem Register xx

Registers x1 and x2 are, like register x0, rather special as well, but in a different way. TAC is equipped to deal with x1 and x2 in

tandem, as if they were a single register twice as long as usual. Taken together, the pair is called double-x, written xx. The purpose of this feature is to facilitate arithmetic operations in which 8 digits are not sufficient. In multiplication in particular, the product of two 8-digit numbers is a 16-digit number, and even if only the 8 most significant are to be retained later, it is useful to be able to form the full product and then divide it by 100,000,000 (for example) to reduce it back to eight digits to be stored. There are in fact a great many times when a double-length register is useful if not necessary, as we will see as we progress. Double-x serves this purpose well.

The four arithmetic operations described thus far (A, S, M, D) and the two which are to be described in this chapter (N, C) all may involve double-length operations in xx, except only that TAC will not divide by the contents of xx. The address xx may, for instance, be used as the fg address in an add or subtract instruction to obtain a double-length sum. No post-mortem (see page 2-3) will occur when sums, differences, products, or quotients are greater than 99,999,999 if the result is to go into register xx. If the result to go into xx exceeds 9,999,999,999,999,999, a post-mortem will occur, but this of course cannot happen unless C(xx) was one or both of the terms going into the operation being performed.

The contents of xx is defined to be C(x1)x 100,000,000 + C(x2). Thus, if C(x1) = +12345678 and C(x2) = +98765432, then C(xx) = +1234567898765432. C(xx) is not defined at all unless C(x1) and C(x2) are both numbers and both have the same sign. Both C(x1) and C(x2) retain signs separately, but TAC will not operate on C(xx) at all unless the two signs are the same -- a post-mortem occurring if they differ. The xx register cannot be used in connection with any of the logical operations to be described later nor can its contents be printed from or read into. This is no real limitation since x1 and x2 may always be manipulated separately.

Precision and Scaling

Adjusting numerical, alphabetical, and other logical information to the Procrustean bed of computer word lengths and magnetic tape block lengths is often a severe problem. The numerical computations of science, engineering, process control, statistics, etc., are generally concerned with numbers which are approximations to physical quantities, and in these cases one usually wants to carry just as many significant digits as can economically be carried, provided that this is greater than some individually prescribed amount for any given problem.

The number of digits needed is usually between 4 and 10 and often between 5 and 8. To allow for these cases, most computers are built to carry from 8 to 13 decimal places. There remains a further problem, that of keeping the quantities "centered" in the registers so that no digits "overflow" the left end (a catastrophe) and not too many drop

off the right end (a loss in precision). The problem of carrying enough digits is one of precision, whereas the closely related problem of making full use of the available digits by keeping numbers centered is one of scaling.

The solution of the scaling problem is straightforward, provided that the programmer knows in advance how large each piece of data and each intermediate and final result can possibly be, and provided that allowing room for this maximum does not make the average precision too small by wasting too much space in the average case. Knowing the maximum, one simply chooses the units in which the quantity is to be expressed in such a way that the maximum value will fit in the register.

For example, in writing a program for TAC to deal with the distances involved in a land survey of Middlesex County, one would probably express distances to a millionth of a mile, or perhaps to a hundredth of a foot or a tenth of an inch, since in each case the maximum tolerable distance would be from one to two hundred miles - which is somewhat more than enough than is needed on the left end, but there is also more than is strictly needed on the right, so it matters little. On the other hand, dealing with distances in nuclear physics, one might choose a thousandth or a millionth or a billionth of an angstrom (a unit which is itself only 4 billionths of an inch); whereas in planning global strategy one would perhaps use thousandths of miles (to allow a maximum of a hundred thousand miles), and in astronomy one might use a million or a billion miles as the unit of measure. Nor is there any reason other than convenience for choosing standard units or decimal fractions thereof. In global strategy, where one might not be interested in distances over 25000 miles, one could use one four-thousandth of a mile as a unit. Of course, there would ordinarily be little point in this as it would not buy enough extra precision to be worth the minor inconvenience.

The choice of units amounts to shifting the decimal point within the number being operated on. In effect, the decimal point of the quantity, if expressed in the most natural or convenient units, will lie at some number of places to the right or the left of the largest digit, while in TAC the point always lies 8 to the right of the largest digit. Scaling is merely keeping track of the difference between these two locations. For example, in global strategy, the largest distance may be 17384. miles, with the point 5 places right of the largest digit. Since we want the largest digit to fall in the left end of a TAC register, and since TAC will assume 8 places to the right, there is a difference of $5-8 = -3$ places, and the scale factor should be $10^{-3} = 1/10^{+3} = 1/1000 = 0.001$, i.e. the unit used should be a thousandth of a mile so that the maximum value would occupy 8 digits, just as was decided earlier.

Floating-point Numbers

In any computation in which the numbers are merely approximations to some physical quantity, scaling is concerned only with leaving

as few zeros at the left end of the words in storage as possible, since these are merely wasted precision. Keeping all the quantities centered close to the left end without overflow is usually a ticklish problem, often a very difficult one, especially if the maximum values of the quantities involved are not known in advance. Since keeping track of the decimal point is itself a computational job, there seems little reason why the computer should not be made to do its own scaling. This is, in fact, done quite often by writing, once and for all, programs which will keep track of decimal points during the entire computation. Such programs are called floating-point routines, because the decimal point in effect floats at its proper place within the register. Some computers are being designed with the ability to do floating-point arithmetic designed right into the hardware, e.g., the IBM 704. Floating-point numbers are made up of two parts, in a kind of logarithmic fashion. For example, the quantity -12345.67 may be represented by -.1234567:+5, since it equals $-.1234567 \times 10^{+5}$; the quantity +.0004567 may, in the same system, be represented by +.4567000:-3 since it equals $+.4567000 \times 10^{-3}$. Since it turns out that floating-point techniques are of relatively little value in business problems, there is little value in studying the techniques any further here.

Integers vs. Fractions in Business

In most business applications, it appears numbers have a different meaning than they do in most engineering and other such applications. Most numbers in business are, by fact or by definition, exact -- not merely approximations of physical quantities. A price, a catalog number, a quantity, are usually exact integers. Rounding a number is not necessary as often as it is in engineering work, and when a quantity is rounded, the number of digits to be kept and even the rule for rounding is usually specified exactly by law or by custom. If some item lists for $20.78, for example, its price is 2078 cents, exactly; if one buys 14 such items, the list price is 14 x 2078 = 29092 cents exactly; but if the discount is 10%, an approximation is required, for the discount cannot be given as 2909.2 cents. The fact that most business quantities are expressed as whole numbers, but that percentages and many other quantities are really fractions less than one raises the question of whether a computer should treat numbers as integers, with the point at the left, or as fractions, with the point at the right, or somewhere in the middle.

Addition and subtraction turn out to be the same no matter where the point is located -- it makes no difference even if the computer thinks the points are on the right and the programmer thinks they are on the left, as long as the point are actually in the same place in both of the numbers being added or subtracted. Multiplication, on the other hand, is a horse of a different color. Consider the two cases:

$$\begin{array}{r} 27. \\ \times\ 12. \\ \hline 0324. \end{array} \qquad\qquad \begin{array}{r} .27 \\ \times\ .12 \\ \hline .0324 \end{array}$$

The product is essentially twice as long as its factors. In the example, the left-hand word (03) of the product is called the major half of the product, the right-hand word (24) is the minor half. A computer which works with integers ordinarily keeps the minor half of the product and throws the major half away, whereas a computer working with fractions keeps the major half and discards the minor half, or perhaps rounds it off by adding one to the major half if the leftmost digit of the minor half exceeds 5. It is evident that neither procedure fits the bill in all cases in either business or any other applications. Floating-point of course avoids this question, but floating-point is not especially satisfactory for exact operations on integers and is completely unsatisfactory for the many non-arithmetical, logical operations required in business.

The question of decimal-point location seems to be answerable as follows:

1. in business applications, where working with integers and logical values is more common than not, a computer operating on numbers with the point fixed at the right is usually more satisfactory than any other single mode of operation.

2. in engineering and other applications dealing with approximations of physical quantities, not-exact-but-best-possible results are usually wanted and a point-fixed-at-left computer is slightly preferable to one with the point fixed at the right, while one with built-in floating-point and fixed-point together is naturally preferable to either fixed-point scheme.

3. in BOTH kinds of applications, IT MUST BE POSSIBLE, IF NECESSARY, TO DEAL WITH FIXED-AT-RIGHT, FIXED-AT-LEFT, FIXED-ANYWHERE-ELSE, AND FLOATING POINTS.

Naturally it is possible to have a machine with the point fixed at the middle or somewhere else between the ends, and there are such computers (e.g., Monrobot); but the advantage is small unless there are enough digits (say ten on either side) to permit the computer to act simultaneously like a fixed-at-left and a fixed-at-right machine (and this then doubles the cost of the computer storage element since only half of each register will ordinarily be useful in any one problem).

In TAC, it is primarily the xx register that facilitates dealing with other than fixed-at-right integers. Register xx also facilitates dealing with numbers which are inherently longer than 8 digits, but a discussion of that situation will be postponed until chapter 5. It should be noted, in connection with discussing special facilities, that while such features are often highly desirable they are never absolutely essential. (In fact, it can be shown to be possible, although not practical, to perform any arithmetical or logical operation with a "computer" whose only abilities are: to deal only with zeros and ones; to be able to change any single digit to a 1 if it is a 0 and to a 0 if it is a 1; and to choose between two possible next instructions depending on whether the digit was a 0 or a 1.)

Use of the xx-Register

As an example of the use of the xx register, take Fred's case. Fred has smuggled 100,021 pounds sterling out of England and is exchanging them for dollars in Lucerne. The rate of exchange offered him there is 2.80173 dollars per pound. Fred wishes to use TAC to tell him the worth of his fortune in dollars. (He does not care how the people in Lucerne decided on the rate of exchange to so many significant figures.) The program is, of course, quite short, assuming that the number of pounds to be converted is punched as a 7-digit number on a tape.

00	R	39	x2	00	01	reads number of pounds into x2
01	M	x2	x3	xx	02	100021 x 280173 = 28023183633 → xx, putting +00000280 in x1 and +23183633 in x2
02	D	xx	x4	xx	03	C(xx)/1000 = 28023184 → xx, putting +00000000 in x1 and +28023184 in x2
03	P	27	x2	$I	04	prints $280231.84
04	P	89	x2	.I	05	
05	H	00	00	00	00	stops, ready to start at zero if necessary
x1	+	00	00	00	00	used as temporary storage during program to store first pounds, then thousandths of cents, then cents
x2	+	00	00	00	00	
x3	+	00	28	01	73	rate of exchange in thousandths of cents per pound.
x4	+	00	00	10	00	necessary constant.

There are several things worth noting about this program. One is that since the result of the division by 1000 is known to be less than 99,999,999, the result will fit into register x2 alone. The fg address of the D instruction could therefore be x2 rather than xx, the difference being only that if address xx is specified, +00000000 will be put into x1, whereas if x2 is specified, C(x1) will remain unchanged as +00000280. Similarly, in the multiply instruction, if xx had been given as the bc address in place of x2, the result would have been unchanged in this case, since C(x1) = +00000000 initially so that C(xx) = C(x2); but this of course is not always true.

Another approach to the problem would be required if Fred had had a million and 21 pounds instead of his paltry tenth of a million. Then the total number of cents would exceed 99,999,999. This can be handled here by thinking of the dollars and the cents separately. Suppose after multiplying 1000021 by 280173, getting 280178883633 in xx, we had multiplied C(xx) by 1000 rather than dividing it by 1000. C(xx) would then contain 280178883633000 millionths of a cents, with +02801788 in x1 and +83633000 in x2. Clearly C(x1) is the number of dollars, and C(x2) is the remainder in millionths of cents, and we can simply print the two parts separately. Another difficulty arises,

however, if we want the result rounded off in the normal way. We could divide by 1,000,000 and get 2 in x1 and 80178884 in x2, which gives the correct result, still in two registers. Unfortunately, not having the split occur between the dollars and the cents means that we must use three print instructions, one for the $2, one for the 801788 and one for the .84. Alternatively, we could do the rounding by merely adding +00500000 to xx, which would give +02801788 (the correct number of dollars) in x1 and +84133000 (in which the first 2 digits are the correct number of cents) in x2. Our program in this case would be:

00	R	29	x2	00	01	10000021 → x2
01	M	x2	x3	xx	02	=280178883633 → xx (+00002801 → x1, +78883633 → 02)
02	M	xx	x4	xx	03	+280178883633000 → xx (+02801788 → x1, +83633000 → x2)
03	A	xx	x5	xx	04	+280178884133000 → xx (+02801788 → x1, +84133000 → x2)
04	P	39	x1	$I	05	prints $2801788
05	P	23	x2	.I	06	prints .84
06	H	00	00	00	00	stops
x1	+	00	00	00	00	temporary storage
x2	+	00	00	00	00	
x3	+	00	28	01	73	constants
x4	+	00	00	10	00	
x5	+	00	50	00	00	

Numerical Shift Operation

One other thing worth noting in connection with Fred's problem is this -- multiplying or dividing by 1000 or by other powers of ten appears to be something one would often want to do. Clearly it should somehow be made easier both for the programmer to require and for the computer to execute a multiplication or a division by a million, say, than by 1279413 or by any other arbitrary number, for all that is required in multiplying or dividing by powers of ten is to shift the individual digits to the right (for dividing) or to the left (for multiplying). In almost all computers there are special instructions to shift right and shift left. TAC is no exception. There are in fact two kinds of shift operations in TAC, one to perform the numerical shifting equivalent to multiplication or division, with proper rounding and with the sign not shifted, and a second to shift ("logically") all characters, digit or not, sign or not, without rounding. The former is defined on the following page; the latter will be described in chapter 5.

Name	Code	Function
Numerical		
shift left	N+	C(de) x 10^c → fg
shift right	N-	C(de) ÷ 10^c → fg

In other words, N shifts the number in de to the left c places if b is + or to the right c places if b is - and places the result in fg.

Let us now take an example of N (and all of the other instructions introduced thus far). Carolyn has been selling homemade candy for 73¢ a pound. When someone buys more than one, they often ask for a quantity discount, and Carolyn has haphazardly given various discounts, amounting usually to about 5% on 2 pounds, 15% on 6, and never over 30%. Bob wants to make life easy for his wife and decides to mechanize the computing of the discount price. He tabulates a few of the discounts that Carolyn has given before and decides on the formula: % discount = 30 - 150/(q+4), where q represents the number purchased. Thus for one pound, q = 1 and the discount = 30 - 150/5 = 0%, for q = 2 the discount = 30 - 150/6 = 5%, and so on, never exceeding 30%. Bob writes the following TAC program for his problem, and the very first customer orders 96 pounds. Bob punches these two digits on tape (he only allowed for two digits at a maximum), and the computation TAC performs is described alongside the program.

00	R	89	x3	00	01	Places 96 from tape into righthand end of x3
01	A	x3	20	x2	02	96 + 4 = 100 → x2
02	D	21	x2	x2	03	150,000/100 = 1500 → x2
03	S	22	x2	15	04	30000 - 1500 = 28500 → 15
04	M	x3	x4	x5	05	96 x 73 = 7008 x5
05	M	x5	15	xx	06	7008 x 28500 = 199728000 → xx
06	N	-5	xx	x2	07	199728000 - 100000 = 1997 → x2
07	S	x5	x2	x2	08	7008 - 1997 = 5011 → x2
08	P	67	x2	¢I	09	Print ¢50, first digit of net price
09	P	89	x2	.I	10	Print .11
10	H	00	00	00	00	Stop
15			~~~			← receives 1000 x % discount
20	+	00	00	00	04	}
21	+	00	15	00	00	} constants
22	+	00	03	00	00	}
x1			~~~			} used in calculating % and total discounts
x2			~~~			}
x3	+	00	00	00	00	- receives the quantity 96 in the right end, leaving +00000096.
x4	+	00	00	00	73	- list price per pound
x5			~~~			- receives total list price

Conditional Sequence - The Comparison Instruction

The last of the four instructions (others were M, D, and N) for operating on numbers, rather than on instructions, alphabetic words, or other logical information, is C, for Compare numberically. It is in many ways the most important of the four, and in it lies the real heart of the automatic digital computer. C permits varying the instruction sequence to be followed depending on the outcome of the computation up to the point in question. A card programmed calculator and other tape- and card-sequenced devices are capable of elementary decision making of the same sort, but only the stored-program computer is capable of repeating certain parts of a computation automatically for a prescribed number of times or until a desired result is approximated closely enough

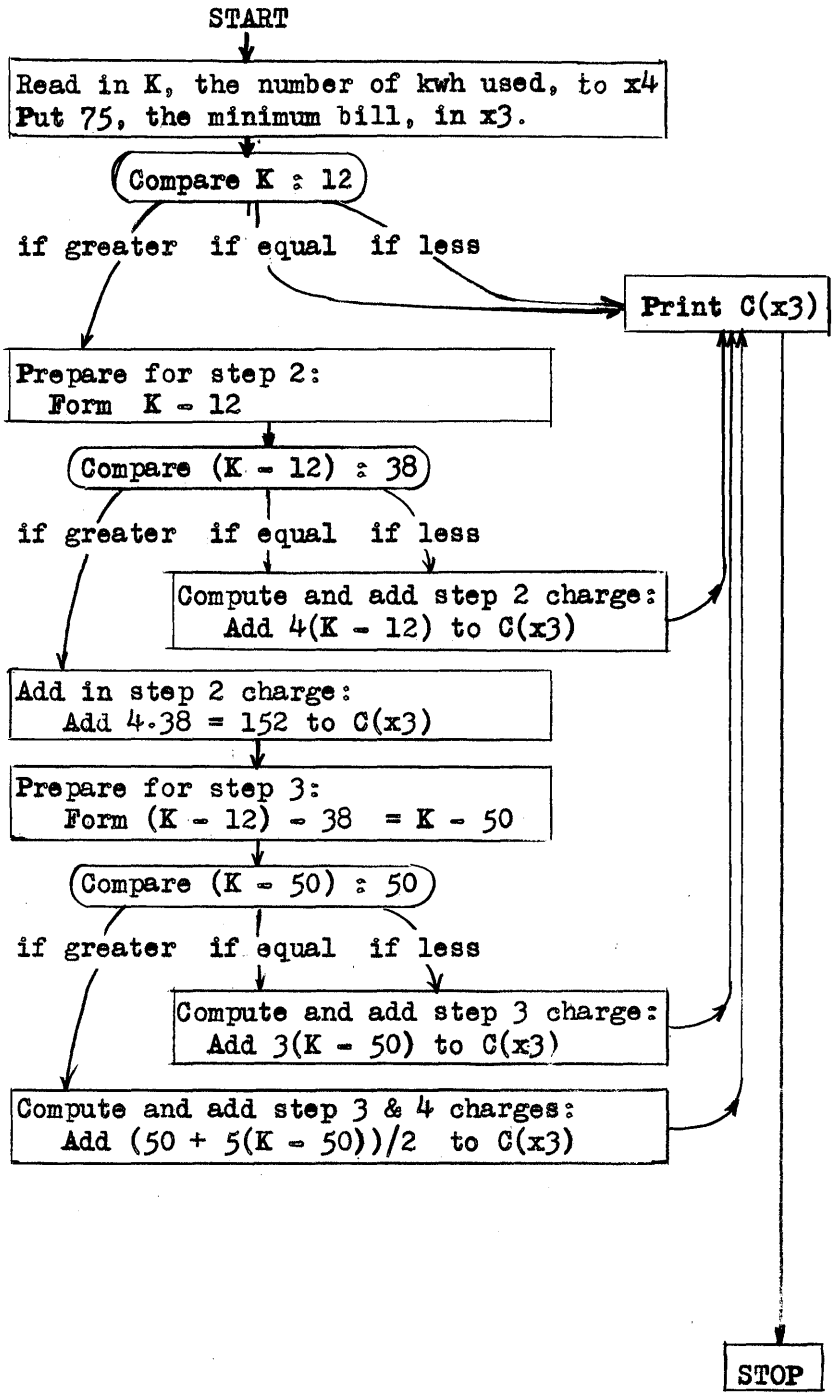
Name	Code	Function
Compare numerically	C	take the next instruction from: fg if C(bc) < C(de), hi if C(bc) > C(de), next register consecutively if C(bc) = C(de).

Notice that this instruction does not store or print any result. Its only outcome is the detection of one of three different conditions: either C(bc) is larger than C(de), or it is smaller, or they are equal. As used here, the symbols > and < , read "is greater than" and "is less than", refer to the sign as well as the magnitude of the numbers in question. Thus, $3 > 2$ (i.e. 3 is greater than 2), $4 > -5$, $-2 > -7$, $0 > -97$, etc. If C(bc) = C(de), the next instruction is taken from the register immediately following the register in which the C instruction was found. Naturally bc and de may be set equal to one another, or either may be made equal to the next consecutive address. C will lead to a post-mortem if C(bc) and C(de) are not both strictly numerical in form.

The comparison instruction will of course show up in a number of examples in later chapters. A simple example here will suffice to illustrate the principle involved.

The Cambridge Electric Light Company charges residential subscribers in 4 steps: 75¢ for the first 12 kilowatt-hours or less, 4¢ each for the next 38, 3¢ each for the next 50, and 2 1/2¢ each for the rest. Charlie used 146 kwh in March. The following program is intended to make TAC compute his light bill and others like it. The general scheme may be seen rather better in a diagrammatic form showing questions and alternative answers. Diagrams such as this are commonly used in planning a program and in documenting a program for later reference or for use by someone else in actually coding the program. They are generally referred to as "flow diagrams".

```
01  R 79 x4 00 02
02  A 03 x0 x3 04
03  + 00 00 00 75
04  C x4 07 08 05

05  P 67 x4 $I 06
06  P 89 x4 .I 28
07  + 00 00 00 12
08  S x4 07 x4 09
09  C x4 13 14 10

10  M x4 11 x4 12
11  + 00 00 00 04
12  A x3 x4 x3 05
13  + 00 00 00 38
14  A x3 15 x3 16
15  + 00 00 01 52
16  S x4 13 x4 17
17  C x4 21 22 18

18  M x4 19 x4 20
19  + 00 00 00 03
20  A x3 x4 x3 05
21  + 00 00 00 50
22  M x4 23 x4 24
23  + 00 00 00 05
24  A x4 21 x4 25
25  D x4 26 x4 27
26  + 00 00 00 02
27  A x3 x4 x3 05

28  H 00 00 00 01
```



Program and Flow Chart for Cambridge Electric Light Company Problem

Exercises

Each of the following exercises is independent of the others in the set, so that any register may be used for one purpose in one problem and for a different purpose in another. Unless there are special reasons, the instructions should be placed more or less consecutively, starting at register 01. Exercise #5 and those beyond generally leave unspecified the locations at which to store the data and constants. Ordinarily, this allocation should end up with constants interleaved as needed in the instructions, or grouped separately on the drum, and with data and intermediate results stored, if possible, in the x registers. In general these problems if done in a straightforward way may be assumed not to give rise to numbers too large to fit in one register, but some thought should always be given to this possibility. One should also take care not to throw away precision by dividing before multiplying, etc. Each program should come to a halt when the job is complete. Do not look for anything difficult hidden in these problems. The first ones are just as simple as they seem. The number in parentheses at the end of each exercise is a reasonable par for the number of registers required in storing instructions, constants, data, and results, including those explicitly designated in the statement of the problem.

3.1. Ellie has placed in register x3 an order for a certain number of cups of tea. In making tea she uses one teaspoonful for each cup wanted and one for the pot. Assuming that register 51 contains +00000001, write a program to put in register x4 the number of teaspoonsful of tea that Ellie should use. (5)

3.2. The gross weights in pounds of four different items to be shipped to New York are stored in registers x4, x5, x6, and x7. Write a program to put the total weight of the shipment into x8. (9-11)

3.3. Morgan's rate of pay, in cents per hour, is in register x3, the number of his withholding tax exemptions is in x4, and the number of hours he worked last week is in x1. The quantity 1300, which is the number of cents exempt per week per exemption, is in register 51. Write a program to put Morgan's withholding-taxable earnings into register x2. (9-10)

3.4. The length and width in inches of Herb's rectangular coal bin (which is of a modest size) are stored in registers x3 and x4, while those of Bob's bin are stored in x6 and x7. The present level depth in inches of the coal in Herb's bin is stored in x5 and that of Bob's in x8. Herb changes over to oil heat and ships his coal to Bob. Write a program to modify the contents of x8 to the proper new depth. (14)

3.5. Centigrade temperature can be converted to Fahrenheit by the formula: Fahrenheit temp. = 32 + (9/5)(Centigrade temp.). The temperature of the drinks Maurice serves has been measured in Centigrade degrees (it is 19°) and recorded in register x9. Write a program to place the equivalent Fahrenheit temperature, to the nearest degree, (which would be 66°), into register x8. (9)

3.6. The Halcyon Investment Trust, with trusts as large as a half-million dollars, has declared a scientifically-computed dividend of 3.41785%. Write a program which will read 8 digits, specifying an amount in cents, from punched tape, calculate the dividend to the nearest cent, and print it as an 8-digit, unpunctuated number. (8)

3.7. The Delphi Electric Light Company wants its electricity bills computed automatically. Initially, the readings of one customer's meter last month is placed in x3 and the reading this month in x4. The Delphi meters read only 4 digits. Write a program which will put the number of kwh. used into register x5, allowing for the case in which the meter has turned over -- so that, for example, this month's reading is 0173 while last month's was 9941. (8)

3.8. Dean and Jack need TAC's help in playing two-card stud, in which each player draws two cards and the winner is the holder of the higher pair, or, if neither has a pair, of the highest single card, always without regard for suit. The numerical value of Dean's two cards are in registers x3 and x4, those of Jack's in x5 and x6. Write a program which will print "Dean wins", "Jack wins", or "Its a draw", depending on the cards each holds. (17-20)

3.9. Kewa of Tagore exacts as tribute from his victims a number of kruls equal to the eighth power of their waist measurement in inches (i.e., the product of the measurement multiplied by itself seven times; for example, $2^8 = 256$, $10^8 = 100,000,000$ and $40^8 = 6,553,600,000,000$). One dollar equals 10,000,000,000 kruls (ten megakilokruls). Alan's waist has just been measured and the result placed in register x3. Write a program to print the tribute Alan must pay in dollars and cents, to the nearest cent, allowing for a 60 inch waist ($16796.16), without zero suppression. (11)

3.10. Martin makes short-term loans of less than $400. On these he charges 12% simple interest and accepts payments of any amount at any time. Each payment is first applied to payment of the simple interest on the previous balance for the number of days since the last payment, and the remainder of the payment is then subtracted from the balance. When a payment is received, Tweetie, Martin's office girl, punches a tape which contains the previous balance in cents (5 digits), the date of the previous payment (using 2 digits for the date, 2 for the month-numbered 01 to 12, and 2 for the year), the date of the present payment (in the same form), and the amount of the present payment in cents (5 digits). For example, if Murphy's balance after his last payment on Sept. 13, 1953, was $183.69 and he now pays $25 on Aug. 16, 1954, Tweetie types, with no punctuation at all, the digits 18369130953160854025 00 , which breaks down in the following way:

1 8 3 6 9 1 3 0 9 5 3 1 6 0 8 5 4 0 2 5 0 0

¢ balance 13 Sept 53 16 Aug 54 $ paid

Assuming a 360-day year of 12 30-day months, one can find the number of days between any two dates, such as between 17/09/53 and 13/08/54, by forming the difference in years times 360 plus the difference in months times 30 plus the difference in days; for example (54-53)360 + (08-09)30 + (16-13) = 333. The simple interest is simply the balance times 0.12 times the number of days divided by 360; for example, 18369 x 0.12 x 333/360 = 2039. The new balance is, of course, the previous balance plus interest minus payment: for example, 18369 + 2039 - 2500 = 17908. Write a program which, given a data tape prepared by Tweetie, will tell Murphy or any of Martin's customers what is his new balance by printing it in dollars and cents (e.g., $179.08) after any payment. (30)

4. Cycles, Counting; Modification of Instructions

Cycles

Problem: the output of a figgle factory is limited solely by the supply of jiggle-nuts, of which 77 are required in each figgle, and is seldom more than five a day. Jiggle-nuts are perishable; they are delivered fresh each morning, and any left over must be sold at the end of the day. The number delivered this morning is given in xl. Print the number that will have to be sold today.

This is essentially a question of finding a division remainder. TAC does not do this directly; it rounds off quotients and gives no remainder. We could actually find the remainder from the rounded quotient by a process of multiplication and subtraction, but there is a more direct way which is quicker if the quotient is small. We simply duplicate the operation of the figgle factory, subtracting 77 from the pile of jiggle-nuts as each figgle is made, until finally there are not enough left to make a further figgle.

The calculation is largely repetitive. We can make a neat compact routine by writing the subtract instruction only once, and arranging for TAC to obey it repeatedly. We make it part of a cycle.

We could easily make it a cycle on its own, by making its fourth address equal to the address of the location containing it, so that it is always followed by itself. This, however, would not cause the repetition to stop; the machine would go on subtracting endlessly (or until stopped by an excessively negative result). Somehow the machine must decide whether or not to repeat the subtraction, and this decision must be made after each repetition.

For this purpose we can use the conditional instruction. If the number of jiggle-nuts remaining is greater than or equal to 77 a further figgle can be made; if it is less than 77 the process is finished. We can code it thus:

```
00 | C  xl  04  01  02      test          )
01 | S  xl  04  xl  00      subtract 77   ) cycle
02 | P  89  xl  RS  03      print
03 | H  00  00  00  00      stop
04 | +  00000077
```

Counting

Henry has succeeded in producing one specimen of a rose plant with fine emerald green blooms. He decides to propagate it by taking one cutting per plant each year, but cuttings cannot be taken from a plant until it is two years old. His first plant is now one year old. Print the number that he will have after twenty years, assuming that he is an infallible gardener and none of his plants ever die.

Let us store the total number of plants at any given time in xl,

and the number that are more than one year old in x2. Then for each year TAC must add C(x2) to C(x1), and replace C(x2) by the old C(x1). It must perform these operations 20 times, and stop. Again the program is repetitive; again we will use a cycle.

The basic operations are caused by the instructions

```
00 | A  x1  x2  x1  01
01 | S  x1  x2  x2
```

As before, we could make an endless cycle simply by writing 00 at the end of the second instruction, but if the process is to terminate we must include a conditional instruction in the cycle. There is a difference, however, between this example and the last. There the number of jiggle-nuts itself provided the condition on which to base the decision whether to repeat. Here we do not know what Henry's roses will be doing after twenty years; the decision must be based simply on the number of years which have elapsed. This essentially involves counting the years; TAC must be made to keep a tally as it goes along.

We will keep this tally in x3. The complete routine could be as follows:

```
00 | A  x1  x2  x1  01 ]  the business          ]
01 | S  x1  x2  x2  02 ]                        ]
02 | A  x3  06  x3  03    add one to tally      ] cycle
03 | C  x3  07  04  00    test for completion   ]
04 | P  59  x1  RS  05    print
05 | H  00  00  00  00    stop
06 | +  00000001
07 | +  00000010

x1 | +  00000001          total plants
x2 | +  00000000          mature plants
x3 | +  00000000          tally
```

Minimal Latency Coding of Cycles

To make them easier to follow, the foregoing examples have not been coded for minimum latency. In practice one frequently finds that most of the machine's time is spent in obeying cycles which appear to be only a small part of the whole routine, simply because they are obeyed much more often than the rest of the routine is. Hence it is worth while coding these cycles for minimum latency, even if the rest of the coding is not done in this way.

Modification of Instructions

The Consolidated Glue Corporation markets glue in containers of fifteen different sizes, numbered 1 to 15. The capacities of these sizes (in hundredths of a pint) are listed respectively in locations 20 to 34. An order is received; the number of containers required is given in x3 and the size number in x4. Print the amount of glue

required to fill this order, to the nearest gallon.

This problem unavoidably requires the use of an instruction of which one address is not known when the program is written. Some instruction must pick out from the list the capacity of the container required, and the location of this information depends on the number given in x4 when this calculation begins. The routine itself must therefore cause the machine to construct the variable instruction before using it. This may be done as follows:

```
00 | N  +4  x4  x1  01     Shift size left to de position.
01 | A  07  x1  02  02     Put appropriate instruction in 02
                            multiply capacity by number of containers.
03 | D  xx  08  x1  04     Divide by 800 to get gallons.
04 | P  49  x1  RS  05     Print result.
05 | P  17  09  II  06     Print "gallons".
06 | H  00  00  00  00     Stop.
07 | M  x3  19  xx  03
08 | +  00000800
09 | g a l l o n s  00
```

Note that no word has been written for register 02, although the second instruction tells TAC to look in 02 for its next instruction. The reason is that before TAC looks for this instruction, a suitable instruction will have been put there. This is done by the instruction in 01, and the instruction that gets put in 02 is

M x3 (19+s) xx 03

where s is the size number.

Programmed Switches

In the previous example we had to shift the given number into the second address (de) position so that, by addition, it could be used to construct the second address of the instruction to be placed in register 02. In a similar way <u>any</u> address in an instruction can be modified by the machine itself. If the fourth address is modified the effect is interesting: it causes TAC to take its next instruction from a location which depends on the numbers from which the fourth address was constructed. This may lead the machine into one of a number of quite different courses of action.

This way we can tackle such a problem as the following. Bingo Products Inc. has ten classes of employees, identified by code numbers 1 to 10. Each class has its own rules for calculating wages, all quite different. An employee's code number is given in x1; print it, and proceed to calculate his wage. The routine for this might begin this way:

```
00 | A  02  x1  01  01     put appropriate instruction in 01.
                            print code number and go to corresponding
                               instruction in 03...12.

02 | P  89  x1  RS  02
03 |
 . |  first instructions for
 . |  each type of wage
12 |  calculation.
```

Instruction Modification Within a Cycle

Problem: Chin Foo has a tape punched with a list of his 57 customers. None has more than nine letters in his name, and each name is followed by spaces to make a total of nine characters. Also a carriage return symbol precedes each name. Read these names into registers 41 through 97.

This is another repetitive job and we shall use a cycle in the program. However, the operations are not exactly the same at each repetition: the first name must go into 41, the second into 42, and so on. Hence the instruction that places each name in the store must be changed at each repetition; the appropriate address will have to be increased by one each time.

To end the repetitions we must again include something equivalent to counting; we shall simply count from 0 to 57 (although this is not the most efficient way of programming it, as we shall see later). The complete routine is:

00	R	11	x0	00	01	discard carriage return symbol
01	R	19	41	00	02	read name into its register (41...97)
02	A	01	06	01	03	increase 2nd address in 01
03	A	x1	07	x1	04	count
04	C	x1	08	05	00	test for end of job
05	H	00	00	00	00	
06	+	00	01	00	00	
07	+	00	00	00	01	
08	+	00	00	00	57	
x1	+	00	00	00	00	tally

Resetting

Note that after the above job is done the second instruction remains as R 19 98 00 02, and C(x1) remains as +00000057, so that if we tried to use this routine a second time it would not work. If it were to be used as part of a big routine (i.e. as a "subroutine"), and were intended to be obeyed several times within the big routine, it would have to be refreshed between applications by having C(01) and C(x1) reset to their initial values. This resetting would have to be done by means of instructions obeyed between successive applications of this name-reading subroutine, preferably immediately prior to each application. Usually such instructions are written along with the subroutine, making the whole subroutine self-resetting.

Exercises

4.1. A bottle contains 15 ounces of horrible medicine. Andrew has to take a certain dose (stated in hundredths of an ounce in register x1) each day until there is not enough left to continue; he may then throw away the dregs. Print to the nearest hundredth the number of ounces which he will have the pleasure of discarding. (Use a cycle.) (6)

4.2. x1 contains an amount of money in cents; x2 contains a rate of interest in tenths of percent (e.g. C(x2) = 30 represents 3 percent.) Print the total after 10 years, assuming that the interest is compounded annually to the nearest cent. (12)

4.3. Using the same data as in question 2, print the least number of complete years after which the total is at least double the original principal. (12)

4.4. How Foo has a TAC input tape bearing the names of his 200 customers; each name consists of nine letters and is followed by a sign and a five digit decimal number which is the amount in cents of the customer's credit (a minus sign means that the customer owes How money).

Write a routine that will read the tape and print out the names and debts of all customers owing $100 or more. (12)

4.5. Chin Foo has placed the names of his 57 customers in registers 41 through 97. Each shirt received by his laundry is marked with the customer's serial number (1 through 57). Read one shirt number (2 characters) from the input tape and print its owner's name. (5)

4.6. The Tipperary Trust Company has 68 customers whose balances (in cents) are stored in registers 20 through 87. Print the sum of the balances, which is less than a million dollars. (12)

4.7. With the data of question 6, subtract from each balance a fixed charge of 75 cents. (10)

5. COMPARING; GROUPING AND PACKING

"Logical" Comparison of Words in TAC

Chin Foo, having placed his customers' names in locations 41 through 97, wishes to have TAC check whether they are in alphabetical order.

For this problem we need to have some way of letting TAC decide which of two words comes first alphabetically; this involves comparing alphabetical characters. The C instruction will only compare numbers. We therefore make use of a further instruction:

Name	Code	Function
Compare Logically	K	Take next instruction from: fg if C(bc)$>$C(de)* hi if C(bc)$<$C(de)* or next register consecutively if C(bc) is identical with C(de)

*The meaning of the symbol $>$ is defined here in such a way that if C(bc) comes later alphabetically than C(de), then C(bc)$>$C(de), and vice versa. In detail, the rule is as follows: compare the characters of C(bc) with those of C(de) column by column from the left until corresponding characters differ. Whichever of these characters comes later in the alphabet belongs to the "greater" word. Since TAC registers may contain a variety of symbols besides the letters of the alphabet, and since we distinguish between capital and small letters, the alphabet is extended for this purpose to include all the sumbols that TAC uses. The full list is given in the TAC summary and is repeated here:

T C £ ¯ /) - + . I 0 2 4 6 8 $^{0\ 2\ 4\ 6\ 8}$ a b ... z

R B H § : (_ = , S 1 3 5 7 9 $^{1\ 3\ 5\ 7\ 9}$ A B ... Z

(Since the decimal digits appear here in their natural order, the K instruction can be used to compare positive numbers. It is however a little untidy when negative numbers are involved, so a separate "compare numerically" instruction has been provided.)

For Chin Foo we need to make 56 comparisons, namely C(41) with C(42), C(42) with C(43), etc. As this is a repetitive kind of job we shall again use a cycle, and since we must operate on successive registers we shall have to modify an instruction within the cycle. The coding is very similar to the last example in Chapter 4, except that we now have to modify two addresses in the same instruction; however, both can be modified simultaneously.

As a further refinement we shall show how it is possible to dispense with the separate tally for counting repetitions, and instead to base the decision whether to repeat the cycle on the instruction which is being modified. The latter starts as K 41 42 14 11 and ends as K 96 97 14 11 ; when it has passed this final value the process may be stopped. For this comparison also we shall use a K instruction; since C is intended for use with pure numbers it would stall on any word containing letters, even though the essential comparison is merely concerned with the addresses, which are numbers.

10	K 41 42 14 11	compare consecutive names	
11	A 10 16 10 12	increase addresses in 10	cycle
12	K 10 17 13 10	test for end	
13	P 13 18 R. 15	print "O.K."	
14	P 48 18 R. 15	print "WRONG"	
15	H 00 00 00 00	stop	
16	+ 01 01 00 00		
17	K 97 98 14 11		
18	0 .K WR ON G.		

Grouping of Registers

We have seen that register xx may be used for arithmetic giving rise to astronomical figures such as the national debt (between 10^8 and 10^{16}). This is the only part of TAC that may be used to perform arithmetic on such large "double length" numbers directly. However, there is no reason why such numbers should not be stored in any two TAC registers; we are perfectly free to regard any two registers as holding two parts of a single number if we wish. Similarly we may break an item of alphabetical information into parts occupying separate registers. Names, for example, frequently contain more than nine letters, but they can be artificially split between the ninth and tenth letters and the two parts stored in different registers. Since there is never any strong reason for doing otherwise, the parts are nearly always stored in consecutive registers.

Business applications rarely call for long numbers to be stored; however, long alphabetical items often arise, and so does the problem of determining their correct alphabetical sequence.

Problem: two names, each of 18 characters, are stored in TAC; the first in registers 15 and 16, and the second in registers 17 and 18. Print them in alphabetical order.

To compare them we must first compare their left-hand halves; if these are identical, we must then compare their right-hand halves. If the left-hand halves are not the same the second comparison is unnecessary. We might code the routine thus:

```
00 | K 15 17 02 06          compare L.H. halves
01 | K 16 18 02 06          compare R.H. halves
02 | P 19 17 RI 03
03 | P 19 18 II 04            print with
04 | P 19 15 RI 05            order reversed
05 | P 19 16 II 10
06 | P 19 15 RI 07
07 | P 19 16 II 08            print in
08 | P 19 17 RI 09            given order
09 | P 19 18 II 10
10 | H 00 00 00 00          stop
```

Packing

Just as some items are too long to be accommodated in registers of a fixed length, so also some items are much shorter than the registers available. There is of course no difficulty in putting a short item in a long register; unused digits can always be filled in with zeros in the case of a number, or with ignores in case of names, etc. However, it is more economical in storage space if two or more such items can be "packed" into a single register.

There is nothing mysterious about this: the items are just placed side by side, in different digits of the same register. The need frequently arises, however, to separate and rearrange such items. Two instructions in TAC's instruction code are designed to meet this need. The first is:

Name	Code	Function
Extract	E	In those columns, and only those, in which C(de) has odd* characters, replace the characters of C(fg) by the characters occupying corresponding columns in C(bc), without altering the other characters in C(fg).

* (See top of next page)

*Where the characters of C(de) are decimal digits the meaning of "odd" is obvious; in other cases, characters in the lower line of the list at the bottom of page 5-1 are considered odd, and those in the upper line even.

In most applications C(de) will be a prearranged mixture of appropriately even or odd digits -- e.g., 0's and 1's. Suppose, for example, that register x1 contains the day of the week in full (ending with ignores if necessary), and that x2 contains a positive number, less than a million. We wish to abbreviate the day to its first three letters so that we may pack it and the number into x1 together. This requires just one instruction and one other word:

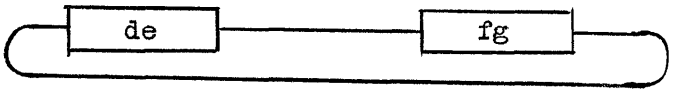
```
00| E x2 01 x1 02

01| 0 00 11 11 11
```

The effect is to replace all the letters of the day except the first three by the digits of the number.

Packing, and the reverse process of unpacking, also frequently involve shifting characters to the right or left within registers. The N instruction is restricted to be used with numbers only, and there is a further instruction for shifting:

Name	Code	Function
Logical Shift	L	Shift C(de) and C(fg) cyclically c places; left if b = +, right if b = -

Note firstly that two registers are involved; the contents of both are shifted. Secondly the shift is cyclic; characters shifted off the end of either register appear at the opposite end of the other register. The two registers form a closed loop thus:



Characters merely move round the loop and none are lost.

If, in the last example, we wished to have the number (without sign) in the left-hand 6 digits of x1, and the abbreviated day in the right-hand 3, then we would need one instruction alone:

```
00| L -6 x1 x2 01
```

This would, however, upset C(x2).

Example of Unpacking

A positive sum of British money is represented in x1 thus: number of pounds (without sign) in digits 1-5, number of shillings in digits 6 and 7, number of pence in digits 8 and 9. Put these numbers into registers x2, x3, and x4 respectively.

The simplest procedure is to shift C(x1) to the right, extracting each number as it arrives in the correct digital position, thus:

```
00| E  x1  06  x4  01      extract pence
01| L  -2  x1  x0  02
02| E  x1  06  x3  03      extract shillings
03| L  -2  x1  x0  04
04| E  x1  07  x2  05      extract pounds
05| H  00  00  00  00
06| 0  00  00  00  11
07| 0  00  01  11  11
x2| +  00  00  00  00
x3| +  00  00  00  00
x4| +  00  00  00  00
```

Exercises

1. The results of British football matches are decided solely on the number of goals scored and are always coded thus:

1 means the "home" team won.
2 means the "away" team won.
x means a draw.

Registers 50 through 99 contain the scores of 50 matches; digits 2 through 5 in each register give the number of goals scored by the home team, and digits 6 through 9 the number scored by the away team. Print the 50 results. (21)

2. Too Foo's laundry has attracted only 23 customers, which is fortunate for TAC because each customer's name occupies two consecutive registers; altogether registers 54 through 99 are used. Print a suitable word to indicate whether the order is alphabetical. (13)

3. Today's date is commonly written in England as 17/8/54. Suppose that this is packed into register x1, allowing 2 digits for each part, and ending with a period, thus: 17/08/54. Write a program that will convert this into the form AUG171954 and will handle any date in the 20th century. (28)

4. The order of precedence of the guests at the Maharajah's Banquets is determined primarily by the numbers of wives possessed; where these numbers are equal the guests are arranged in alphabetical order. Read a tape carrying 357 names each followed by the associated number of wives, and print out the name of the most distinguished guest. Each name is followed by ignore characters to a total of 15, and numbers consist of 3 decimal digits (it is considered unethical to possess more than 999 wives.) (15)

5. Too Foo (see question 2) has rearranged his list of customers to allow three registers per customer; the third holds the customer's credit balance in cents. The list now occupies registers 31 through 99. Read one name (18 characters) from the tape; if this is the name of a customer print the name and his credit balance, otherwise print "no record." (18)

8. INTRODUCTION TO SAC

TAC and SAC have been chosen to present two rather different conceptions of an electronic digital computer, and to include between them most of the basic features of logical design and coding conventions found in present-day computers.

SAC differs from TAC in the following important respects:

1. SAC has a single-address instruction code.

2. Whereas a TAC register holds any group of 9 symbols, which in a special case may be a number or an instruction or neither, a SAC register cannot hold anything other than a number or an instruction. Alphabetical items can only be stored by coding them to appear as numbers.

3. SAC has a "B-box" which improves the efficiency of the machine in operations involving counting and the modification of instructions.

4. The process which loads the program into SAC is considerably more elaborate than that for TAC and permits a program to be written in a more convenient form.

SAC Instruction Code

Since each instruction contains only one address it cannot describe such a big operation as is defined by a TAC instruction. For example, the operation

$$C(21) + C(22) \rightarrow 23$$

which is accomplished by one TAC instruction, involves reference to three storage registers and therefore requires three SAC instructions in succession. The first of these obtains C(21) from the store, the second adds C(22) to it, and the third puts the sum in 23. This is all one operation for TAC; in SAC the three steps are taken separately, each in response to a different instruction.

Some part of SAC must retain C(21) while waiting for C(22) to be obtained, and must hold the result of the addition while waiting for the third instruction to be obeyed. A very special register is provided for this purpose; it is called the accumulator.

The actual instructions required to tell SAC to perform the above addition are:

```
ccf  21    copy contents from register 21 into accumulator
add  22    add contents of register 22 to number in accumulator
cci  23    copy contents of accumulator into register 23
```

We shall be introducing further SAC instructions from time to time; meanwhile the following few will serve to begin with. The abbreviation AC stands for the accumulator; n is any address.

Code	Name	Function
ccf n	copy contents from	C(n)→AC
cci n	copy contents into	C(AC)→n
add n	add	C(AC) + C(n)→AC
sub n	subtract	C(AC) - C(n)→AC
mby n	multiply by	C(AC) · C(n)→AC
dby n	divide by	C(AC) ÷ C(n)→AC (rounded off)
tyn 0	type number	Print C(AC)
stp 0	stop	stop the computer

The above SAC instructions have nothing corresponding to the fourth address in TAC. In general, when a SAC instruction has been obeyed, the machine automatically looks in the next consecutive register for its next instruction. Only certain special types of instructions allow this sequence to be broken; we shall consider these shortly.

SAC Registers

SAC has 299 registers numbered 0 through 298. Register 0 always contains the number zero; also C(290) = 1, C(291) =10, C(292) = 100, C(293) = 1000, etc. up to C(298) = 100,000,000 (in general, C(290 + p) = 10^p for p = 0...8). Every other register (1 through 289) is capable of holding either an instruction (in which the address must be less than 299) or a signed number of up to 8 decimal digits in length. The accumulator may hold either an instruction or a signed number of up to 16 decimal digits in length. As in TAC, numbers are considered as integers, with the decimal point at the extreme right; fractions can only be stored by scaling.

It is assumed that the store of SAC is all of a high speed type; there is no division into rapid-access and slow-access registers as in TAC.

Example of SAC Program

The program on page 2-7, if recoded for SAC, would appear as follows:

	instructions		comments		numbers
16	ccf	7	z → AC	1	+5
17	mby	1	5z → AC	4	+3
18	add	6	y + 5z → AC	5	x
19	add	6	2y + 5z → AC	6	y
20	add	5	x + 2y + 5z → AC	7	z
21	mby	1	5x + 10y + 25z → AC		
22	cci	12	Gross → 12	12	Gross
23	dby	4	Gross/3 → AC	13	Gross/3 = J
24	cci	13	Gross/3 → 13	14	2J
25	add	13	2J → AC	15	Gross - 2J = A
26	cci	14	2J → 14		
27	ccf	12	Gross → AC		
28	sub	14	Gross - 2J → AC		
29	cci	15	A's share → 15		
30	stp	0	stop		

Transfer of Control

It has been mentioned above that certain instructions allow the normal consecutive sequence of execution to be broken, so that after one of these instructions has been obeyed the next instruction to be obeyed may not be in the next consecutive register. Such a break is called a "transfer of control." Of these special instructions, one (the first in the list below) is unconditional, i.e., it always causes a transfer of control. The others are all conditional, i.e., the location of the next instruction depends on some condition inside the machine.

Code	Name	Function
jmp n	jump unconditionally	take next instruction from register n, and continue consecutively from there.
jip n	jump if positive	ditto if C(AC)>0, otherwise ignore.
jin n	jump if negative	ditto if C(AC)<0, otherwise ignore.
jiz n	jump if zero	ditto if C(AC) = 0, otherwise ignore.
jix n	jump if excess	ditto if the absolute value of C(AC) exceeds 134,217,727; otherwise ignore.

The "jump if excess" instruction has been provided to assist the programmer in computations with numbers which might perhaps exceed the capacity of an ordinary storage register. Such numbers may be formed in the accumulator by multiplication or addition, up to a limit of 10^{16}, but cannot be copied into the store unless they are less than about 10^{8}. The above instruction helps one to design the program so that overlarge numbers are detected in the accumulator before an attempt is made to copy them out. The critical limit, as will be seen from the above description, is not exactly 10^{8} but is a little larger, due to the fact that numbers are actually stored as 27 binary digits and sign.

With these instructions, programs containing cycles and programs of the electricity bill type (page 3-11) can be coded. It is not possible to compare two numbers directly with one instruction, as in TAC, but such a comparison can always be made by first subtracting one number from the other and then testing the difference.

To give a simple example, the figgle factory problem might be coded as follows. It turns out to be simplest, in SAC, to continue to subtract 77 until the number of jiggle-nuts actually becomes negative, and then to correct by adding 77 once afterwards.

```
 1 | Number of jiggle-nuts delivered
 2 | +77
10 | ccf  1   put number delivered in AC
11 | sub  2   subtract 77                    }
12 | jip  11  }repeat if                     } cycle
13 | jiz  11  } positive or zero             }
14 | add  2   correct
15 | tyn  0   print
```

Some Further Instructions

The following instructions are also understood by SAC and are occasionally useful:

Code	Name	Function
cnf n	copy negatively from	-C(n)→AC
cmf n	copy magnitude from	C(n) copied into AC with positive sign, regardless of sign of C(n) itself.
xch n	exchange	C(AC)→n, C(n)→AC

SAC also possesses a second division instruction, for divisions involving whole numbers where a remainder is required as well as a quotient, the latter not being rounded off. (This makes it quite unnecessary to use a cycle for the figgle factory.) The remainder is placed in a special "remainder register," RR. Associated with this are two other instructions; all three are given below.

Code	Name	Function
dhr n	divide holding remainder	Divide C(AC) by C(n). quotient→AC remainder→RR
cri n	copy remainder into	C(RR)→n
jir n	jump if remainder	Take next instruction from register n if C(RR) is not zero.

When SAC obeys a dhr instruction it gives a remainder smaller than the divisor in magnitude and having the same sign as the dividend. The following relationship always holds:

quotient x divisor + remainder = dividend

The only way in which C(RR) can be changed is as the result of a dhr instruction.

The B-Box

The B-box is a device containing seven registers which are distinct and separate from the accumulator and remainder register and from the store, and which are called the counters or B-registers; they are denoted by the letters a through g. Each counter contains two integers

called the index and the criterion respectively. Provision is made for increasing the index by 1 and testing to see whether it has become equal to the criterion. The following single instruction combines these operations:

Code	Name	Function
jii n b	jump if counting is incomplete	increase i_b by 1, then take next instruction from n if $i_b < n_b$

Here i_b and n_b are the index and criterion respectively of counter b. Any one of the counters a through g may be used by substituting the appropriate letter for b.

It will be noticed that this instruction contains more than the usual operation section plus one address; it also contains a letter specifying which counter is involved. We shall see later that in fact most of the SAC instructions may have such a letter attached.

One important application of the B-box is to the counting of the repetitions of a cycle. Before we can use it in this way we need one further instruction:

Code	Name	Function
rst n b	reset counter	$i_b = 0$ $n_b = n$

Let us now apply this device to Henry's roses (p. 4-1).

```
 1 | +0          number mature
 2 | +1          constant

10 | ccf  2      put 1 in AC initially
11 | rst  20 a   b set to count 20 times
12 | xch  1      new mature = old total  }
13 | add  1      new total = old sum     } cycle
14 | jii  12 a   b count                 }
15 | tyn  0      print
16 | stp  0      stop
```

Throughout this computation the total number of plants in each year is held in the accumulator.

The B-box here acts as a useful auxiliary to the accumulator for the simple side-operation of counting. However, its most essential feature lies in a direct connection between the B-box and the control unit of the machine. The design of the machine makes it possible for instructions to be modified after being read from the store and before being executed by the control unit. This modification consists of adding the index of any counter to the address in the instruction.

To specify which counter index (if any) shall be added, the letter corresponding to that counter is simply added to the instruction. Suppose for example that $i_b = 6$.

Then the instruction

ccf n b

will copy into the accumulator not C(n), but C(n+6).

Note that the instruction as it stands in the store is not affected in any way by the counter. The addition occurs after the instruction has been read from the store and as it is about to be executed.

If no letter is written at the end of an instruction, no such modification will occur. The following instructions are like ccf in that a letter may be added to indicate that modification is required.

ccf, cnf, cmf; cci; add, sub, mby, dby, dhr; cri; xch; jmp, jip, jin,

jiz, jix, jir.

As an example of the use of this facility, we shall code for SAC problem 4.7 on page 4-5. This requires the subtraction of 75 from each number in registers 20 through 87. We use a cycle, in which two of the instructions refer to the address of the register being dealt with, and therefore must in effect be changed at each repetition. By using the B-box we may leave these instructions fixed in the store, and yet still have them refer to a whole series of registers.

```
 1 | +75
10 | rst  68  a        set to count 68 times
11 | ccf  20  a      } subtract charge    }   cycle:
12 | sub  1          |  from amount in    |   i_a =
13 | cci  20  a      |  register (20 + i_a) |   0.....67
14 | jii  11  a      | count              }
15 | stp  0          | stop
```

The foregoing description shows the principal uses of the B-box. There are four more instructions relating directly to it, which are occasionally useful.

Code	Name	Function
inc n b	increase counter	$i_b + n \rightarrow i_b$
		$n_b + n \rightarrow n_b$
dec n b	decrease counter	$i_b - n \rightarrow i_b$
		$n_b - n \rightarrow n_b$
jic n b	jump if complete	$i_b + 1 \rightarrow i_b$, jump to n if new $i_b > n_b$ or $= n_b$
cii n b	copy index into	$i_b \rightarrow n$

Exercises

Recode the following problems for SAC. Where quantities were given in X-registers of TAC, adopt any suitable registers in the store of SAC to hold these quantities.

3.2 3.4 4.2 4.3 4.6

9. SAC: EDITING, PACKING, SYMBOLIC ADDRESSES

Further Input and Output Instructions

The rin 0 instruction is the converse of the tyn instruction; it reads one whole number from the input tape and places it in the accumulator. The end of the number is denoted by a carriage return or tab.

In Chapter 8, the tyn instruction was described only in its simplest form, with the address 0. There are many variations, distinguished by inserting different address values, which can be used for printing numbers in a variety of different forms. The address values are listed, with their effects, in the SAC Summary.

In order to be able to accept and present data in alphabetical as well as numerical form, SAC has special instructions for reading and printing alphabetical characters. These are the ric 0 and tyc instructions. The former reads a single character from the input tape, and the latter prints a single character. The character may be any of those available on the Flexowriter; however, since most of these characters cannot be represented directly in SAC, they are coded as numbers inside the machine. The numerical equivalents of the various characters are given in the table headed "The Flexowriter Code". The ric 0 instruction places this number in the accumulator; the tyc instruction prints the character defined by its own address. For example, if the symbol b is read from the input tape by the instruction ric 0, the number 62 will be put in the accumulator; and the instruction tyc 62 will print this character. Notice that there are two forms of each character, depending on whether the printer is on upper or lower case; two special characters (71 and 75) serve to change the case.

One of the principal uses of tyc is to insert carriage return, tab, space and punctuation symbols into the results produced by SAC.

Both tyc and tyn normally use the delayed printer; the information is not printed during computation but is stored on magnetic tape, which is later played back and printed. The recording can be done without slowing down the machine appreciably, and the printing can be done while the machine is engaged on another problem. However, there is also a printer linked directly with the machine and this may be used if desired. Adding 100 to the address of a tyc or tyn instruction will cause the direct printer to be used.

To summarize, the instructions for reading and printing data are:

Instruction	Meaning	Definition
ric 0	read in character	read the next char. via the PETR into AC as a positive integer 77
rin 0	read in numerically	read the next complete integer via the PETR into AC

tyc m tyc 100+m	type character	record on delayed printer (m), or on direct printer (100+m), the Flexo. char. specified by the integer m.
tyn m tyn 100+m	type numerical	record on delayed printer (m), or on direct printer (100+m), C(AC) as specified by m.

Modification of Instructions in SAC

Although the B-box removes the need for changing instructions in many cases, there are still some cases in which it is desirable to alter an instruction in the machine during the execution of a program. This may involve performing arithmetic on the address part, or changing the operation part, or both.

In order to be able to distinguish such enterprising and ingenious feats of programming from mere mistakes, which tend to look the same, SAC forbids direct arithmetical operations on whole instructions. Instead, a way is provided of extracting and of replacing the address parts of instructions, which are numerical quantities, so that arithmetic may be performed on them alone. (For this purpose any counter letter that may be appended to an instruction is not considered part of the address.)

Code	Name	Function
caf n b	copy address from	copy address section of C(n) into AC
cai n b	copy address into	replace address section of C(n) by C(AC)

caf and cai instructions may have counter letters appended to them in the same manner as ccf, etc.

While it is in the accumulator, an address is regarded as a number and may be operated on arithmetically like any other number.

Whole instructions may be moved from place to place via the accumulator without causing a post-mortem, provided that they are not changed on the way. To set both the operation and address parts of an instruction during computation, the operation part should be set first by copying another instruction which has the same operation; the address can then be corrected if necessary by means of cai.

Packing of Alphabetical Data

As mentioned at the beginning of Chapter 8, a SAC register cannot hold any arbitrary group of symbols, but only a number or an instruction. All alphabetical data must therefore be coded as numbers inside the machine.

The ric and tyc instructions provide a means of conversion between individual characters and single numbers in the machine. To economize

in storage space, however, it is desirable to pack as many characters as possible into one register. Now each character is represented by a number of at most 2 decimal digits, whereas there are 8 decimal digits available in each register. A simple way of packing is therefore to use successive pairs of decimal digits in one register to hold 4 numbers representing 4 different characters.

The packing process can be done by shifting decimally, i.e. by multiplying by powers of 10; e.g. to read 4 characters and to pack them, left to right, in register 1, the program would be:

```
10 | ric 0      first character to AC
11 | mby 292    shift first character 2 places left
12 | cci 1      and copy into 1
13 | ric 0      second character to AC
14 | add 1      attach first character
15 | mby 292    shift both 2 places left
16 | cci 1      and copy into 1
17 | ric 0  ]
18 | add 1  ]   assemble first 3 characters
19 | mby 292]
20 | cci 1  ]   shift and copy into 1
21 | ric 0  ]
22 | add 1  |   assemble all characters and copy into 1
23 | cci 1  ]
```

or better,

```
10 | ric 0
11 | rst 3 a
12 | mby 292 ]
13 | cci 1   |
14 | ric 0   |  cycle 3 times
15 | add 1   ]
16 | jii 11 a
17 | cci 1
```

Unpacking can be achieved by successive division (using dhr) by 100; the remainders are the numbers representing the original characters, although they appear in the reverse order to that in which they were packed on the previous page.

It is worth inquiring what the effect would be if we used a number other than 100 for the multiplications and divisions when packing and unpacking. Clearly, unless we use some power of 10, the packed numbers will not be recognizable as distinct groups of decimal digits in the decimal form of the whole. However, there is nothing magical about the decimal notation, and we need not worry if the decimal number we get looks quite unlike any of the numbers we packed, provided that we can still unpack them successfully.

In fact, the process works with any number in place of 100, with one important restriction. The number used must be greater than the highest number to be packed. For example, the highest Flexowriter code number is 77, so that we may use 78 or any higher number in place of 100. What we are doing, in fact, is simply converting a number from the scale of 78 into the decimal scale, or into whatever scale we imagine SAC to use. Just as in the scale of 10 no digit exceeds 9, so in the scale of 78 no digit may exceed 77 -- and this is true of the numbers in the Flexowriter code.

In practice there is a limit to the size of numbers that can be accomodated in one register, and so the number resulting from the packing process must not be too large. This means that the number used as the base for conversion must not be too large. Thus we can pack 4 Flexo characters into one SAC register using the base 100, but we cannot do it with base 150, because that might lead to a number as great as $77 \times (150)^3$ which is a bigger number than a SAC register can hold.

Unfortunately we cannot get more than 4 Flexo characters into one register. Even using the smallest possible base (78), the packing of five characters might produce a number as great as 2887174367 which is too big for SAC.

In most machines there is a practical advantage in using a base which is a simple power of the number base used in the arithmetic unit of the machine; in a decimal machine, for instance, the base 100 would be particularly convenient for packing. The reason is that multiplication and division by such a number can be performed merely by shifting left or right, which is a very simple and rapid operation. Most machines (e.g. TAC) have special shift instructions that are quicker than the equivalent multiplications and divisions. SAC has no special shift instructions, but if a multiplication or division instruction refers to one of the registers 290 through 298 (which contain the powers of 10) it takes a time which depends on the power of 10 involved, but which is in any case quicker than an ordinary multiplication (see SAC Summary). From this point of view, therefore, SAC resembles a decimal machine.

Symbolic Addresses

At the beginning of Chapter 8 it was mentioned that SAC has facilities for automatically processing programs after they have been read from the input tape, and before they are executed, so that programs need not be written in precisely the form in which the machine will finally use them.

The most important feature of this process is the conversion of symbolic addresses. These are groups of symbols that may be written, if desired, in an instruction in place of a numerical address. When the program is processed by the machine before the calculation begins, all symbolic addresses are replaced by numerical addresses so that the execution of the program may proceed exactly as already described. A symbolic address merely represents, temporarily, some numerical address.

A symbolic address always consists of a lower case letter followed by a number less than 1,000: e.g., a1, b40, p792, etc. Its use lies in instructions which refer to any word that itself appears as part of the written program. Any word can be labelled with a convenient symbolic address merely by writing that address alongside it (for details see below). Instructions that contain this same symbolic address will then be automatically adjusted during loading, so that by the time the calculation begins they will all refer to the register containing the word.

The same effect can, of course, always be obtained in the old way simply by finding out which register the word will eventually occupy in the store, and by writing its actual numerical address in all instructions referring to the word. Using a symbolic address merely avoids having to predict what the numerical address will be. The advantage of this becomes more apparent when one is faced with the probability of having to rearrange large parts of the program, either to correct a mistake or to make some revision. Numerical addresses must be changed wholesale, but symbolic addresses are not affected.

A symbolic address, used to label a word, is sometimes said to be assigned to the word. The symbolic is written on the left of the word and separated by a comma, thus:

```
b3, 750
```

If the instruction

```
ccf b3
```

appears somewhere in the same program, then it will be adjusted during loading so that, when executed, it will cause the former word (i.e. the number 750) to be placed in the accumulator.

Example of Use of Symbolic Addresses

If the coding example on page 8-6 were written using symbolic addresses, it might appear as follows:

```
1|   a1, +0
     a2, +1
10|  b1, ccf a2
         rst 20 a
     b2, xch a1
         add a1
         jii b2 a
         tyn 0
         stp 0
```

Notice that only two numerical addresses have been written on the left, to indicate which registers are to contain the words written. As symbolic addresses are used, there is no need to fill in all the numerical addresses on the left; indeed, the idea of symbolic addresses is to make this unnecessary. If, however, we want the machine to put the various words in the same places as were used on p. 8-6, it must be told that there is to be a gap between registers 2 and 10. The number 10 on the third line tells SAC that the instruction ccf a2 is to be placed in register 10. In the absence of such an indication SAC places each word in the register following that in which the previous word on the program sheet was placed. Thus, there is no need to indicate where all the other instructions are to go; they will be placed consecutively.

In point of fact, since we are using symbolic addresses we do not really care very much where the instructions go. If the number 10 were omitted, the instructions would go into registers 3 through 9 but they would still work. We can moreover leave out the number 1 preceding the first word, for the loading process will automically put this word into register 1 unless we specify otherwise.

The symbolic address b1 labelling the first instruction is not, in fact, used in any other instruction in the program. The reason for attaching this label is to enable SAC to be told (without referring to an absolute address) where to begin execution of the program. This is indicated on the program sheet by writing, after the program,

```
b1|  start
```

These are the principal uses of symbolic addresses. A further use (the "assignment" of a word, for correction purposes, to a register to replace a word appearing earlier in the program) and some further details of tape preparation will be found in the Summary of Specifications for SAC.

11. SAC - INPUT TAPE PREPARATION AND POST MORTEMS

Input

The process of preparing coded programs for actual input to the computer is a simple and straightforward one. However, there are a few conventions that must be observed. These conventions are described below.

Instructions, integers, and control information must be typed by a Flexowriter tape perforating machine. In addition to performing the function of an electric typewriter, this machine produces a 7/8" punched paper tape. For each key depressed on the Flexowriter, a unique combination of holes or no holes is punched in each of 7 positions across this tape.

Coded programs are actually typed almost exactly in the form in which they are written. The basic rules are

1. Beginning of tape: the first line of the page must begin with the lower case letters f2s followed by the summer session identification number, 198, a dash followed by the programmer's number and another dash, followed by the number assigned to the particular tape by the programmer. This information will be used by the computer to maintain a log of computer operation. Following this information, but on the same line, the programmer may put any other convenient identifying information such as name(s), date, and purpose of the program. All identifying information must be on one line followed by a carriage return. For example:

f2s 198-5-2 Customer billing Billetdoux - 8/21/54

Following this line, the program is typed with the first instruction going into register 1 unless an absolute address assignment comes first.

2. Absolute addresses: typed as 1, 2, 3 decimal digits in the range 0-298.

3. Absolute address assignments: typed as an absolute address followed by a vertical bar. However, since register 0 permanently contains the number 0 and registers 290-298 permanently contain the numbers $10^0 = 1$, to 10^8 respectively, any attempt to assign the absolute addresses will result in a conversion post mortem (see Locating Mistakes in Programs). Any number of tabs or carriage returns can intervene between an absolute address assignment and the instruction or number to which it refers. Instructions and numbers following an absolute address assignment are assigned to addresses in sequence beginning with the absolute address assigned.

4. Floating addresses: typed as any lower case letter except 0 or 1 followed by 1, 2, or 3 decimal digits with the restriction that the first digit must be non-zero.

5. Floating address assignments: typed as a floating address followed by a comma. The floating address preceding the comma will be assigned a value equal to the address of the next register in normal sequence unless the floating address was preceded by an absolute address assignment. In this case, the floating address will be assigned the value of the absolute address indicated. Any number of tabs or carriage returns may intervene between a floating address assignment and the contents of the register to which the floating address is assigned.

6. Instructions: typed as three lower case letters followed by an absolute or floating address and terminated with a tab or carriage return.

7. Numbers: typed as a plus or minus sign followed by 1 to 9 decimal digits (the plus sign may be omitted if desired). The number should contain no commas or decimal points and should be terminated by a tab or carriage return.

8. End of tape: the end of a program is marked by the word "start". The word "start" is preceded by the address at which the programmer wants the program to start, followed by a vertical bar, e.g.,

127| start

One precaution must be observed. If the same piece of tape is to contain two or more programs or one program with two or more "start" indications, there must be no character (including tab, space and carriage return) between the final "t" of a "start" and the "f" of the "f2s" which must follow.

9. Typographical errors: errors which are caught soon enough may be deleted on tape by repositioning the tape and punching all seven holes of the erroneous character by means of the "code delete" lever, yielding the character known as "nullify", which is completely ignored by the computer during input. If an error is not caught soon enough, the tape must ordinarily be reduplicated up to the error, the correction made, the erroneous character skipped, and the rest of the tape duplicated. Simple changes can sometimes be made by manual nullification of characters in the middle of the tape and/or by adding words at the end of the tape, preceded by absolute address assignments which cause the new words to be read in over the incorrect words, replacing them. The incorrect word, however, must be a legitimate one which the computer can interpret; otherwise the computer will stop on the illegal word.

10. Ignored and synonomous characters: the space and back space are completely ignored by the computer, as is the nullify. Spaces may be used for typographical reasons wherever desired, but neither back spaces or spaces can be used to make corrections or to take the place of tabs. Carriage returns and tabs are interpreted identically, and have the

logical funttion of terminating a word. Extra carriage returns or tabs may be used at will between words or addresses, but not within them. Comma, period, plus, minus, equals, vertical bar, letters and digits all have certain meanings and must not be used indiscriminately. The digit zero and the letter o are always completely interchangeable as are the digit one and the letter l. Upper case should never be used, except in the title line. Both upper and lower case shift keys punch characters on tape which are not ignored, but as long as all of the typed characters come out in lower case it does not matter if any shift characters are put on the tape accidentally. 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 probably valid, providing spaces (or back spaces) have not been used to give the effect of a tab (or of the absence of one).

11. Layout: rules 1 through 8, above, specify the structure of individual words, address assignments, and the beginning and end of the tape. Rule 10, making carriage returns and tabs synonymous and permitting more than one of them wherever there is one, permits great latitude in page layout. Ordinarily, several words are typed in 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. When using absolute addresses, it is good practice to put an address at the beginning of each new line to help during typing and proofreading (since it helps prevent losing one's place), and to put a blank line between non-consecutive registers.

(The following example computes two totals of 5 numbers each and verifies the result by checking the sum of the totals against a cumulative total of all ten numbers (crossfooting). The program is perfectly general but the format of the printed page could be altered by changing the print program at a3-2.)

```
f2s 198-5-2 Eisenhower - Crossfooting 8/23/54

11|
add50c          rst2a           rst10c          a1,rst5b        a2,ccf60a
inc1c           cci60a          ccf62           add50c          cci62
jiia1a          jiia2b          cci60a          add63           cci63
tyc51           sub62           jiza4           stp0            a4,rst2a
stp0            a3,ccf60a       tyn19           tyc51           jiia3a

50| 8125331     7631900         10093726        53110221        9872865
55| 15694230    8513960         7010123         4001523         25834821
60| 0           0               0               0

11| start
```

Locating Mistakes in Programs

When a computer program is performed several alternatives may result. The computer may

1. produce correct results and stop as planned;

2. produce results of unknown significance and stop as planned;

3. produce incorrect results or none at all and stop as planned;

4. produce incorrect results or none at all and not stop;

5. produce some kind of results or none at all, and stop because of a violation of the computer's rules after recording a post-mortem.

In cases 2 and 3, the burden of checking usually lies on the programmer, but in cases 4 and 5 the computer can aid in the trouble-shooting process by printing out useful information concerning the state of the contents of various registers at the time the machine stopped, and a little of the "history" of the way the program operated before the stop. This recorded information and the process of recording it is called a "post-mortem." In case 4, the computer must be stopped by the operator, who then obtains a post-mortem, but in case 5 the post-mortem is performed automatically. In cases 2 and 3 a post-mortem can be obtained by request of the programmer.

The results of the computation, whether correct or incorrect, are often very useful in mistake location. This is especially true in cases 2 and 3 when a hand computed check may be an effective way of detecting an error. In any event the results of the program and speed with which they were obtained should be carefully considered.

Computation Post-mortem

A typical SAC post-mortem is given below and the meaning and significance of each type of information discussed.

(see next page)

198-5-2 Eisenhower - Crossfooting 8/23/54

Computer ran for 2 min. 11.57 sec.

Tape 1 is at block 41

Tape 2 is at block 52

Tape 3 is at block 99

Tape 4 is at block 35

STOPPED AT a2+1 a2+1| ccia3+30 a3+30| 124053879

AC| 149888700 RR| 45

COUNTERS a| 1,2 b| 4,5 c| 9,19 d| 0,0 e| 0,0 f| 0,0 g| 0,0

START..a2+6 $(a2..a2+6)^3$ a2..a2+10

a1..a2+6 $(a2..a2+6)^3$ a2..a2+2 stop

a3+29| 88834043 124053879 88834043 124053879

The post-mortem information in the example can be interpreted as follows:

*Line 1: The title line, reprinted for identification.

*Line 2: The computation time to the nearest hundredth of a second from the time the program started to the time it stopped. This time will be the time required for the hypothetical computer SAC to perform the program and may not agree with the actual elapsed time on Whirlwind.

*Lines 1 and 2 will be automatically recorded before any computer stop, even if no post-mortem is to be performed.

Lines 3, 4, 5, 6: If any tape unit has been used, the block at which each unit is positioned will be recorded, with the number of the last unit used typed in red.

Line 7: The computer refused to perform the instruction at a2+1 which was, according to the post-mortem, cci a3+30 where a3+30 contained the number 124053879.

Line 8: The contents of the accumulator and remainder register at the time the computer stopped were 149888700 and 45 respectively. If dhr had not been used, the contents of the remainder register would not have appeared in the post-mortem.

Line 9: The index and the criterion associated with each counter are printed on this line, with the index printed before the comma and the criterion printed after the comma.

Lines 10, 11: This "jump table" illustrates the last ten distinct sequences of instructions performed before the computer stopped. In this example the computer had not gone through ten distinct sequences of instructions, so the "history" of the program from the start is given. In this example the computer performed the sequence of instructions from the "start" to a2+6. At that register a jump to a2 occurred, starting a new sequence at a2. The jump at a2+6 occurred 3 more times causing the sequence of instructions from a2 to a2+6 to be repeated 3 times. This is indicated on the post-mortem by enclosing the sequence a2..a2+6 in parentheses and using the exponent 3. Note that the jump in a2+6 was performed 4 times with the result that the instructions a2...a2+6 were performed 5 times. After proceeding to a2+10 the cycle was apparently repeated until the computer stopped on the fifth repetition of the instruction in a2+2.

Line 12: The final contents of every register whose contents have been altered in the course of the program is printed here.

In this example the trouble is located by observing that the contents of the accumulator are larger than 2^{27} = 134217728 which is the largest integer that can be placed in a register. The machine therefore recognized that the cci instruction was attempting the impossible and stopped.

The above post-mortem might be the result of the program on page 11-3, except for the fact that lines 3, 4, 5, and 6 and RR 45 would not have been present since no magnetic tape instruction or dhr is present in that program. However, the numbers being added by that program would result in the error indicated above.

In general, to locate a mistake which the printed or plotted results do not make obvious, one examines the post-mortem to determine where and thence why, symptomatically, the computer balked. Since the possible sources of P-M for each instruction are specified in the code, and since the instruction on which the P-M occurred is clearly indicated on the P-M along with the numerical quantities involved, this is never

difficult. If the source of the trouble is not then obvious, one tries to establish how the situation arose, by tracing the path of the program back from the "stop" by means of the "jump table", and by examining the contents of critical storage registers.

When no obvious causes are noted, a wise procedure is to test, carefully, each and every piece of available information for consistency, to make sure that it agrees with what you expected of the program. This involves mentally confirming the results printed or plotted, the exact value or at least the order of magnitude of the contents of every register listed in the P-M, and the validity of every jump in the jump table. If everything jibes and still no explanation of the source of the mistake can be found, the possibility of a computer malfunction should be considered and, in some cases, the program should be re-run to make certain that exactly the same symptoms are obtained. If so, and if no able counsel can be found, relax; then try the above procedure over again.

Conversion Post-mortems

Some programming errors can be detected before computation begins; for example, an integer with magnitude larger than 2^{27}, an instruction with address section larger than 298 or an illegal combination of letters as an operation code. For errors of this type the computer will print a description of the error before the program is run. For example, the computer may print

Integer magnitude too large at a1-5

Improper instruction used at b6-3

Undefined floating address used at h7

Counter letter missing at c106-5

Correction of Mistakes

After a tape has been prepared, corrections can always be made by duplicating the tape up to the mistake, typing the correction, advancing the tape reader beyond the mistake and finishing the duplication. Sometimes the correction can more readily be made by adding to the end of the tape. If the word in register 23 is ccf231 and should be cnf 123, adding 23 cnf123 to the end of the tape (before the "start") will correct the mistake. If register 23 contains cni 123 (a non-existent operation), it must be corrected where it appears as it will otherwise be treated during input as an improper operation code regardless of what is put into register 23 later. Frequently the mistake can be judiciously nullified (by punching holes manually in the tape) without disturbing the sequence, but completely nullifying one word will ordinarily result in

the following word going into the wrong register. In most cases it will be possible to nullify everything but a decimal digit or two (and the tab or carriage return needed to terminate it), which will convert without trouble to some positive integer which can be replaced by a correction tacked on the end of the tape. For example, if 23 contains cnil23 and should contain cnfl23, the c, n, and i can be nullified, leaving the integer 123 in register 23, and then 23|cnfl23 can be tacked on to the end of the tape, correcting the mistake.

Corrections of this type can also be made using floating address notation. For example, in the sequence

```
al, add x2
    cci x3
    cni b7
```

the cni b7 can be corrected by nullifying the c, n, and i as described above, and placing

```
al+2|cnf b7
```

at the end of the program.

If several registers must be inserted in a program, the use of floating addresses makes it feasible to duplicate the program tape up to the point of insertion, type the registers to be inserted, and continue duplicating the tape. If fixed addresses have been used, and it is desirable to avoid renumbering all the instructions after the insertion, the program must be "patched." For example, if the sequence

```
add 23
mby 24
```

must be inserted between dby 32 and cci 33 in the segment

```
100|ccf 31
   dby 32
   cci 33,
```

then either the dby 32 or cci 33 can be replaced by a jump instruction which jumps to a hitherto unused part of storage where the necessary insertion can be performed followed by a jump back to the original program. Assuming that the registers 200 through 203 have not been used by the original program, the insertion mentioned above can be accomplished by placing the sequence

```
101|jmp 200

200|dby 32
   add 23
   mby 24
   jmp 102
```

at the end (but before the start) of the original. It is good practice

when typing the original tape to leave blank tape between the last program register and the start indication, to facilitate corrections.

Summary

The process of preparing a coded program for any digital computer consists of planning, then coding. The planning is usually difficult but unavoidable, regardless of what kind of a computer is to be used. The coding is, in principle, trivial. In practice, the details of the conventions and the many possible sources of misunderstandings and careless mistakes make the process a rather lengthy one. The amount of learning required can be reduced by use of simple, mnemonic conventions and by making the computer do as much of the clerical work as possible. SAC incorporates many mnemonic features and simplifications. It also incorporates powerful means to help locate mistakes which do occur.

SAC is reasonably typical of digital computers generally, but incorporates many features (some of which have not been described in these notes) which make it one of the easiest digital computers for which to do the detailed, trivial, burdensome, but frequently fascinating job of coding.

12. NUMBER SYSTEMS

It is, perhaps, not always generally realised how far we depend upon convention to interpret the things we see written down. Suppose I write down the symbols 1954. You will probably conclude, unless something to the contrary is said, that I am writing down the year of Our Lord (according to the Gregorian Calendar). If I put a comma after the one - 1,1954 - you might well be right in thinking that I am about to refer to one thousand nine hundred and fifty four (= 1 x 1000 + 9 x 100 + 5 x 10 + 4) jiggle nuts. A variety of possibilities might occur to you - for instance, nineteen point five four (= 1 x 10 + 9 + 5 x 1/10 + 4 x 1/100), $19.54, 1954 hours (using a 24 hour clock, D.S.T., E.S.T.) - the choice would depend on the context. If I write 1.954, however, it is likely that the only suitable convention that will occur to you is the so-called decimal notation, and you will suppose that I mean 1 x 10 + 9 x 1/10 + 5 x 1/100 + 4 x 1/1000, where by 1/10 I mean 1 ÷ 10, and by ÷, +, and x I mean that the set of symbols are combined according to a certain set of rules to form a new set of symbols. (By the rather obscure wording of the last sentence you will see that we are near the edge of quicksand and in fact readily become bogged in defining our symbolism, but I shall assume that the normal meanings of the symbols such as ±, x, ÷ and so on are known to you).

In the few examples above I have tried to show that the meaning you attach to a set of symbols is largely dependent upon the conventions that are commonly adopted. Now it is obvious that the ones that <u>have</u> been adopted are not by any means the only ones that <u>might</u> be adopted - it is merely a matter of convenience to use the particular ones chosen. This is an important point to realise, since what is convenient as a convention for you and I is not necessarily so for a piece of electronic apparatus, as we shall see, and there is no good reason why we should not adopt different conventions for use <u>inside</u> such an apparatus. The only difficulty introduced by this is that a conversion must take place at some stage from our conventions to those of the machine and vice-versa. This, however, can often be carried out by the machine itself, thus <u>not</u> affecting our reading of the results and feeding in of data, which is done in a familiar convention.

Because the most usual convention about numbers is the decimal one, there is a tendency to think of all numbers in this way, but that is not in fact how we use them. For instance, let us write down the time - about 0910, say. This is really two separate numbers 09 and 10 each expressed in decimal notation and written together for convenience. Now increase the time by 59 minutes. We do <u>not</u> write 0969, which would be a simple, logical thing to do if we were really using a decimal convention, but we <u>do</u> write 1009, since there are exactly 60 minutes in 1 hour. It would, in fact, be more logical not to write each number in decimal form, but to invent additional symbols to correspond to 10, 11, 12, ... , 59 minutes. Let us suppose 10 → A, 11 → B ... 35 → Z, 36 → a, ... , 59 → x; we now write the

time 1009 as A9, and 0936 as 9a. Of course the first symbol could only go up to C, or, on the 24 hour system, to N. This is just the sort of system we use when we define the unit of time called a month, for here we use a combination of letters in place of symbols to specify the numbers 1, 2, ... , 12, i.e. January, February, ... , December. Whatever the representation of numbers employed commonly, we can, of course, always fall back on the decimal notation, but this is not the logical notation for a system if more than 10 symbols would arise naturally in counting. The number of symbols arising naturally in designating time is strictly 60 for seconds and minutes, and 24 for hours; these numbers are called the "bases" of the system, and we speak of counting "to the base 60". To the base 60 the number 95 = 9 x 60 + 5 = 545 in decimal notation. We employ a great many different bases in practice, as a few examples will readily show: - (i) measures of distance: 12 inches = 1 ft. (base 12), 3 ft. = 1 yd. (3), 220 yds. = 1 furlong (220), 8 furlongs = 1 mile (8); (ii) angular measure: 60 seconds (of arc) = 1 minute, 60 minutes = 1 degree, 90 degrees = 1 right angle; and (iii) fluid measure: 2 pints = 1 quart, 4 quarts = 1 gallon. It is easy to think up other examples for yourselves.

We have already seen that the decimal system for numbers employs all the symbols 0 to 9 and systems with larger bases can employ more symbols. What happens if a base less than 10 is used? Obviously we shall need less, rather than more symbols and it is convenient to employ the appropriate decimal symbols. Thus for base 4 we might use 0, 1, 2 and 3 only. 5, 6, 7 and 8 only would be permissible but less easy to understand, and so would probably not be used. The simplest possible base is 2, which uses the symbols 0 or 1 only; this offers a method of recording a choice between two possibilities in each digit position. In this notation the number 11, of course, is no longer 1 x 10 + 1, but is 1 x 2 + 1 = 3 (decimal notation). We may readily set up conversion from binary to decimal if we wish, by the rule that, if abc...is a binary number of (t + 1) digits, then the decimal equivalent is found by $a \times 2^t + b \times 2^{t-1} + \ldots + u$. Conversely we may convert from decimal by writing the number as the sum of powers of two and writing coefficients down as required: thus 25 = 16 + 8 + 1 $= 2^4 + 2^3 + 1 \rightarrow 11001$ in binary.

coeff. $2^4\ 2^3\ 2^2\ 2$

Using the same form of positional notation as in decimal, but with the base 2 substituted, we may express numbers which are not integers, e.g. $19.50 \rightarrow 10011.1$, since $0.5 = 2^{-1}$ (or 1/2). Engineeringwise it is more convenient as a rule to employ binary notation within the machine. This enables one to use such physical properties as an electric current being switched on or off, soft iron being magnetized or not, or electronic switches being set in one of two positions to represent numbers rather than relying for them on a current having one of several possible values within fairly narrow tolerance limits. In this way it is possible to ensure reasonably high reliability of operation.

Arithmetic with binary numbers goes much like that with decimal numbers, but tends to look a little peculiar when you aren't used to it. Let us consider a sum in decimal - 7 x 8 + 15 x 3 = 56 + 45 = 101. In binary this would be written as 111 x 1000 + 1111 x 11 = 111000 + 101101 = 1100101. We note that 1 + 1 = 10, and 1 x 1 = 1, 1 x 0 = 0 x 1 = 0. As in decimal notation, if two numbers of a and b digits respectively are multiplied together the result contains a + b - 1 digits altogether. An important difference is that adding a zero to the right of an integer, which multiplies a decimal number by 10, has the effect of multiplying a binary number by 2. Similarly moving the binary point multiplies or divides a number by 2; thus 1101.11 = 1/2 x 11011.1 = 1/4 x 110111 = 2 x 110.111 = 4 x 11.0111 = 8 x 1.10111.

Most machines are so constructed that only a fixed number of digits can be dealt with arithmetically at one time, these having a correspondence, usually, with the number of digits contained in a single unit or location in the store. All machines are such that only a finite number of digits can be accommodated altogether. As a consequence of the first restriction certain difficulties are bound to arise in dealing with numbers of very different sizes. For instance, if our machine is designed to deal with four-digit numbers only (in decimal), then we cannot readily add 1954 to .0003 and retain full accuracy without some special arrangements. As a consequence of the second restriction we cannot store certain numbers at all - the number denoted by the symbol π (= 3.14159 etc.) for instance. However, for most practical purposes it is sufficient to work to 10 or 11 significant figures, and this is normally provided against. It is, of course, necessary to have at least one register in the machine capable of storing $2k - 1$ digits (where the numbers stored are of k digits) in order to accommodate the result of multiplying two numbers, unless these can always be accepted rounded off in the least significant place. Since the numbers stored are of fixed length as a rule we must introduce a scaling operation to accommodate two numbers such as 1954 and .0003. For addition purposes, so long as the two numbers to be added are adjusted so that they are correct relative one to another, the result will be correct, irrespective of the supposed position of the decimal point. Thus if we can only store 19540000 and 3 the result of adding these, 19540003, can be interpreted as 1954.0003 if we imagine that we did store 1954.0000 and .0003. The same number is held in the machine whether we call it 19540000 or 1954.0000. However, the position of the point is vital in multiplication or division. Consider 3 x 3 = 9. In a 4-digit machine holding only integers, the result will appear as 0000009 in the register, whereas if we interpret the original numbers as .0003 each, the result should be .00000009. We can overcome many of the difficulties of this nature either by scaling the numbers adequately (i.e. multiplying by appropriate factors before storing or before multiplication) or by some machine or programming device involving special representation of the numbers. We often use a device of this kind ourselves when writing, for instance, fractions. This is convenient for representing, for instance, 1/3, which has no exact value in decimal notation. This is an example of the use of two integers to define a single decimal number. Another form of representation

is to write 1954 = 0.1954 x 10^4, and .0003 = .3 x 10^{-3}, and record in each case two numbers. These are 0.1954 and 4 for the first and .3 and -3 for the second. The rule is to write the first number as between -1 and 1 (excluding the letter) together with the power of 10 required to give its correct scale (10 is here called the "radix" of the representation). A similar convention can be adopted for any radix, e.g. for radix 2 we write the first number as between 1/2 and 1 and the second as an appropriate power of 2; for instance, .1954 = .7816 x 2^{-2} and we write (.7816, -2) as the representation. This is convenient for preserving the maximum accuracy in a calculation, but is apt to be slow during addition of numbers, since the numbers have to be adjusted in the arithmetic unit before adding them and re-adjusted afterwards. This type of representation is usually called "floating-point" representation, and is often employed in machines for preserving accuracy and avoiding carrying out scaling explicitly in programming.

Finally we come to the important subject of sign representation. The convention commonly used is to write the sign followed by the modulus (numerical value) of the number; thus +98, -5.4 and so on. This convention is used on some machines but necessitates the sign being treated as a different entity from any of the digits of a number. It is therefore more convenient to use a system of complements to represent negative numbers. This involves representing negative numbers by their complement with respect to some number, say 10 for decimal numbers, so that -00005 is written as 99995, -0.1 as 98000 - clearly -1 is not permitted. In binary it is usual to use complements with respect to 2, so that -1/2 → -0.1 → 1.1000000 for an 8 digit machine. An alternative is the so-called "1's" complement when we write -1/2 → -0.1 → 1.0111111, i.e. all zeros and ones are replaced by their opposite. All of these methods of complementing are employed in machines, and the decimal equivalent of "1's" complement (called "(9's" complement) is used in I.B.M. punched card equipment.

In conclusion, I would like to point out that it is not necessary for the programmer to be fully acquainted with the details of binary arithmetic in order to operate a binary computer. On well-designed binary machines there is no need to think of numbers as binary at all, except possibly in preparing the standard input and output routines. Once these are done the only indication that binary is being used is usually that an order multiplying and dividing readily by 2 both exists and is rapid!

14. ORGANIZING THE ATTACK ON A PROBLEM

It is difficult to draw hard and fast lines between the stages of carrying out a computation on an electronic machine, since they are necessarily closely related. A rough division was shown on one of the slides shown in the introductory lecture by Professor Adams, and it seems reasonable to distinguish five main stages: (1) preparation of the problem, (2) coding, (3) preparation of the physical input medium (e.g. cards, tape, etc.), including verification, (4) operating the computer, and (5) analysis and final presentation of the results, including any checking not carried out during the computation. In this chapter I am concerned primarily with stage (1) (preparation of the problem), but it is necessary to look at the whole picture of operation in order to understand better the necessity for certain preparations and the effects which may flow from failure to prepare the problem correctly.

Whilst it is possible to plan in a general way without considering the particular computer being used, this independence cannot be carried too far, and, in the present day machines, full efficiency can only be achieved by catering to the peculiarities of the computer used. For the purposes of this discussion, I shall not attempt to review all the possible peculiarities, but will try to point out what sort of peculiarities affect planning and in what way. The most important effects are due to the sizes of the various stores of the machine and the access time to information contained in them, and the full significance of this factor will not appear, as a rule, until coding is attempted. For this reason and because time of operation as a whole may be important and can seldom be assessed before the coding stage, coding and programming are difficult to separate out. Indeed, an eminent programmer has remarked that programming consists of writing down the operation symbols and coding putting in the addresses! However, it is quite possible to program without carrying the coding into such detail, and to have the coding carried out by someone else - with a probable consequent loss of efficiency in machine operation - and this is desirable if the overall time of carrying out the problem is thereby shortened. Such a policy would obviously be justified for problems which could be solved in a few minutes, or even a few hours of machine time, since the preparation time would largely out weigh the time of operation in calculating how long it would take to solve the problem. Equally obviously there are problems, particularly those of a day-by-day nature - inventory control, wages and the like - where machine operation must be as efficient as possible. The saving of 10 minutes per day may lead to a yearly saving of some 60 hours, perhaps, or $18,000 using an I.B.M. 701, leaving quite a margin over the consultant's fee for the efficient programmer employed, whose time might only need to be engaged for a week or so.

Let us assume that we are to separate coding from programming for the present and that we are considering taking over some part of the running of a commercial firm on a computer. Firstly, the problem for the computer must be stated. This must be done with the utmost care, and is probably the most difficult part of the process in

practical cases. I might here quote Dr. Bowden from "Faster Than Thought", for he puts the commercial aspect of this process succinctly.

> "....A typical commercial computation is probably handled by several hundred clerks. Each individual operation is perfectly straightforward and there is no mystery about the underlying principles, but because of the complexity and the ramifications of the work it may well be that no single individual understands the office procedure in detail, so that the would-be programmer may have to spend months in finding out what is in fact done...."

It must be emphasised that what is necessary is not really to find out what is at present done, but to find out all the possible exceptions to any general rules which are being applied and to specify what decisions are to be made in these cases. From the point to view of a programmer, the person who presents the problem, if he is not himself the programmer, must present <u>only</u> what data is available (all of it) and what results he requires, together with specification of what is to be done when any special situations arise. Do <u>not</u> try to tell the programmer <u>how</u> to get the results, or attempt to give him a digest of the data and facts available. This will usually detract from the efficiency of the program, unless you are capable of doing the programming yourself.

Given the data and the results required the programmer must restate the problem in a form suitable for the computer to tackle, i.e. fundamentally numerical. In particular, he must reduce any criteria for acceptance or rejection of data, or for deciding between two or more possible courses, into a comparison of the magnitudes of two or more numbers. For instance, suppose we require that an elevator should stop to pick up passengers at any floor between where it is now and where it is going, if the button of that floor is pressed. A numerical statement of this is that, if $n \rightarrow$ nth floor, the button of which is being pressed, $m \rightarrow$ the nearest floor <u>below</u> where the elevator is now, and $1 \rightarrow$ the floor to which it is going, then we require to stop at the nth floor when $m \geqslant n > 1$ or when $m < n < 1$. In any other case, including if $m = n < 1$, then we do not stop before reaching 1. This numerical expression of qualitative ideas is an essential preliminary process in all programming not concerned with direct mathematical computation.

We now have a problem suitably stated and reduced to a series of computations connected by numerically determined decisions, and can draw up a flow chart both for convenient reference and to ensure that all the possibilities have been considered and treated suitably. Next, we must consider the computations themselves and decide upon suitable numerical techniques for carrying them out. It is usually convenient to break down the individual computations somewhat further in order to make use of any library subroutines that are available, and to make the process of coding easier. On single address machines,

such as SAC, my experience is that I can conveniently code routines containing up to 100 - 150 instructions, preferably less, containing not more than three or four cycles within one another. If a computation looks like involving more than this it is preferable to break it down into two separate stages by some suitable rearrangement of the logic, if necessary. If possible, it is best to make each section complete in itself, so that it can be tested separately from the rest of the problem. This enables an economical use of the machine for error diagnosis to be made. Standard input and output routines are used to put in the section to be tested and its data, and any errors arising can be directly attributed to failure of the machine or errors in the coding, and the latter should be relatively easy to find. Although proving each section separately does not mean that they all will work correctly when put together to form the program required, the sources of error are more readily located when it is certain that they arise from misuse of the individual sections or oversights in programming, rather than coding.

This technique for aiding error diagnosis can also be regarded as a part of the process of checking required to ensure correct results from any program. Not only must the programming and coding be checked, preferably on a trial calculation or calculations designed to test all the ramifications of the problem, but also it may be necessary to provide against faulty operation of the machine. A commonly held misconception is that an effective check is provided by running the same program twice. It is true that, provided the machine does not contain a random number generator, two consecutive runs which do not agree are prima facie evidence that the machine is wrong. However, it does not follow that, if two consecutive runs agree, they are correct. At least two cases are known to me to have occured in practice: (1) the machine contained a consistent fault, and (2) a third run gave a different (and, incidentally, correct) result. The best method of checking is usually to carry out the computation by a different method and to compare the results to ensure correctness. Next best is to institute some internal checks within the calculations, verifying that certain equations hold, for instance. As an example, if we are computing the sum

$$\Sigma^2 = S_1^2 + S_2^2 + S_3^2,$$

and do the sum of cross products

$$\Sigma^{12} = S_1S_2 + S_2S_3 + S_{31},$$

we might verify that the resulting sums satisfy

$$\Sigma^2 + 2\,\Sigma^{12} = (S_1 + S_2 + S_3)^2.$$

Finally we might resort to spot checks by, say, hand calculation to verify arbitrarily chosen results. None of these systems is infallible alone or combined, of course, but the errors in the results can be reduced to a very small proportion by suitable application. Clearly it is necessary to incorporate some of the checks indicated in programming the problem and this can sometimes be done without greatly increasing the time of the computation.

I may say that I do not hold with indiscriminate use of programming techniques to overcome possible engineering failures. These not only waste time in the computation, but also lead to the concealment of faults which are best dealt with as they occur. No test program has yet been devised that ensures that a machine is fault free, and it is fair to assume that faults will occur during long consecutive runs of the machine. The longer the run without output, or, at least, recording results on permanent storage devices, the more catastrophic is a failure. It is most important to plan the reading in and out of high speed storage to more permanent forms, and from more permanent storage onto printed sheets at intervals reasonable with respect to their average failure times. Few high speed stores should be trusted for very long, on principle, though trouble free running for more than 8 consecutive hours has often been experienced on most machines. A good average maximum time to assume is an hour, as not too much is lost if failure occurs; personally I prefer to use 20 minutes as a maximum period - this is more than enough on a fast machine and adequate on slow ones. Enough should be recorded after this time to enable a restart to be made from that point, and this should not be destroyed at the end of the next period until some verification has been carried out (this may just consist in not stopping the machine, if it appears error free, but this should only be judged by an experienced operator). Such intermediate results may, of course, be adequately stored on mediums such as magnetic tape or a magnetic drum for the whole period of an average calculation, and may or may not be printed out, as required. However, it may well be that some intermediate results are required by the problem setter, or are desirable for checking purposes, and these may be tied up with the break down indicated above.

This brings us to consideration of another important point: how shall the data best be recorded for the machine, and how are the results to be presented? Of course the form of the results must be determined primarily by the needs of the problem setter, but he should be encouraged to demand them in the most convenient form for use by those whom they concern. It is usually possible to arrange the setting out suitably by the machine, thus saving much time in rearrangement. In the same way data should be recorded in the order and format most convenient for the recording agent, subject to a preference being given to recording directly onto the medium used by the machine for input - this should not however be allowed to complicate the process of recording. Whilst in each case this throws some extra work on the programmer, who must arrange for the machine to sort out the data into the form required for operation of the program, my own experience in handling scientific data would lead me to believe this to be a good thing. Convenience for the recorder means less mistakes in the original entries, and such mistakes cannot usually be easily detected or rectified - it has been well said that the most important entries in a ledger are the original entries, the accuracy of which cannot be checked by the most highly paid accountants, and that these are usually made by the lowest paid member of the staff!

Let us now review the points which we have to bear in mind when programming. First, we must state the problem carefully in a form

fundamentally numerical; secondly we must decide the form of our data and of our results; thirdly, we must break down the problem into convenient units for (1) coding, (2) storage, and (3) time of operation; fourthly, we must decide the numerical methods to be employed; fifthly, we must decide on the checking techniques to be used. After all this we may begin to carry out the coding, first selecting any library subroutines which may be available for carrying out units of the computation, and secondly, writing any new routines required, finishing up with the master routine tying all the bits together. At this stage we should estimate the time required to carry out the sections of the computation and reconsider our breaking down of the problem to see if this requires revision. Having satisfied ourselves on all these points we are ready to proceed to the testing of individual routines, and finally to complete testing of the program. Your own experiences with TAC will show that the amount of time taken by the latter processes are by no means negligible matters! There are few things that require more patience and perserverance than getting a program right. However, once the program is tested and found correct, there is no greater satisfaction than to watch the machine producing results in the knowledge that each one represents many man hours of effort saved - not to mention a few dollars!

15. ORGANIZATIONAL PROBLEMS

The purpose of this chapter is to discuss some of the problems that are encountered in organizing a computing center. First we will consider certain basic distinguishing characteristics and then we will go on to the more common problems that are found at most centers.

It is difficult to give a general procedure for organizing a computing center since each center represents or is part of a system having its own characteristics. Of these characteristics, the following four seem quite basic.

1. Size of the computer. In the surveys already offered in this course you have had an opportunity to see the wide variety of computers being offered on the market. Last year Dr. Wilkes of the University of Cambridge suggested that in organizing a center a guiding principle to remember is that machine time is valuable. This is especially true when dealing with the larger more expensive machines. Considerable effort should therefore be spent in insuring that as much machine time as possible is used productively. However it is also true that any routine can be shortened if enough time (both coding and machine testing) is spent on it. Consequently a compromise must be reached.

Associated with the size of the computer is the amount and speed of the auxiliary storage equipment available. A center having a central computer whose operating speed is comparable to the input-output speeds can make more extensive use of the auxiliary equipment without sacrificing too much in efficiency. Such considerations are usually predominant in deciding what sort of automatic coding the center will adopt.

2. Type of coder. In many organizations the person who proposes the problem does the coding. This kind of operation is often referred to as "open shop". On the other hand, the coding may be done by a resident group of experienced coders in what might be called a "closed shop". In the open shop, provision must be made for the training of a large number of new programmers, for their supervision, and for the carrying-over of results from one problem to another (for example, by the use of a library of subroutines). Closed shop operation creates the problem of communication between the person proposing the problem and the coder. Closed shop operation can also lead to personnel problems since coding someone else's problem can become very tedious. This leads to efforts to reduce the coding to a sufficiently low level so that it can be done by relatively unskilled personnel or by the machine itself.

3. Type of problem. Here we might distinguish between the production-type problem and the short-run problem which is completed after a few hours or less of computation. Once a production-type problem has been successfully coded, the resulting routines will be run again and again with little strain on the programming staff. For this type of problem, of course, any effort that makes the routines more efficient will pay off in rich dividends. Also the scheduling of machine time to handle production runs is greatly simplified since the time required for a run is usually well known. For short-run problems the ratio of production time to checking time will not be very high. For these problems, methods that reduce the time required to detect and remove mistakes from a routine may be extremely helpful. Special routines for assembling a final program from a set of subroutines, for carrying out floating-point, matrix, or complex arithmetic, and for providing extensive post-mortem information are often used for this purpose.

4. Kind of computing center. This may best be described by examples. A computing center that is renting out machine time to a different division of the same company or to a different company may not be too concerned about how efficiently the machine time is used. On the other hand a center whose income or support is based upon productive results will of course make every effort to increase production. A research center may devote a great deal of time to computing certain constants to many decimal places or to exploring theorems in Number Theory. However, a center that has to justify its existence to a cost-conscious management may find itself setting up routines to take over pay-roll calculations, etc.

It should be pointed out that the problems associated with the characteristics just described are not mutually independent. For example, the type of coder one employs is related to the type of problems to be solved. To keep a large machine busy with a set of short-run but complex problems requires a large staff of programmers. In such a case it would be more practical to let the person who proposed the problem learn and carry out the coding. Also the kind of problem to be solved is related to the kind of computing center. For example, one would not expect that a research center, such as one finds at the University of Cambridge or at M.I.T., would allow one programmer to monopolize a large fraction of machine time over an extended period. On the other hand, General Electric can assign a large fraction of its IBM 701 time to the design of steam turbines for its Lynn plant.

So far we have been considering the organizational problems of computing centers having certain characteristics. One of the more interesting problems is to select the characteristics for a center that is about to be set up. The questions of which computer to buy (or even whether to buy one), which problems to put on the machine, and whether to train present personnel to code can be difficult ones. The main difficulty is that there usually is no one simple answer. For example, in the choice of a machine, some people feel that it is better to own two smaller computers than one large unit so that if one machine is down

the other is still available. On the other hand, the speed and flexibility of a large machine is far more than double that of a machine of half its price. Consequently the cost per operation and the number of operations performed between down periods may make it the better buy if there is enough computation to keep it busy. (Of course, each computer has its own basic properties.)

It might be pointed out that the selection of problems to be placed on a machine or the investigation of ideas for the development of new control systems can be carried out on time rented at one of the many available computing centers. Some companies have found it useful to purchase a smaller computer to gain experience to indicate what future steps should be taken.

In the preceding paragraphs we have emphasized the differences between computing centers. However, all of these centers do have a great deal in common - they all want to make use of a high-speed computer to solve a problem. The fact that this course is being offered to you indicates that there is a good deal of common ground. You have already gone over the steps needed to solve a problem on a high-speed computer (in Prof. Adams's introductory lecture, in the movie "Making Electrons Count," in solving your problem on TAC, and, in lecture 14 given by Professor Douglas). However, in each of these cases the approach was from the programmer's point of view. I would now like to review these steps once more, but this time from the point of view of the person who is organizing the center. There will, of course, be some overlap in the two points of view but there are enough new ideas, I believe, to make this review worthwhile.

The steps indicated on page 1-9 are common to most computing centers although the importance of each step may vary widely. For example, in a business application where a procedure for inventory control has been developed and put into regular use, the main steps of interest would involve preparing a tape or cards with the input information, running the machine, and checking the results. The other steps might enter infrequently when variations or improvements are introduced into the system. Let us now consider the steps in some detail.

1. Proposing. Sometimes the main problem here is to get people to propose problems. It can take an awful lot of problems to keep a large machine busy. Some people feel that a problem has to be very complex before it should be coded for a machine. On the other hand, there are many problems that could be solved quicker calendarwise on a desk machine. Also many people tend to ask for far more results than they need or can ever hope to process. As Prof. Douglas has pointed out, many problems are stated in a misleading form.

2. Planning. This involves not only the selection of a procedure (which may or may not be numerical) but, in many cases, the selection of the computer to be used. It is interesting to note that by the techniques used to simulate TAC and SAC on WWI, many centers have been transformed from single computer units to multi-machine projects. Each of

these computers, whether real or simulated, has its own advantages and disadvantages. The factors involved in choosing among them include: availability of the machine, ease of coding (as measured by the time it takes a programmer, who may be untrained, to code his problem), available storage, computing speed, computer reliability (for a simulated computer, this will depend upon the degree of testing), ease of error detection and tape correction, available precision, and available subroutine library.

3. Coding. In an open shop center the coding of a problem can be greatly simplified by the use of such techniques as floating-point representation, symbolic and relative addresses, and counting facilities. Moreover, the use of mnemonic instruction codes, compiling routines, subroutine libraries, etc. abbreviates the training period of a new programmer. Of course, the availability of more than one computer (real or simulated) does increase the number of conventions that a programmer may have to learn. Also the slowing down of the machine by interpretive routines must be taken into consideration. Such techniques can also be used, of course, in a closed shop operation but here the need of them is greatly reduced. For production-type problems it will, in general, be preferable to develop routines that are as efficient as possible.

4. Clerical. In this category we can include typing or punching, verifying, and the filing of input and output information. When possible the use of routines that assemble previously prepared subroutines and allow the use of special pseudo-codes (such as tyn for calling in a special output routine) simplifies the typing by reducing the length of tape needed and by making possible the use of more common terminology (e.g., start). Also versatile read-in programs that can ignore certain characters (such as color shifts, spaces, treat the letter l and the number one as being synonymous, etc.) make it possible to check the correctness of a tape by the visual inspection of the typed copy produced while the tape was being punched.

Checks like verifying and proofreading are very desirable since they can be done at a cost far less than that of the machine time wasted because of undetected mistakes. Mistakes that result because of the illegibility of a programmer's writing can only be detected by the programmer himself. The use of a read-in program that can detect typing mistakes will also save a great deal of computing time.

The filing of tapes and results can, as in so many other cases, be facilitated by a suitable numbering system. The handling of machine output is simplified if these results are suitably labeled. In the Whirlwind computer this feature has been made automatic.

5. Testing and debugging (isolating and removing mistakes). In a computing center that must deal with a large number of new problems, the task of debugging the corresponding routines can be very time consuming. SAC and TAC have illustrated how useful conversion and computation postmortems can be. However, it should be noted that the incorporation of mistake detection routines can materially slow down the computation. Moreover printing out too much information each time a mistake is detected.

can also lead to a waste of computer time.

In many centers programmers are allowed and even encouraged to operate their own routines. This has obvious advantages but has often proved objectionable because of the tendency of programmers to try to correct their routines on the spot without carefully considering what the changes should be. Thus machine time can be wasted both while the programmer in question is deciding what changes to make and also because the hasty changes he makes may be in error.

6. Solving. The distribution of the machine time among the accepted problems will, of course, depend upon the particular system involved. This system will specify who gets priorities, etc. At the Digital Computer Lab. we have found it worthwhile to distinguish between long runs and short runs (less than five minutes) where short runs usually get the priority. At IBM in New York City a schedule has been set up so that time assignments can be made automatically by computations from a rigid formula.

The system of scheduling adopted should be one that does not encourage programmers to overestimate their required machine time. This is particularly liable to happen if the system is such that the programmer will have great difficulty in regaining access to the machine again. Hence a fluid system is desirable. Of course a programmer who is paying at a substantial rate for the machine time he uses tends to be at least moderate in his time requests.

Keeping a record of the actual time used by a given problem is a clerical job that can be done by the use of time clocks. However, a very suitable system results if the machine itself has some way of automatically recording the time. For example the Whirlwind I computer contains a timing register that can count up to almost 10 hours in steps of 1 1/15 seconds. Thus by setting this register initially to zero it is possible to punch out the tape number and time for each problem that is read into the computer. If, for some reason such as machine breakdown, the machine run is interrupted, this fact can be manually punched on the time record or log with an indication of the time lost. This log can later be processed by the machine itself to produce weekly or monthly statements of time used, etd.

A type of interruption that can prove most frustrating to a programmer is one that is caused by a transient machine malfunction occurring after the routine has run a considerable length of time. A common type of such malfunction is the loss of a digit either in the memory itself or during the transfer of information. Such a loss is usually detected by the computer through a parity check or transfer check. A parity check adds up the digits in a word and adjusts an extra digit tagged to the word so that the sum is even. A transfer check simply uses two different paths of transmission and checks the end results. Another useful check is a sum check where all the words in a block of information are numerically added, reduced modulo some convenient number, and the sum stored. Whenever this block is transferred, the sum is checked. Whenever the machine detects a malfunction

by one of these checks it will stop. In many types of calculations the only way the routine can be restored to its correct state is by completely rerunning the problem. This can result in a serious loss of computer time.

To reduce the amount of machine time lost as just described, the programmer can set up rerun points in his routine by periodically storing on some form of auxiliary storage all the information necessary to reconstitute his routine. At the time of a machine failure a rerun or rollback routine can then be employed to continue the calculation from the last rerun point. It should be noted here that in many cases the auxiliary equipment itself must also be reconstituted.

Finally care should be taken to avoid idle machine time in the interim when one problem has been completed and a new one is to begin.

7. Analyzing Results. Some checks must always be maintained on the numbers being obtained from the computer. This is basically a problem for the programmer and Prof. Douglas discussed various checking methods yesterday. It might be mentioned that it often pays in the running of a computing center to provide a source of questioning for the programmer to be sure that he is aware of this checking. This sort of questioning usually comes in the preparation (or screening) stage when the programmer is asked to indicate how he will be able to verify his results.

8. Maintenance. The engineers who maintain the machine are usually in attendance or on call while the machine is being operated. The manufacturer usually sets up some sort of maintenance schedule. Some difficulties that arise during operation are readily recognized by the machine operators. It is a good practice for the machine operators to refer any suspected difficulties to the engineers. Log books are usually kept so that the operators can describe any unusual machine behavior. Most programmers will not try to push runs on a machine that is misbehaving, since it is sometimes sufficiently difficult to trouble shoot a routine on a well machine. The organization of the center should not encourage programmers by making them lose their turn to try to get results anyway when there is evidence of machine malfunction.

Standby terminal equipment should be provided when possible since the cost of such equipment is relatively low. Thus the giving up of machine time because of a malfunctioning typewriter can be avoided.

Special routines have proved very useful in the testing of computers. The extent to which such routines can be used will depend, of course, upon the ingenuity of the programmers and the nature of the computer. At the Digital Computer Laboratory two sets of routines have come into use. The first is used with marginal checking for the routine testing of the various computer sections. The second set is used for diagnostic purposes to locate actual failures in the auxiliary drum system and terminal equipment. In the future it is planned to combine some of the features of both sets of routines.

16. TECHNIQUES OF AUTOMATIC CODING

Choice of Coding System

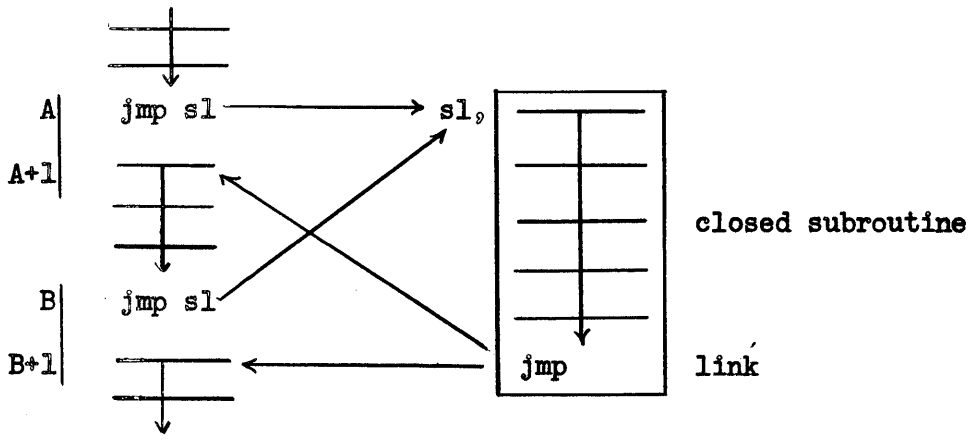
When planning to use a large electronic digital computer, we are free to choose, from a very large range of possibilities, the form of coding in which programs shall be written, with little or no effect on the amount of hardware required. The reason for this is that a large computer can, as it stands, be made to handle a very wide range of information-processing tasks; one such task is the receipt of the machine's own programs and the conversion of these programs into operations to be carried out. Having decided on our code, we merely have to provide the machine with a program that will tell it how to convert this code into the form which it requires. There are of course limitations, and these we shall consider later.

Our aim in choosing a code is to reduce the amount of labor involved in programming. This is particularly important at the present stage of automatic computing, when big new projects are being initiated daily, but our educational institutions are not yet generally equipped to produce programmers. It is therefore desirable to relieve the programmer of unnecessary chores, and, in particular, to see that no work is repeated.

Subroutines

The first and most obvious development is to keep a record of all pieces of program that are likely to be of use to several people; the so-called library of subroutines. A written description of each subroutine enables a programmer to see how it might fit into his program; he denotes it on the program sheet by its catalog number. At first subroutines were incorporated into programs by copying them in the form of punched tape or cards; the advent of large auxiliary stores has made it possible to keep a library in the auxiliary store and so to make the incorporation of a subroutine into a program fully automatic.

Fitting a subroutine into a program is by no means a simple business, and a whole host of conventions grows up around the use of a library. A difficulty soon arose over subroutines that were required to be used at two or more different points in a program: should such subroutines be copied afresh at each point, or should one copy be made to serve, thus saving storage space but raising a problem in ensuring the correct sequence of execution of a program? The latter course led to a special type of subroutine known as a "closed" subroutine, which is allocated a place in the store away from the main part of the program. A jump instruction sends control to the subroutine when required, and the subroutine is so arranged that control always jumps back, when the subroutine has been executed, to the instruction following the one which caused the original jump. (See figure on next page.)



The jump instruction at the end of the closed subroutine, known as the link instruction, must contain an address depending on the address from which the subroutine was last entered (in fact, it must be one more than this address, namely A+1 or B+1 as the case may be). Various methods are used in various machines; in SAC, a special instruction has been provided (sra - see SAC Summary for details) which can be used at the beginning of a closed subroutine to set the correct address in the link.

Another difficulty, encountered before the invention of symbolic addresses, arose from the fact that, since it was required to be able to place a subroutine anywhere in the store, some of the numerical addresses in its instructions would have to be adjusted according to its position in the store. This was overcome by a system of relative addresses, viz. written addresses defining registers according to their position in relation to the beginning of the particular subroutine. The relative addresses were converted to ordinary or "absolute" addresses during loading.

Several factors must be considered in making a library subroutine as useful as possible; one of these is its generality of application. For example, a subroutine might be made to carry out some operation (e.g. to locate the largest number) on a string of numbers in consecutive registers in the store. To be really useful, the same subroutine should be applicable however long the string and wherever its first member may be, although of course in any instance the subroutine must be told these particulars which are called parameters of the subroutine. Parameter values are indicated by the programmer when he uses the subroutine, and are used by the subroutine to suitably adjust its own internal working so as to produce the required result.

Often two classes of parameters are distinguished: preset parameters and program parameters. The former are indicated in the written form of the program, and the corresponding adjustment of the subroutine is made during loading. The latter are indicated by putting them into the machine as register contents, and the adjustment is done each time the subroutine is used, during the execution of the program. Commonly, they are put into registers following the jump instruction which sends control to the subroutine (assuming that the subroutine is of the closed type).

If it has been decided to place certain closed subroutines in the store, the task of writing the rest of the program becomes somewhat easier. The programmer knows that in order to cause the machine to carry out a certain complex operation he needs merely to write a jump instruction referring to one of the subroutines (and perhaps to follow it by one or more parameter values to specify the operation exactly). This is, on a somewhat grander scale, just what he does when he uses an instruction to cause the machine to carry out an operation listed in the machine's instruction code. The presence of the subroutines may be looked upon as extending the basic instruction code of the machine.

Interpretive Subroutines

This idea has been further developed and has led to the concept of an interpretive subroutine. This is used to deal with a situation in which a whole series of operations of a certain general type is required to be performed in succession, each operation requiring a small subroutine, with no ordinary basic machine operations intervening. Such might be the case, for example, if a lengthy calculation were to be performed on numbers too big to be held in single registers, so that each arithmetical step involved a set of basic machine operations, i.e., a subroutine.

In such a case, the machine would spend most of its time executing the subroutines; if control were sent back to the "main" part of the program it would immediately be referred on to another subroutine. It is possible to arrange that control, in fact, never returns to the main part of the program between operations but remains within the subroutines. These are all welded into one, an interpretive subroutine, which includes also a section to supervise the sequence in which the various operations are performed.

The jump instructions in the main program which formerly directed control to the subroutines are eliminated; all that remain are the parameters fixing details of each operation. These parameters must now specify which operations are to be performed as well as fix their details.

While an interpretive subroutine is being used, the instruction code of the machine is not merely augmented; it is entirely replaced. Instead of instructions, the programmer writes parameters defining operations to be performed by the interpretive routine according to rules laid down when it was created. These parameters are in fact instructions in a different sense.

Processing During Loading

SAC illustrates most of the ways in which a program can be processed on its way into the machine; foremost among these are the conversion of symbolic into numerical addresses and the conversion of mnemonic 3-letter function codes into the binary digital form used within the machine. In scientific applications the acceptance of numbers written in a great

variety of different forms is also useful, and if a library is available for automatic reference, the loading process must recognize symbols which stand for a subroutine and select that subroutine to be copied from the library.

Debugging

Finding mistakes in his programs is one of the most tedious chores facing a programmer, and it is made much easier if he has plenty of useful information about events within the machine during the run. Information about the final state of the machine is given by post-mortem routines. If an interpretive technique is being used, it can with slight elaboration be made to provide suitably selected information about the actual running of the program.

If the program has undergone considerable processing during loading, all devices for providing evidence for debugging purposes should reverse this process and present their evidence in a form corresponding to that originally used by the programmer. This may be even more difficult to plan than the original processing.

Elaborate conversion and interpretive schemes do, however, have one great advantage when it comes to debugging. Out of the very large variety of things which the programmer is at liberty to write on his program sheet many will be nonsense; and moreover many of these nonsensical things can be automatically identified as such during one of the processes through which the program passes. Thus SAC will automatically indicate which of many different rules a programmer has violated in drawing up his program. To this extent, therefore, the debugging is practically automatic.

The Principle of Imitation

Although in TAC the store was described as being a magnetic drum, this fact was not essential knowledge for a user of the machine. The physical nature of the SAC store has in fact not been disclosed. To a user, a whole machine may be summarized as a set of rules for writing instructions and data, and a list showing the machine's response to each instruction (the instruction code).

We have just seen, however, that the rules for writing instructions and data depend entirely on the programs used for feeding them into the machine, and that the instruction code can be augmented or completely changed by means of subroutines of greater or less complexity. Consequently, by the use of suitable programs, a machine may be made to appear, logically, as a different machine; i.e., it can <u>imitate</u> another machine. In these ways, both TAC and SAC have been imitated by Whirlwind, which is basically quite unlike either.

The Limitations of Automatic Coding Techniques

There are unfortunately several obstacles in the way of extensive exploitation of the foregoing ideas. Firstly, all these techniques absorb some machine time which might otherwise be saved. Secondly, they also use up part of the storage capacity of the machine. Thirdly, (and this is a point which is easily underestimated) they require a considerable amount of programming effort to prepare, and some effort for the user to learn. Fourthly, there are limits set by the nature of the problem and by the transcription devices available. Let us consider these points in turn.

The processing of a program during loading does not usually take any appreciable time and, where it is done, it is nearly always well worth the machine time required. Interpretive techniques, on the other hand, slow down the rate at which the actual execution of the program takes place, and may (in the case of long computations) absorb considerable amounts of time. Objective comparisons are impossible because interpretive subroutines are normally only used in cases where the machine's instruction code is inappropriate to the job; the only alternative to the use of an interpretive subroutine is then presumably to use several small subroutines or to write a very long and circumlocutory program without subroutines. The former would probably save little or no time; the latter might save machine time but might also require stupendous amounts of programming time. Nevertheless it should be borne in mind, when considering the application of an interpretive technique, that the machine time involved is liable to be large and must be justified by savings in other directions.

The number of instructions contained in many existing program-processing schemes is of the order of several thousands. Where auxiliary storage is available, it is usually adequate to hold the processing routines, but the absence of an auxiliary store severely limits the facilities that can be provided. Moreover, an interpretive routine, which operates during the execution of the program rather than during loading, must be held in the high-speed part of the store during the computation and so limits the high-speed storage space available for the program. Again, however, the only alternative may be worse: to perform the same computation without an interpretive routine may require much more storage space.

The work entailed in preparing a program-processing or automatic coding scheme does not end with the writing of the processing routines, or even with their successful debugging. It is then necessary to prepare a description of the scheme as it affects the user. Much of the effectiveness of a good scheme can be lost if the description is badly prepared. The aim is to save the future programmer's time; this can unfortunately only be done at the expense of having him spend some time learning the rules. This is both a psychological and an economical barrier which it is important to minimize. Clear writing is essential, with the subject matter so arranged that no person need read passages that do not directly concern him.

It has been stated that the net result of automatic coding techniques is to cause one machine to imitate another. Alternatively, they may be thought of as allowing the programmer to write his programs in a different language, one that is translated to the machine's internal language by the processing routines. In order to make things easy for the programmer, we should make the new language as convenient as possible for him to use. The original statement of the problem to be solved comes to him in some language, and his job is to convert it into a language that can be presented to the machine. We should make this gap as narrow as possible, i.e., the language accepted by the processing routines should be as similar as possible to the language in which the problems arise.

Here, however, there are technical difficulties. Firstly, the languages that are used in practice contain an immense variety of symbols (especially in the fields of science and engineering). Not only would the automatic processing of such information be complicated, but there simply do not exist transcription devices with keyboards adequate to handle all the symbols (and any manual translation would defeat our purpose). Secondly, (and this is more fundamental), very few of the seemingly precise statements made even in technical subjects can in fact be interpreted unambiguously without an intelligent knowledge of the subject matter -- and this is quite beyond the ability of machines at present.

Notwithstanding these formidable obstacles, automatic coding is slowly advancing. No one system can be considered the best for all purposes; there will inevitably be many, appropriate to different types of applications. There is a tremendous amount of work entailed in developing each one, and we must not expect any spectacular advances.

18. OPERATIONS RESEARCH

by Prof. Philip M. Morse, reprinted from
MECHANICAL ENGINEERING, March, 1954

Operations Research is the application of research techniques to the study of the operations of war and peace. It examines what occurs when a team of men or machines does the job assigned to it. It is an activity; a pattern of operations, susceptible of being related to other diverse activities. Its applications encompass such unrelated matters as determining time of waiting in line in a restaurant; fixing the inter-relation between sales fluctuations, size of inventories, and production scheduling; or developing a pattern of search operations for an enemy submarine or aircraft.

In Operations Research there is an opportunity for scientists and engineers to help in administrative problems, not by becoming the administrator, but by providing the administrator with quantitative understanding of aspects of his operational problems, so that he can reach a wise decision, fully conscious of the implications of his choice.

This is a progress report on a relatively new branch of applied science. First utilized during the last war on military problems, it proved valuable enough so that most military staff, here and in England, now have operations-research groups. More recently its usefulness in industry is coming to be recognized, and groups are also being attached to top industrial staffs. The Operations Research Society of America, formed two years ago, now has nearly 1000 members; its Journal is now in its second volume. What is this new activity, and how is it related to other branches of science and engineering?

Defining a branch of science in a few nontechnical terms is not easy. Perhaps the safest definition is that Operations Research is the activity carried on by operations-research groups and reported in the Journal of the Operations Research Society of America. But often some less circular sort of definition is desirable. Students who may wish to

learn about the field have to be told what it is; people want to know what it's good for; workers in related fields want to know why it should be differentiated from their own fields, and so on.

What is Operations Research?

Operations Research has been defined as the application of the scientific method to problems of management, but this is obviously too concise and too general a statement. There are many sorts of "scientific methods," and many sorts of people study problems of management. Part of the definition must describe the way these problems are studied. Here the word "research" in the title may give a hint. The research scientist, at least in the physical sciences, uses the quantitative language of mathematics, employs the well-known but difficult-to-describe procedures of experimentation and theory making. He looks at the phenomenon he is studying in a certain impersonal way, being more interested in how than in whither, more interested in why than in for what use. Many centuries of experience have taught him that this impersonal viewpoint, this dual employment of theory and experiment, will usually procure for him results of value in his science and that too great a preoccupation with questions of the worth of the result, or the immediacy of the need, actually will hinder his progress.

Operations Research, then, is the application of research techniques to the study of the operations of war and peace. It is concerned with an attempt to understand something, in the scientific sense of the word "understanding." It is an effort to discover regularities in some phenomenon and to link these regularities with other knowledge so that the phenomenon can be modified or controlled, just as other scientific research does. The difference comes in the phenomema which are studied, the subject matter. Instead of studying the behavior of electrons, or metals, or gasoline engines, or insects, or individual men, Operations Research looks at what goes on when some team of men and equipment goes about doing its assigned job. A battalion of soldiers, a squadron of planes, a factory, or a sales organization is more than a collection of men and machines; it is an activity, a pattern of operation. These operations can be studied, their regularities can be determined and related to other regularities; eventually they can be understood, and they then can be modified and improved.

Research at the Operational Level

Operations Research is concerned, not with matter or with individual machines or with men, but with the operation as a whole; with battle tactics, with strategic and logistic planning for future operations, with the interrelation between sales fluctuations, size of inventories, and production scheduling, with the flow pattern of goods in a group of factories or of traffic in a city, to mention a few examples. We might use the word "level" to distinguish between the different subject material,

if we can divorce the word from any connotation of relative importance or difficulty. Physics and chemistry would then correspond to research at the basic or material level, the study of bridges and television sets, research at the engineering or applied level. Operations Research would then be "research at the operational level."

Although the name is relatively new, research at the operational level is not new, of course. Taylor and his followers, in their time-and-motion studies, have investigated a small part of the whole field, traffic engineers have been working on another part, systems engineers encroach on it, and so on. Perhaps the most useful service the new term Operations Research has performed is to emphasize the essential unity of the whole field, to force the recognition of similarities in behavior in areas hitherto separate, and to make apparent the broad usefulness of a number of research techniques and mathematical models.

Techniques Used

I have not yet said anything about the techniques used in Operations Research. As with other research, any technique of measurement or of calculation, any portion of a basic science is used which will produce results. We should expect that the theory of probability and of statistics would be very useful tools; we also should expect that the techniques of the psychologist would be needed in other cases. This does not mean that Operations Research is applied statistics, on the one hand, or is a branch of social psychology, on the other. It uses any and all of these disciplines to study operations in order that they may be understood and thus controlled. Since a wide variety of basic science is involved, much of the research can best be carried on by a team of workers having a variety of background training, each contributing his specialized knowledge to the solution of the operational problem. The advantage of a mixed team for the study of many operational problems is obvious. In fact, some persons have said that the use of mixed research teams is a characteristic of Operations Research. It certainly is important in many investigations; whether it is characteristic or necessary might be questioned.

But certainly further generalities will not be helpful here; a few specific examples may help clarify the picture. Certain particular aspects of operations have been the subject of intensive study in the last few years, and special mathematical models have been developed to help understand the phenomena. As is usual with models, they represent only part of the phenomena, and since Operations Research is new, most of these models need further development before they can be satisfactorily general in their applicability. Here the Operations Research worker needs the help of the basic scientist, particularly the mathematician.

Waiting in Line

Take the simple business of waiting in line - the British call it queuing. All of us do it too much of the time; if we drive to work in the morning, we wait at traffic lights; if we go to a cafeteria at noon, we wait for our lunch. It is the headache of many businesses; it is a vital problem for airlines when an airport clouds in and the planes begin to stack up, waiting to land. Let us see what can be said about this sort of problem.

We start as usual with a fantastically simplified case, one where the front of the line is served at some constant rate, say, S per second, and where the rear of the line is being filled up by people (or planes) coming in at random times but with an average rate of arrival, A per second. We also will assume that this has been going on long enough so that a steady state has been reached; we can consider the transient case as a later elaboration. The key to this mathematical model lies in the working out of the various probabilities that the line will have 0, 1, 2, or n persons in it. Call the probability that there are n persons in the line P_n.

If P_{10} is large, for example, this means that we are quite likely to find 10 people ahead of us when arriving in line; what the restaurant tries to do is to make P_0 large.

To have a steady state none of the P should change with time.

But every time a person arrives, all the P step up one by one, P_0 changes to P_1, and so on; and every time a person is served, they all change downward. So, in order that A persons arriving a second and S being served a second will not change the probabilities continually, they must be related in some special way. For example, the rate of disappearance of a line of zero length is AP_0, the rate of arrival times the chance that a zero-length line is there; the rate of appearance of a line of zero length is SP_1, the rate of serving times the chance that a single-length line is present. To have a constant probability of zero-length line, we must have these two rates balance; $AP_0 = SP_1$. Similar balance for lines of unit length, of length n, and so on, gives rise to the sequence of equations.

$$AP_0 + SP_2 = (A + S)P_1;\ AP_{n-1} + SP_{n+1} = (A + S)P_n$$

and so on.

These can be solved without much trouble, giving

$$P_n = (S - A)\ (A^n/S^{n+1})$$

as long as the rate of serving S is larger than the rate of customer arrival A. It is obvious that if customers are arriving at a rate faster than they can be served, the line cannot ever be stationary in length and, if they value their reputation or peace of mind, our restaurant or airport managers must avoid this at all costs. But even when

customers arrive more slowly than they can be served, we see that there is a finite chance that a line will form. In fact, the average length of the line turns out to be A/(S - A).

This quantity is quite small as long as the maximum serving rate S is at least twice the arrival rate A. But if people arrive nearly as fast as they can be served, the average waiting line rapidly lengthens; if A is 0.8S, then the average number in line is 4; if A is 0.9S, the line has 9 in it, on the average, and so on. For example, if A is 0.8S, if customers are served 25 per cent faster than they arrive, on the average, then 20 per cent of the time there will be no line, 16 per cent of the time one will be waiting, 13 per cent of the time two will be in line, 8 per cent of the time four will be waiting, 2 per cent of the time ten will be in line, and so on; the average line length will be four.

It may seem peculiar that there should be any waiting line when the mean rate of service is greater than the average rate of arrival; this is due to our assumption of randomness in service and arrival. We assume that each customer doesn't conveniently arrive just when the last customer has been served; the customers arrive at random, which does not mean regularly. Also, one customer may take longer to be served than the next, and a bunch of customers every now and then arrive just when a slow-poke is being served.

These random mismatches between customer and server don't matter much if the service is considerably faster than the average rate of arrival; once in a long time two or three may come in a bunch, but most of the time no one is waiting. But if customers arrive nearly as fast as the line can be handled, these mismatches occur more and more often, and the chance of a long line occurring quickly is large. Of course, if the servicing process could be made absolutely regular, each service completed exactly in 10 seconds, for example, and if also we could regiment our customers to arrive exactly 10 seconds apart, so that one walked in the door exactly at the end of each 10 seconds, then S could equal A, and still no line would form.

But service is very seldom as perfectly timed as this, and we practically never can regiment the arrivals. Customers, automobiles, and airplanes do arrive in a random manner at restaurants, street intersections, and airports, and it turns out that the results of our simple quantitative reasoning fit actuality remarkably well in spite of our preconceptions to the contrary. Here is a case where theory and actuality contradict our intuitive "feelings."

In every case where this theory applies, gross errors of estimate have been made, regarding the expected length of waiting lines, on the basis of nonmathematical "hunches." Often long arguments have occurred before the manager would be willing to face the consequences of the theory. They would continue to say, "But why should there be a waiting line when I can serve them faster than they are coming?" in spite of the line which was there before their eyes. The results of such irrational behavior only produce irritation in the case of restaurants, gasoline

stations, and the like; it is much more serious in the case of airports or docking facilities in harbors, particularly under wartime conditions.

Industrial Problems

The simple theory, sketched so quickly above, can be expanded and complicated almost indefinitely. For example, the problem of machine maintenance in a factory is of this sort. The machine can be said to "arrive in the waiting line" when it breaks down; it is "served" when it gets repaired. The flow of parts through an assembly line is another example. The theory can tell us how many parts must be kept on hand at each stage of the process, in order that no machine should be kept idle by delay in the earlier processing, for example. Many aspects of the over-all problem of industrial inventories also can be analyzed by this technique. Here it is the sales, the outflow, which has the large fluctuations; we need to balance between the requirement that orders be filled as soon as they come in and the added expense of running a factory overtime if our inventory runs out.

Another sort of problem which turns up in a large number of operational studies has to do with the optimization of some function of a number of variables, subject to boundary conditions which limit the range of the variables. For example, an oil company can produce various proportions of fuel oil, gasoline, and aviation fuel from its cracking plants, depending on the kind of crude oil used, and can produce various proportions of these end products from a given crude, depending on the cracking process used. But crudes differ in price, and cracking processes differ in cost. Suppose the company has orders for definite quantities of end products to be delivered in the next 3 months. What amounts of which crude shall it buy, and which processes shall it use in its cracking plants, to produce the required amounts of products at the least cost, subject to limitations of supply of crudes and of output of its plants?

The variables here are the various amounts of crudes to be bought and the degree of utilization of each plant. The function to be minimized is a linear function of these variables, and the limits on each variable are known accurately. Such a problem is known as a "linear programming" problem. There are many such problems which turn up in Operations Research. Techniques of solution are not simple, and many of them require high-speed computing machines; much further mathematical research is needed to simplify computing procedures in linear-programming calculations.

Parenthetically, the optimization of the crude-oil-cracking problem has been worked out by the research or the engineering departments of many large oil companies. The persons who worked out those solutions did not call what they were doing Operations Research; many of them had not heard of Operations Research. It is also true, however, that most of these workers were not aware that many other problems in the company's operations were likewise amenable to the same analysis. The value of

the concept of Operations Research to these companies lies in making their research men aware that the techniques of theoretical analysis they have been using for one problem can be applied to a much wider range of operational problems than they hitherto had conceived, and in showing the company executives that they can use their own research departments to help solve production and sales and distribution problems, where formerly they had not been used.

Linear Programming

The linear-programming problem can be visualized most simply in geometrical terms. The n variables define an n-dimensional space; a point of this space corresponds to a solution. Each limitation on the range of the variables corresponds to a hyperplane in this space, restricting the allowed solution points to one side of the hyperplane. By the time we have finished specifying all the restrictions (negative production not allowed, maximum limits on storage capacity, limits on production, and so on) we find that we have surrounded the region of possible solution by hypersurfaces, so that the allowed region is the interior of a convex polyhedron in the hyperspace. If the function to be optimized is a linear function of the variables, then the requirement that this function have some constant value also corresponds to a hyperplane which may or may not cut through the polyhedron; if it does, it then corresponds to an operationally possible value of the function to be optimized. By changing the value of the constant, we can generate a family of hyperplanes parallel to each other, their distance from the origin being proportional to the value of the function to be optimized.

Some of the hyperplanes in this family cut through the polyhedron containing the region of solution; some do not. There are two limiting hyperplanes, one corresponding to the largest value of the function for which the hyperplane just touches the polyhedron, and one corresponding to the smallest value which just touches. Consideration of the geometry shows that for most orientations of the family of planes the two limiting planes just touch a vertex of the bounding polyhedron and thus contain just one possible solution compatible with all the boundary conditions. The outermost limiting point is the optimum solution if the function is to be maximized; the innermost point is optimum if the function is to be minimized. Once the geometry is clear in one's mind, it is easy to visualize the solution. But, at present, it is not easy actually to compute the optimum vertex when there are several dozen variables and about a hundred boundary faces of the polyhedron.

Production Planning Needed

Important as linear-programming techniques are, they need further generalization to be able to solve many problems in Operations Research. Production planning is an example. A factory can produce so many units of some product each month, but sales of the product are small during

the summer and very large in December, so large that fall production cannot equal December sales. One solution is to run the factory overtime during the fall; but overtime production costs more than normal output. Another solution is to produce more during spring and summer and store it ready for the winter rush; but warehousing also costs money, in storage and handling charges and in interest on the money tied up. A third solution of course is to fail to meet orders in December, but this is a counsel of despair.

It should be evident by now that this is also a linear-programming problem. The variables are the regular production each month and the overtime production each month. The excess of production over sales each month is warehoused. The boundary conditions are the limits on the production and overtime production each month and the additional requirements that the total production from the first of the year shall never be less than total sales from the first of the year. The quantity to be minimized is the total cost, including overtime charges and warehousing charges.

As stated, this is a straightforward linear programming problem, if we can predict exactly our sales throughout the coming year. If our sales forecast is exact, we can proceed to find the distribution of production and overtime production each month to minimize total costs and to satisfy all forecast sales. The trouble is we never know exactly what the sales are going to be, and if we have underestimated them, we will not be able to meet orders; if we have overestimated them, we will end the year with unsold product in our warehouse. All we really have is a probability distribution of expected sales; to put it pictorially, some of the sides of the bounding polygon are fuzzy, not sharp.

Problems of "Bounded Optimization"

At present, our techniques of solution are not adequate for such problems, nor are they if the function to be optimized is not a linear function of the variables. Such more general problems might be called problems of bounded optimization. The problems are clear, but a great deal of further analysis and devising of computational techniques is needed before solutions can be obtained with the requisite ease. Speed of solution is needed here, for in many cases we wish to find a whole sequence of solutions as we vary some of the limits: What happens if we build another factory, or if we close down our factory in August, for example? When solutions of problems of bounded optimization are easy to obtain, many tough problems of planning, of production, of sales effort, of logistics, and so on, will be easier to solve.

Another kind of problem for which a mathematical model can be built came up first in naval operations research but has numerous business analogs. It concerns the operation of "search" for an enemy vessel, or submarine, or aircraft. The enemy is somewhere in a given area of the sea. How do you deploy your aircraft to find him? The central idea here is the "rate of search." A single plane can see the enemy vessel (by radar or sonar or visually as the case may be) R miles away, on the

average. The plane can "sweep" out a band of width 2R as it moves along; the picture is analogous to a vacuum cleaner, of width 2R, sweeping over the ocean at a rate equal to the speed of the plane and picking up whatever comes beneath it. An area equal to the speed of the plane times twice the mean range of detection will thus be swept in an hour. The sweep rates of planes vary from a few hundred square miles per hour to several thousand square miles per hour, depending on the plane, the radar equipment, and the vessel searched for.

If the enemy is equally likely to be anywhere within a certain area, then the problem is a straightforward geometrical one. The search effort is evenly laid out over as much of the area as one has planes available. The problem is a little complicated by the fact that detection is not certain at extreme ranges, so the probability of detection falls off near the edge of the swept band and there should be a certain amount of overlap between bands to improve the chance of detection near the edges.

But if the chance that the enemy is present varies from area to area, the problem becomes quite difficult; non-mathematical intuition may lead to quite erroneous use of available effort. For example, if the enemy is twice as likely to be in one area than in another, then, if only a small amount of search effort is possible, all this effort should be spent in searching the more likely area. If more effort is available, some time can be spent on the less likely area, and so on. A definite formula can be worked out in each specific case. Search plans for various contingencies were worked out by the Operations Research team attached to the Navy during the war; they materially aided the naval efforts in many cases.

From War Effort to Industry

It seems a far cry from planes and ships and submarines to industry and business activities. But the utility of the mathematical models is their wide range of applicability. One possible business application of search theory comes in the problem of assignment of sales effort. Suppose a business has a limited number of salesmen, who are to cover a wide variety of dealers. Some of these dealers are large stores, which usually will produce large orders when visited, some are small stores with correspondingly smaller sales return. If there are enough salesmen, every dealer can be visited every month, and the optimum number of sales can be made, although the sales cost will be high. With fewer salesmen available, search theory indicates that the larger stores should be visited more often than the small stores; with very few salesmen it may be that only the large stores should be visited. If the probable return per visit for each store is known, the optimum distribution of sales effort can then be calculated.

An interesting and typical variation on this problem comes when we consider the action of the individual salesmen, when we try to make their behavior conform to the best over-all distribution for the company. For

each individual salesman, with his limited effort, it may be best for him to visit only the large stores; if his visits are uncontrolled and if he is paid a flat commission, it may turn out that the large stores are visited too often, the small stores too seldom, for best returns for the company as a whole. It then becomes necessary to work out a system of incentive commissions designed to induce the salesmen to spread their efforts more evenly between large and small customers. If the general theory has been worked out, this additional complication can be added without too much difficulty.

This problem of balancing the tendencies of different parts of a large organization is one which is often encountered in industrial Operations Research. The sales force is out to increase sales of all items, though some items may return less profit than others. Production resists changeover to making another product, though sales on the other product are increasing; and the financial department frowns on building up large inventories, though small inventories always put the production division at the mercy of sales fluctuations. It is often not too difficult to suboptimize each of these divisions separately, so each is running smoothly and effectively in so far as its own part of the business is concerned. But to be sure that all these parts mesh together to make the company as a whole operate most efficiently requires much more subtle analysis and very careful quantitative balancing.

In the interest of reducing factory overtime and to keep down inventory, for example, it may be necessary to modify the salesman's incentive commissions, so he will be induced to push one line over another. It may be necessary for the production division to allow more overtime in one department than another, to make some part of its operation run at less than optimum in order that the over-all operation be optimum; and one must take care not to penalize the production department, by reduced bonuses or the like, for reducing its efficiency so that the effectiveness of the whole is improved.

But perhaps these few simple examples are enough to show that the research techniques developed to increase our understanding of the nature of the physical world also can be used to help us understand operational problems. In many cases in industry and war, a simplified quantitative model of the situation can help us see what goes on and can help us devise the best way to proceed. In many cases it is not necessary to have a complete picture of all that goes on, clear down to all the basic details. As long as our mathematical model can be adjusted to fit some of the regularities which appear, we can abstract these parts of the behavior from the rest and study them separately. The process of abstraction, of keeping clear of local details, has the advantage of providing a model which may fit a variety of circumstances - restaurants, production lines, or landing aircraft. By gaining in generality, of course, we lose in detail.

Perhaps it also can be seen that such methods probably cannot be used to solve all problems. Just as it is quite unlikely that the methods of analysis used so successfully in genetics can be used to solve all biological problems, for example, so it is unlikely that the

Operations Research scientist, with his specialized techniques of analysis, can ever replace the usual business executives or army generals, with their practical experience and their intuitive grasp of the complicated effects of morale and applied psychology, for example.

But as new techniques are tried in more and more different fields, it should become clear what operational situations can be analyzed by its means and what situations cannot. Already there are Operations Research teams working closely with military and industrial administrators, exploring these possibilities, reporting their findings to the administrator that he may be able to combine their quantitative results with his experience and judgment to reach more understanding decisions.

In general, scientists and engineers have not been active in administering government or business. This is not surprising, for the business of science is to understand, not to act. In Operations Research, however, the scientist and engineer can provide a better understanding of operational problems so administrative decisions can be made wisely.

20. COST REDUCTION THROUGH ELECTRONIC PRODUCTION CONTROL

by

R. G. Canning

Reprint of paper presented at the semi-annual meeting of the American Society of Mechanical Engineers, Los Angeles, California, June 28 - July 2, 1953 and published in the November, 1953, issue of Mechanical Engineering.

Of the four main aspects of cost reduction that come to mind probably the most common is that of product improvement; redesign of the product to simplify or eliminate parts, making for easier fabrication, and so on. The second aspect is methods improvement, a familiar subject to those in Industrial Engineering; this calls for more efficient use of tools, work space, motions, and the like. The third aspect is better utilization of productive facilities; this includes production planning, loading, and scheduling, and covers more efficient decision-making and more effective control. The last aspect is reduction of overhead, by the mechanization of the office.

In a current paper, Dr. M.E. Salveson[1] indicates a mathematical framework for the loading and scheduling of productive facilities. The development of such a mathematical model is most important, because it would provide production management with a systematic means of determining optimum (or near optimum) loads and schedules. The application of mathematical methods in practical situations undoubtedly will depend to a great extent upon the use of electronic data-processing equipment. However, the introduction of such electronic equipment also can result in the reduction of overhead by the mechanization of the office, if an adequate systems design is considered from the outset.

The main points made in this paper are that, in the author's opinion, the primary value of electronic production control for cost reduction will be in the form of increased output of product, using the same productive facilities (although it is realized that this "intangible" gain is often harder to sell to management); then to a lesser extent electronics also will reduce overhead, by replacing clerical employees.

To give a clearer picture of how these cost reductions might come about and why the two points are so rated, a sketch of an electronic system designed for one local company that shows promise of meeting these objectives, and an order of magnitude of these two types of cost savings, will be presented.

---

[1] "A Computational Technique for the Scheduling Problem", by M.E. Salveson, presented at the Semi-Annual Meeting, Los Angeles, California, June 28-July 2, 1953, of the AMERICAN SOCIETY OF MECANICAL ENGINEERS.

Mechanization of the Office

For logical sequence of presentation, it will be necessary to consider the latter of these two points, the mechanization of the office, as background material for the first point. One of the major objectives of the project, as set forth by Dr. Salveson[2], is the design of a master scheduling computer which will fulfill the functions of loading and scheduling production operations. In attacking the problem of how to design such a machine, it was apparent that a data-handling system would be necessary for two function: (a) to translate the schedule generated by this master computer into specific shop instruction, and (b) to measure and feed back the actual rate of progress, as initial conditions for the next scheduling computation. The present state of the electronic-computer art is such that a data system to perform these functions appears quite feasible even though the design of the master scheduling computer may not be as yet. Furthermore, a data system might well pay for itself in a short time by means of savings in clerical salaries, and thus pave the way for the introduction of the master scheduling computer at a later time.

To investigate this application of electronic machines to production data processing, a two-phase study was planned. The first phase was to consist of a number of plant visits to companies in the Los Angeles area to determine some of the characteristics of those firms which might be interested in electronic data system, i.e., number of employees, type of product, type of production organization, and so on. The second phase was to locate one firm in the local area that met many of these requirements and study it in detail, with the aim of designing an electronic system to meet its needs. Thus we started out with one objective in mind of "mechanizing the production-control office" -- we wanted to find out where employees could be replaced more efficiently by electronic machines, and an indication of how many employees could be so replaced.

The remainder of the paper will be devoted to presenting some of the conclusions of this two-phase study, with respect to cost reduction. Based on the results of the first phase of the study, we will first split the field of production into two main segments, and choose one of them for analysis; within this segment, we will point out the types of firms most in need of electronic production-control systems. We will state briefly the present methods used by such companies and finally, by the example of the case study, we will show how electronics more nearly can provide what is desired in the way of a production-control system for the firms.

One main segment of the production field about which much has been written recently (especially with respect to its probable use of electronics) has been given the name of "automation" -- automatic materials handling and automatic control of the continuous production

---

[2] "On a Quantitative Method in Production Planning and Scheduling," by M.E. Salveson, Econometrica, vol. 20, October, 1952, pp. 554-590.

line. The continuous line appears to be the objective of much of American industry, in order to achieve mass production and low unit cost. The use of electronics here would be that of control, to replace some production employees in the routine operations of meter-watching, switch-throwing, and so on. This is an important field and is receiving considerable attention today, not only by industrial engineers but also by electronic and servosystem engineers. However, in a continuous-line plant, a relatively large production-control staff is usually not needed. The main production problem is estimating the size of the market (rate of demand) and then adjusting the rate of production to meet this demand. The small number of clerical employees in the production-control office does not hold much hope for necessary cost savings in such firms.

Applying System to Job-Shop Production

Rather, it is the other segment of the production field with which we will be concerned -- the job-shop operation, stressing customer service rather than mass production, where a large variety of products is possible, and production is to customer order rather than to finished-goods inventory. These are the firms that do have a relatively large production-control department. The results of the first phase of the study can be stated briefly. It was evident that job-shop plants with less than 500 employees probably could not justify the purchase of an electronic production-control system; also, certain plants with over 1000 employees almost certainly could consider the purchase of such a system. Firms with over 500 but less than 1000 employees varied in their need, and would require individual detailed analysis. It is at this size of plant (over 1000 employees) that the concept of "freedom of choice" at the foreman level becomes important.

As an illustration of this concept, the continuous-line plant obviously has little freedom of choice as to which job will be worked on next; the next job coming down the line will be the one. If the necessary materials do not arrive at the right place at the right time, the line stops and the general manager of the plant knows about it in a matter of minutes. At the other extreme, in the large job shop, there are any number of jobs waiting to be worked on in any department, and the foreman is faced with the task of choosing which sequence to work them.

Other forms of this "freedom of choice" problem were encountered, including plants with a large volume of engineering changes (as in the aircraft firms), a complicated payroll structure where any one employee might work on as many as ten different wage rates during a week, and large-volume inventory control where it is difficult to pick up the individual disbursements easily.

In the course of these plant visits, one plant stood out above the others as an interesting one for a detailed study. At this company, the production-control manager had developed quite an efficient manual system of production control, by the use of centralized inventory

control, a priority system based on due dates, control boards in each department showing the due dates of the jobs in that department, and so on. With about 1000 employees in the plant, there were only eight expediters (or about 0.8 per cent of the total employees), which compares very well with the 2 to 5 per cent figure reported in other plants. It was felt that if an electronic system could "compete" with the present manual system, this would augur well for the use of electronics in other plants.

The company produces on a job-shop basis -- to customer order, rather than to finished-goods inventory. A large variety of products is possible, and individual customer specifications are common. Since the firm does much of the fabrication of component parts, and all of the assembly, this means that the loading and scheduling problems are present.

Now, restricting attention to this type of plant, let us assume that customer orders are received and processed. This processing calls for a fair amount of clerical effort-- exploding bills of material, posting requirements, recapping requirements and comparing with inventory, deciding whether to enter an order (shop or purchase) to obtain the parts, preparing such an order, investigating the status of raw materials and ordering them if needed. Let us assume that several thousand such orders are in the plant or in the purchasing department at any one time. To make matters realistic, we will consider that some of these thousands of orders are behind schedule, owing to machine breakdown, tooling troubles, reworks, and so on. Finally, let us concentrate our attention on the production controller, or chief expediter, whose responsibility it is to coordinate activities so as to get all of the parts of one customer order into the assembly department at one time.

The Chief Expediter's Job

Under present methods, what are the activities of such a person? Obviously there are too many orders in the shop for him to remember the status of all of them. So decentralization is used; expediters are assigned to groups of departments to watch the status of orders within those departments. One of the main functions of these expediters is to observe the "exceptions", e.g., orders behind schedule more than a certain amount, and either take corrective action themselves or report the matter to the chief expediter. The other important source of such "exception" data is from the assembly department, which informs the chief expediter that certain assemblies cannot be worked since they have parts missing.

Since there is no regular flow of work within the shop, it is next to impossible to predict how many shop orders will arrive in any given department during a day. Thus bottlenecks can and do develop overnight. When bottlenecks do occur, the decision must be made immediately whether to authorize overtime or send some of the jobs outside on a subcontract basis. The former course has its obvious drawbacks, while the latter course costs heavily in time, in order for

bids to be requested, and received, tools and materials shipped to the outside firm, and so on. Life often becomes a continuing succession of "putting out fires" for production-control management.

What Electronics Offers

What does electronics have to offer for such a situation, in so far as the mechanization of clerical functions is concerned? Fig. 1 is a flow diagram of the proposed method of processing customer orders at the company studied. The first operation is that of typing a standard sales order from the customer's order. Such items as inspection, procedure, renegotiation clause, customer code number, and product code must often be added to the information supplied by the customer. By using a special electric typewriter, a punched paper tape is obtained in addition to the regular typed document. This punched paper tape has the information in a form suitable for direct entry into an electronic data-handling machine. Bills of material are prepunched into punched cards (which are still important "building blocks" in a systems design, even with the advent of magnetic tapes, and the like). The appropriate decks are selected by the operator and fed into the machine. The machine combines these variable data (quantities, due dates, and so on) with the standard data and posts them on the requirements magnetic tape. After all postings are made, the machine scans the requirements for each part number, and compares it with the inventory data for the same part on the adjacent tape. Parts that may need ordering can be used for a "loading" computation described by Dr. Salveson, or the decision on what and how much to order can be made by the human operator after the machine prints out the facts on these questionable items. In addition to storing the order information inside the machine, we also ask the machine to prepare the customary papers to which we are accustomed and cannot live without.

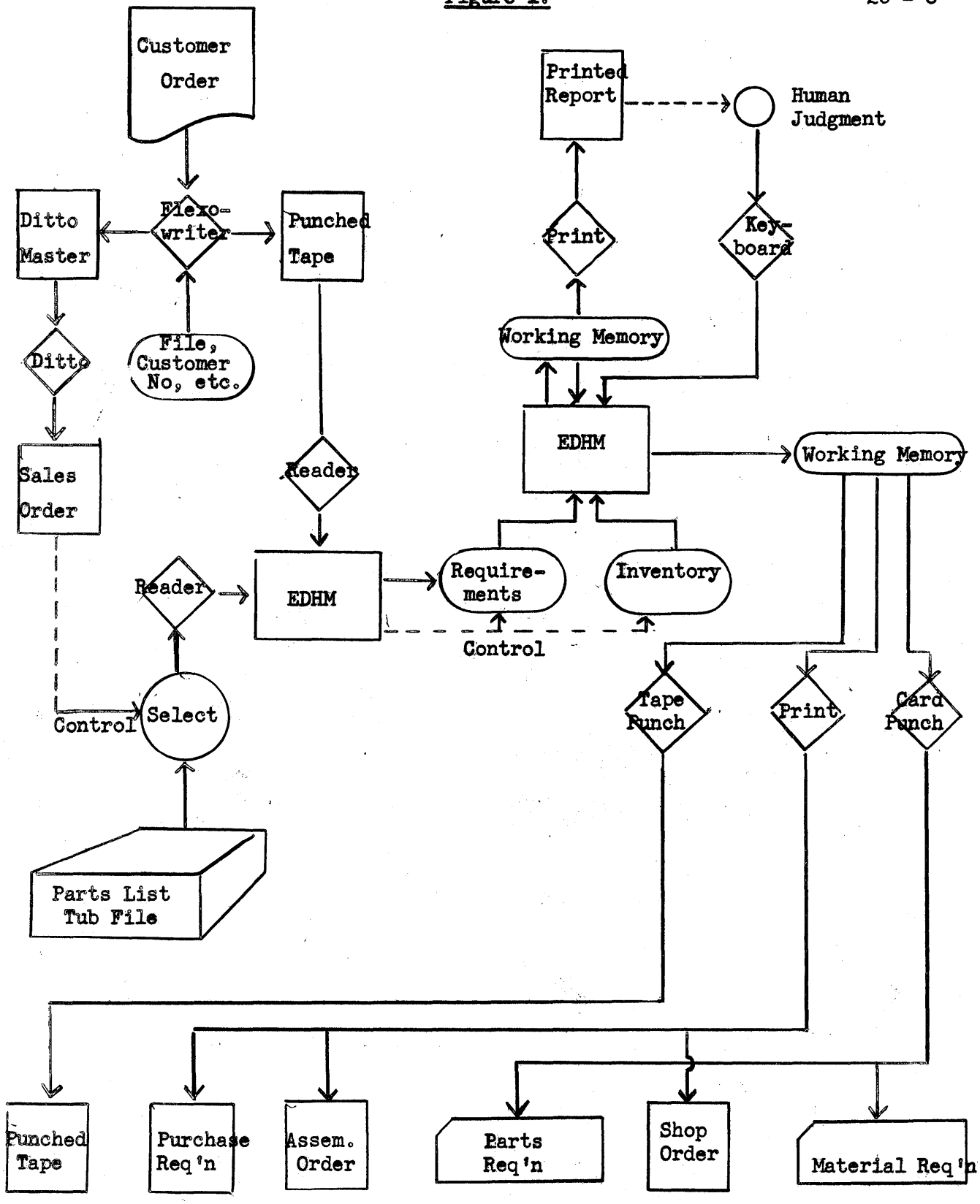
What has the machine done so far? Nothing that is not done already by manual methods, except that the operations of writing, computing, sorting, selecting, and so on, are done by machine instead of by clerks. As an important by-product, we have the pertinent information stored in the machine, where it can be used for other purposes. In much this same way, information on the progress of shop orders within the shop can be picked up and stored in the machine. Space does not permit a discussion of how this is accomplished in the system we propose, but further details may be obtained directly from the author or from a special report on the subject.[3]

Now, what does the production controller do with an electronic system? Fig. 2 is the block diagram of the analysis part of the proposed system.

---

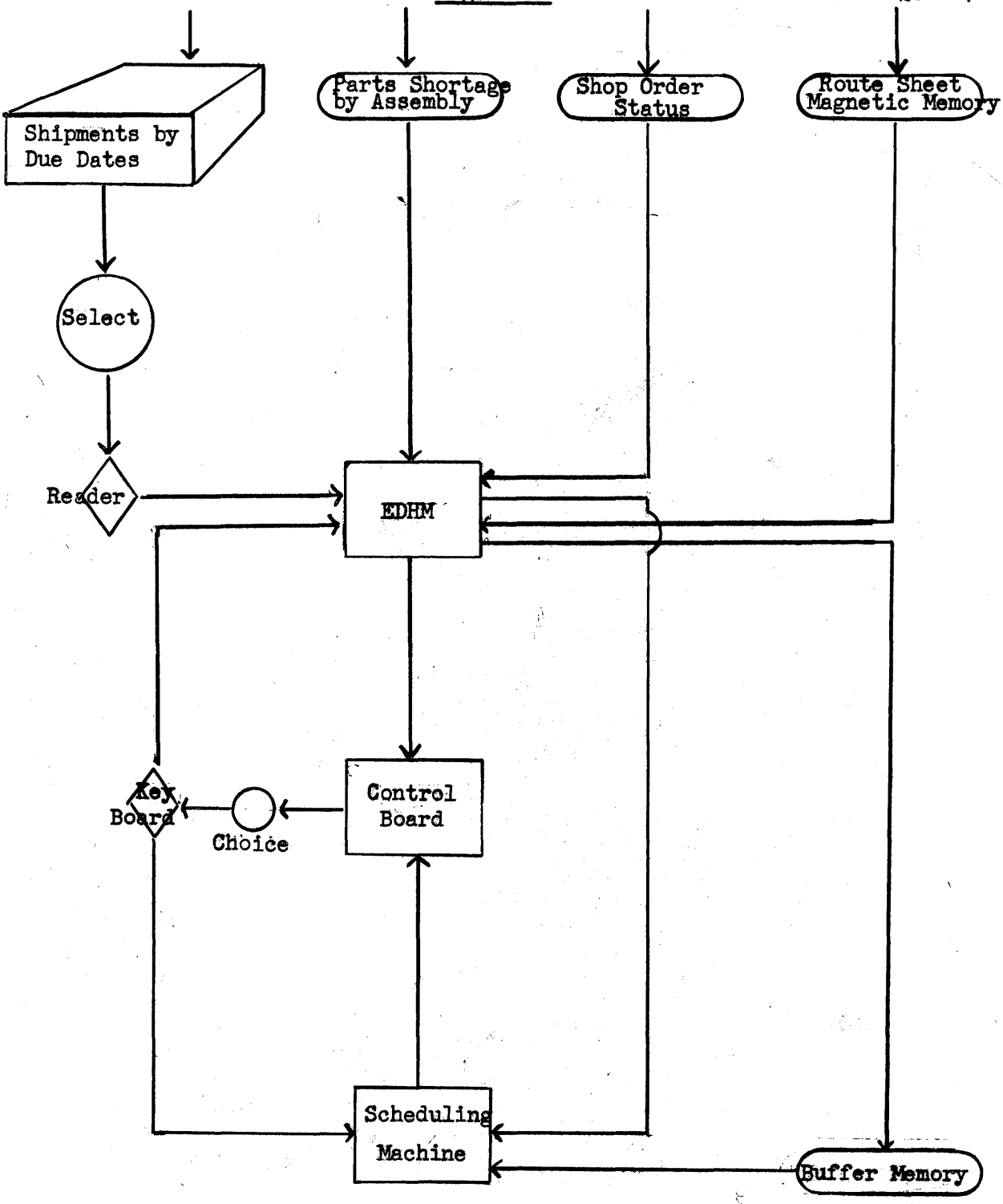
[3] "A Proposed Electronic Data Handling System for Production Control", by R.G. Canning, Research Report No. 10, February, 1953, Industrial Logistics Research Project, University of California, Los Angeles, California

Figure 1.



Posting Requirements and Ordering

Figure 2.



CONTROL AND EXPEDITING

To begin with, we see another tub file of punched cards; each card indicates a customer order for one month. If the customer enters an order for the same assembly over a period of several months, a similar card would be prepared for each month. These cards are then sorted and collated by shipment-due dates. As a first step in the analysis, the production controller selects the cards from the front of the deck; these cards represent shipments that are past due, due this week, and due during the next two weeks or so. These cards are then read into the elctronic data-handling machine (EDHM).

How EDHM Works

For each card, the machine then automatically refers to one part of its memory, a magnetic tape showing how many parts are short for each assembly order. The information is presented graphically so that the production controller's attention is directed, for example, to those assembly orders that are past due and have only one part missing. It is on such critical parts that he will concentrate his attention. He then asks the machine to indicate the part numbers of these critical parts.

The next step is to find the present status of all shop orders that are making these critical parts. To do this, the machine automatically refers to another magnetic-tape memory (there being at least four such tape units tied to the machine, each storing the equivalent of 12,000 punched cards). After this step, the production controller is able to concentrate his attention on the critical shop orders.

Notice the difference between the present manual systems and the electronic system. In the manual systems, no one man can keep track of the status of all orders in the shop, so that this function is split up between a number of men. In the electronic system, the machine has all the information available and presents the desired information on demand to the production controller, for his decision. Except for this, however, the electronic system is still not too different from the present manual methods using clerical help. The system so far, then, is the mechanization of the office.

Perhaps the reader is questioning why we have mixed up punched cards with magnetic tapes -- why not all or the other? Punched cards are still very useful for the operations of printing, sorting, and collating data. Also, they constitute an economical and efficient form of data storage where sequential access from small decks of cards is sufficient and where few changes occur in the data; it is hard to "erase" a hole in a card, for example. Magnetic tapes, on the other hand, have the advantage of automatic look-up (called random access), ease of erasing and changing the data, and no need for the machine operator constantly to feed new decks of cards into the machine. It is likely that for some years to come, electronic data-processing systems will make use of both methods.

The Scheduling Problem

Now let us consider the scheduling problem -- the anticipation of bottlenecks and the decisions on the most effective corrective actions needed. As was pointed out earlier, the large number of shop orders and other variables cause this to be a difficult decision-making problem for the production controller. Owing to the limited memory span of the human mind, the number of variables that enter into these decisions must be reduced to the point where one person can comprehend them.

Simplification of the manual scheduling operation is accomplished as follows: The main criterion of priority for a shop order is due date -- primarily, the date on which it must be ready for the assembly department, and then the individual operation due dates which must be met in order to achieve the final due date. If the shop order is for parts that are holding up the completion of an assembly, a higher priority can be given by setting back the individual operation due date until there is no other job in the department with an earlier date. However, if the coordination of several shop orders is involved, to make them all arrive at the assembly department at the same time, this is often too complicated a situation to solve mentally with any degree of accuracy, owing to all the interactions. The time estimate for a shop order to progress through several operations is not calculated from standard times plus waiting times, but is likely to be an average "flow time" based on experience. Thus "rules of thumb" must be used, and the expediters concentrate their attention on the "exception" orders.

It is in such a situation that electronics begins to show a marked advantage over manual methods. An important feature of the system is that very little additional equipment is needed for this function, since all the pertinent production data are stored already in the machine.

Referring again to Fig. 2, we have added a block called the scheduling machine. For those familiar with industrial engineering, this machine is an electronic analog of the well-known Gantt chart. For those not familiar with Gantt charts, let us say that the scheduling machine assigns shop orders to machine tools in just the same decision-making manner as is done in the shop -- only on a much faster time scale. The machine is then able to deduce logically what is most likely to be happening in the shop for each hour during the next few weeks.

The scheduling machine is first loaded from the shop-order status tape, which gives an up-to-date picture of the status of each shop order. The machine then starts working its way into the future, hour by hour. When a machine tool is available, the scheduling machine scans through all waiting shop orders and picks the "one" with the highest priority that is slated to go on that type of machine tool. At any desired time, the machine can stop working its way into the future and total up the number of shop orders waiting in each department,

to give a picture of scheduled versus available hours.

When a future bottleneck becomes apparent, the production controller has several choices, in order to smooth out the peaks and valleys: changing priorities to move some jobs faster, overtime work, sending certain jobs outside on subcontract well in advance of when the bottleneck would occur, and so on. By "playing" with the schedule in this way, it is believed that he can derive a satisfactory schedule for the next week or two. Also, he can get a rough idea of the future by letting the machine run out a month or two in advance. A rough estimate of the time scale is 15 min. machine time for 40 hr. shop time.

Therefore, two brief (and, it is hoped important) statements can be made about the contribution of electronics to the scheduling problem: The electronic machine helps the production controller to include more of the important variables into his decision-making process, instead of using simplifications and "rules of thumb". Also, the scheduling machine allows the production controller to see the consequences of several alternative decisions, and to choose the decision with the better consequences. Bottlenecks and valleys can be foreseen and corrective action started in time to do some good. We feel confident that a better utilization of production facilities will be realized from such a system, with resultant savings even greater than those obtained from mechanization of clerical operations.

Possible Cost Reductions

The question then arises -- what is the magnitude of cost reductions that an electronic system might produce? Educated estimates only are available so far. At the company studied, it is estimated that the functions of about 14 of the 29 people now in production control could be handled by machine. This direct saving from salaries and overhead would amount to some $175,000 or more, in 2 1/2 years. Since the company's product output in 2 1/2 years would be in the neighborhood of (and this is an estimate based on the number of direct labor employees) $12,000,000, even a 3 per cent increase in output from the reduction of bottlenecks and more optimum scheduling and loading would mean a saving of about $360,000, or about twice as much as the clerical savings. The two savings total some $535,000, in 2 1/2 years, and the cost of the equipment is estimated to be between $250,000 and $300,000.

Conclusion

This paper gives an idea of how an electronic system could take over many of the routine clerical operations in production control, and to assist in some of the nonroutine operations. However, space does not permit a discussion of some of the more interesting issues such as how a particular company can determine whether or not electronics would be of interest to the management. Nor has it been

possible to consider the likely and very important reactions of employees, unions, supervisors, and top management to the idea of such a system, or the possible changes in a company's way of doing business. These are questions requiring further investigation as actual applications are made.

TAC -- a Three-Address Computer

TAC is a hypothetical computer developed for use in MIT's special summer program in Digital Computers: Business Applications, August, 1954. A compiler-interpreter program for MIT's Whirlwind I permits Whirlwind to simulate TAC.

TAC Definitions

A character is any of the following letters, digits, superscripts, symbols, or machine functions, comprising the complete vocabulary of the MIT Flexowriters (augmented by ∅ and ¢).

T C ¢ ⁻ /) - + . I 0 2 4 6 8 $^{0\ 2\ 4\ 6\ 8}$ a b...z

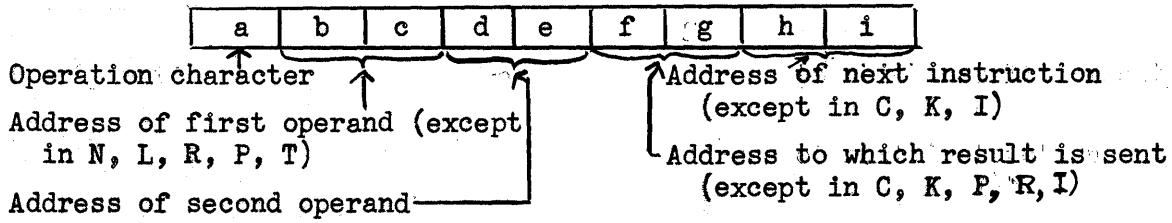
R B H ∅ : (_ = , | S 1 3 5 7 9 $^{1\ 3\ 5\ 7\ 9}$ A B...Z

(where R = return carriage, T = tab (stops every 10th space), B = back space, C = color shift (starts black), H = stop, I = ignore, S = space)

A word consists of nine characters, which are identified from left to right by the letters a b c d e f g h i or by the digits 1 to 9.

A number is a word in which the first character, a, is -, +, or 0 and each of the remaining eight characters bcdefghi is one of nine digits 0, 1, 2,...,9, forming an 8-digit decimal integer.

An instruction is a word in which the first character, a, represents one of the fifteen operations listed below and the remaining characters, paired, form four "addresses". The scheme is shown below:



An address normally refers to one of 110 registers which are numbered 00, 01, ..., 99, x0, x1, ..., x9. However, there are certain exceptions to this:

(1) The bc address associated with operations N, L, R, P and T is omitted, and the characters are used as a continuation of the operation designation (see e.g. N+, N- below);
(2) The fg address for operation P is omitted and the characters are used to continue operation specification;
(3) The registers x1 and x2 can be treated as a single register for certain purposes and this register is denoted by the address xx. For operations A, S, M, D or N, the contents of xx will be treated as a single 16-digit number, provided that x1 and x2 both contain 8-digit numbers having the same sign. Thus if x1 contains +12345678 and x2 contains +98765432, then xx will be deemed to contain +1234567898765432.

We note that address x0 always contains exactly zero. A post-mortem (see below) will always occur if any instruction contains an address which is not legitimate accoding to the above rules.

A post-mortem is performed automatically whenever TAC encounters any impossible instruction (in particular, any word not starting with one of 15 legitimate operation code letters, or containing an illegitimate address, or violating the special conditions listed on pages 2&3). The post-mortem consists of printing the location of the illegitimate instruction itself, the contents of the registers it refers to, a sequence table listing the locations of the instructions performed just prior to the post-mortem, and an altered-word table listing the contents of all of the registers whose contents have been altered during the program.

The time required to perform a given instruction may be calculated from the following facts: TAC storage consists of a magnetic drum with 4 groups of 25 words (0-24, 25-49, 50-74, 75-99) revolving at 40 revolutions per second (1 millisecond per word); access to the x registers is one word time (no waiting time required); and TAC always follows the control sequence: it acquires the instruction, acquires the operand(s*), performs the required operation (in the number of word times indicated in the following list), and stores the result(s*) (omitting any storage accesses where none is required, as with the bc address of N, L, R, P, or T).

*if two references to the drum are required to acquire operands or store results, TAC searches for both simultaneously, so that the time required is the longer of the two access times computed independently.

Symbols Used

abcdefghi represents the nine characters of a TAC word.

C( ) represents "contents of". Thus C(bc) represents "the contents of the register whose address is bc" (where b and c, the second and third characters of the instruction, must each be either one of the decimal digits or the letter x).

→ represents "becomes the new contents of". Consequently: C(bc)+C(de)→fg should be read as "the contents of bc plus the contents of de becomes the new contents of register fg, replacing whatever was in fg but not changing what is in bc or de.

xx represents TAC's double-length register made by pairing x1 and x2.

> represents "is greater than". < represents "is less than".

10^c (read as "ten to the c") represents a one followed by c zeros. (for example $10^5 = 100{,}000$).

TAC Instruction Code

Name	Code Letter	Word-times per operation	Function	Post-Mortem will occur
Read	R	5/char.	Read enough characters from punched tape to fill the positions numbered b thru c in register de, without changing the other digits of C(de). (ignores deletions, deals properly with upper and lower case, but reads all other characters including back spaces, underlines, tabs, and carriage returns explicitly, and ¢ and £ are each read as 3 characters).	If there is not enough tape or if tape contains illegal characters, or if b=0, or if c=0 or if b>c or if de is not a legitimate address,
Print	P	10/char.	Print the characters in the positions numbered b thru c in register de, preceded by the character f and followed by g.	unless b=1, 2,..., 9 and c=1, 2,..., 9 and de is a legitimate address. (NOT XX) or if b>c.
Tape Read	Tr	30/block	read 10 words from block de (or from next consecutive block if de=00) on tape unit c into registers fg, fg+1, ..., fg+9	unless b=w or r " c=1, 2, 3, 4 and " fg=00, 01,..., 90 or x0 " de=00, 01,..., or 40 if de=00 & block 40 has been just read or written.
Tape Write	Tw		write 10 words onto block de (or onto next consecutive block if de=00) on unit c from registers fg, fg+1,... fg+9	
Input	I	5/char.	Without altering the present contents of storage, start reading a punched tape containing a TAC program in conventional form, ending by taking the next instruction from the address preceding "start" at the end of the tape.	
Halt	H		Stop computing. Start at hi only if the start button is depressed.	

Add	A	2	C(bc)+C(de)→fg	1.* if the magnitude of the result exceeds 99,999,999 (or 9,999,999,999,999,999 if fg=xx) 2. if bc=xx or de=xx or fg=xx and (bc) and (de) are not both numbers. 3. if the column-by-column addition or subtraction involves any non-digit N in any arithmetic operation other than N+0=N, N-0=N, or N-N=0. (see next page for details)
Subtract	S	2	C(bc)-C(de)→fg	
Multiply	M	10	C(bc)xC(de)→fg	1.* if magnitude of result exceeds 99,999,999 (or 9,999,999,999,999,999 if fg=xx) 2a. in M and D only, unless C(bc) and C(de) are both numbers. 2b. in N only, unless b=+ or -, c=decimal digit, and C(de)=a number. 3. in D only, if de=xx
Divide	D	25	C(bc)÷C(de)→fg (quotient rounded)	
Numerical Shift Left	N+	2	$C(de)\times 10^{c}\rightarrow fg$	
Shift Right	N-	2	$C(de)\div 10^{c}\rightarrow fg$	
Compare Numerically	C	2	Take next instruction from: fg if C(bc) > C(de), hi if C(bc) < C(de), or next register consecutively if C(bc)=C(de).	* if C(bc) and C(de) are not both numbers.
Logical Shift	L	2	Shift C(de) and C(fg) cyclically c places (left if b is +, right if b is -). ←b=+ b=- → de ↔ fg	unless b=+ or -; c=0, 1, ..., or 9; and de and fg are legitimate addresses. or if bc, de, or fg=xx.
Compare Logically	K	2	Take next instruction from fg if C(bc) $\succ$ C(de), hi if C(bc) $\prec$ C(de), or next register consecutively if C(bc) is identical with C(de), where the symbol $\succ$ is defined as follows: Compare the characters of C(bc) with those of C(de) column-by-column from the left end until identity is established or until one character is found to be to the right of the other in the list below,** in which case the word containing said character is said to be greater than the other logically.	*if bc, de, or fg=xx.
Extract	E	2	In those columns, and only those, in which C(de) has odd characters (listed in the lower line of the two lines below**), replace the characters of C(fg) by the characters occupying corresponding columns in C(bc), without altering the other characters in C(fg).	*if bc, de, or fg=xx

** small end { even: T C ¢ ¯ /) - + . I 0 2 4 6 8 ° 2 4 6 8 a b ... z } large end
{ odd: R B H $: (= , | S 1 3 5 7 9 1 3 5 7 9 A B ... Z }

1. Register x0 contains 000000000. If an instruction attempts to put other information into it, the information is lost. No Post-Mortem occurs.
2. The next instruction is taken from hi (unless otherwise specified in operations C and K) except in I.
3. A Post-Mortem always occurs if hi is not a legitimate address (except in I, and in H if not restarted).

* A Post-Mortem occurs on A, S, M, D, C, K, and E unless bc, de, and fg are all legitimate addresses.

Summary of Specifications for
SAC - a Single-Address Computer

SAC is the 1954 version of the Summer Session Computer of 1953, which was developed for summer classes. SAC is the Summer Session Computer without floating-point numbers but with auxiliary magnetic tapes for a larger amount of storage.

SAC Definitions

A word is either an instruction or an integer written to be stored in one register. Each register holds 27 binary digits and a sign.

An integer is composed of 27 binary digits and a sign. Hence its magnitude must be less than $2^{27} = 134,217,728$ which is somewhat greater than 10^8. Hence SAC integers may be thought of as being 8 decimal digits long. They are written as a + or a - sign (the + may be omitted) and 1 to 8 decimal digits, followed by carriage returns or tabs.

An instruction is made up of an operation section and an address section, and possibly a "counter letter" to add the value of a counter to the value of the address before execution. Instructions are followed by tabs or carriage returns.

An operation is indicated by a code of three lower-case letters as given on pages 3 and 4.

An address may be either absolute or symbolic. An absolute address is any integer from 0 to 298 (since SAC has 299 registers). Using these integers, words are assigned to registers using the address followed by a vertical bar:

```
102| add 15
```

A symbolic address is a single lower-case letter (except o or l) followed by one to three decimal digits. It is used to "tag" registers which are referred to by the program and are written:

```
ccf b3
```

The register referred to is tagged by the address and a comma:

```
b3, +750
```

To correct a program and assign a new word to a previously used register, the tag is written with a vertical bar instead of a comma:

```
b3| +700
```

If the register to be changed has no tag, a count from the nearest tag may be added to that tag and written with a vertical bar:

```
b3+5| +2
```

In either case, words from this point on (after the correction) will overwrite the registers following b3 or b3+5. Counter letters (a, b, c, d, e, f, or g) do not alter the actual instructions in the store but rather cause the instruction to refer to a register with an address which is the sum of the address section of the instruction and the value of the counter called for at the time of execution of the instruction.

A post-mortem may be of two types, conversion or computation. A conversion post-mortem occurs when the program which changes the SAC program into Whirlwind language finds a gross error on the tape. Such errors may be 1) unassigned symbolic address, 2) undefined instruction, 3) duplicate symbolic address, 4) absolute address too large, 5) program longer than 289 words plus the 10 fixed constants, 6) integer too large, 7) number in symbolic address too large, and 8) no counter letter on operations rst, jii, jic, inc, dec, and cii. Computation post-mortems occur during the execution of the program and are defined for each instruction. See pages 3, 4, and 5.

SAC Equipment

SAC is intended to be representative of the general field of single-address digital computers. It doesn't exist as a configuration of electron tubes and wire, though. It is simulated through compiler-interpreter programs on M.I.T.'s Whirlwind I computer. It has a speed of about 3000 operations per second. SAC has 289 registers to hold integers or instructions and 10 registers with fixed constants. Input is by means of tape prepared on the Flexowriter tape perforating equipment and read through a Photoelectric Tape Reader. Output is provided by three different devices. Through the oscilloscope and the camera, curves may by plotted point by point and photographed for permanent record. Information may be typed in two ways, either on a typewriter directly connected to the machine or on special magnetic tape for typing at a later time (recording is done at a much higher speed than direct typing).

The whole computer centers around the arithmetic element, which is made up of an accumulator twice as long as a register, capable of holding 54 binary digits (which means it may hold an integer whose maximum value is about 18×10^{15}), and the remainder register to hold the remainder after a dhr instruction. SAC has an auxiliary store composed of four magnetic tape units, each capable of storing 990 words. They may be used only through the SAC instructions mts, mtr, and mtw.

Data may be placed in the machine in two ways. The prepared program tape may be read into the machine by placing the machine in the loading mode by pressing the READ-IN button, or by using the rip instruction. Complete words, terminated by tabs or carriage returns, may be read in using the rin instruction. Both instructions read Flexowriter tape in the PETR.

A program is prepared on the Flexowriter typewriter which punches a tape as the typewriter operates. There are four parts to the program:

1) the title line, written as f2s 198-4-- - --- and some identifying information. The first set of holes on the actual tape must correspond to an f. The title line is followed by a carriage return.

2) the actual words of the program, each terminated by tabs or carriage returns, using symbolic or absolute addresses as desired.

3) tags on the words as needed, or absolute addresses when desired.

4) an order telling the computer at which instruction the program is to begin being executed, using either a7| start if a tag has been assigned to the word to be started at, or a2+6| start if the word has no tag but one is near, or an absolute address 27| start.

INSTRUCTION CODE OF THE SINGLE ADDRESS COMPUTER (SAC)

INTRUC.	MEANING	DEFINITION	TIME (ms.)	POST-MORTEM*if
ccf al b	copy contents from	$C(al+i_b) \Rightarrow AC$	0.1	14
cci al b	copy contents into	$C(AC) \rightarrow al+i_b$	0.1	2, 5, 9
cnf al b	copy negative from	$-C(al+i_b) \Rightarrow AC$	0.1	14, 15
cmf al b	copy magnitude from	$\lvert C(al+i_b) \rvert \dashrightarrow AC$	0.1	14, 15
cri al b	copy remainder into	$C(RR) \rightarrow al+i_b$	0.1	5
xch al b	exchange	$C(AC) \rightarrow al+i_b$, $C(al+i_b) \rightarrow AC$	0.1	2, 5, 9, 14
add al b	add+	$C(AC) + C(al+i_b) \rightarrow AC$	0.1	1, 3, 4, 9, 14
sub al b	subtract+	$C(AC) - C(al+i_b) \rightarrow AC$	0.1	1, 4, 9, 13, 14
mby al b	multiply by	$C(AC) \times C(al+i_b) \rightarrow AC$	1.5++	1, 12
dby al b	divide by	divide C(AC) by $C(al+i_b)$, rounded quotient $\Rightarrow$ AC	1.5++	11, 12
dhr al b	divide holding remainder	divide C(AC) by $C(al+i_b)$, quotient $\Rightarrow$ AC, remainder $\rightarrow$ RR	1.5++	11, 12
jmp al b	jump	take next instruction from $al+i_b$	0.1	17
jip al b	jump if positive	ditto, if $C(AC) > 0$	0.1	10, 17
jin al b	jump if negative	ditto, if $C(AC) < 0$	0.1	10, 17
jiz al b	jump if zero	ditto, if $C(AC) = 0$	0.1	10, 17
jir al b	jump if remainder	ditto, if $C(RR) \neq 0$	0.1	17
jix al b	jump if excess	ditto, if $C(AC) \geqslant 2^{27}$	0.1	10, 17
sra al b	set return address**	replace address section of $C(al+i_b)$ with 1 + the address of the register containing the most recent jump or conditional jump which took effect	0.1	5, 16
caf al b	copy address from**	address section only (as an integer) of $C(al+i_b) \rightarrow AC$	0.1	5, 16
cai al b	copy address into**	address section of $C(AC) \Rightarrow al+i_b$ as address section	0.1	5, 7, 9, 16
rst m b	reset (counter b)	set $i_b = 0$, $n_b = m$	0.1	19
jii al b	jump if incomplete	increase i_b by 1, then jump to al if $i_b < n_b$	0.1	17, 18
jic al b	jump if complete	increase i_b by 1, then jump to al if $i_b \geqslant n_b$	0.1	17, 18
inc m b	increase (counter b)	increase both i_b and n_b by m	0.1	18, 19
dec m b	decrease (counter b)	decrease both i_b and n_b by m	0.1	18, 19
cii al b	copy index into	i_b as an integer $\Rightarrow$ al	0.1	5

INTRUC.	MEANING	DEFINITION	TIME (ms.)	POST-MORTEM*if
pat al b	plot al	plot a point on the scope at x = C(al+i_b) & y = C(AC) (see drawing on page 4)	1	6, 12
frc 0	frame (scope) camera	move the next film frame into place and open the camera shutter if it was closed	500	
ric 0	read in character	read the next char. via the PETR into AC as a positive integer ≤ 77	100	(Comp. stops if no tape in PETR)
rin 0	read in numerically	read the next complete integer via the PETR into AC	400	1 (also see ric above)
rip 0	read in program	read in program via paper tape in PETR, storing and starting as directed by new program		22 (also see ric above)
mts mno b	magnetic tape search	search magnetic tape unit m for block no and stop at beginning of block. If no=00, select next block in order.	+++	23, 24
mtr al b	magnetic tape read	read from most recently selected magnetic tape unit and block into registers al+i_b, al+i_b+1,...,al+i_b+9	+++	25, 27
mtw al b	magnetic tape write	write on most recently selected magnetic tape unit and block from registers al+i_b, al+i_b+1,...,al+i_b+9	+++	26, 27
tyc m tyc 100+m	type character	record on delayed printer (m), or on direct printer (100+m), the Flexo. char. specified by the integer m	‡ 15 100	20
tyn m tyn 100+m	type numerical value	record on delayed printer (m), or on direct printer (100+m), C(AC) as specified by m (see table page 4)	‡ 30 400	2, 10, 21
stp 0	stop	stop the computation		

*The programming mistakes which result in a post-mortem are listed on the next page. A post-mortem results while performing an instruction if any of the programming mistakes listed with that instruction are made. A post-mortem will always occur if (al+i_b) $\geq$300 or if (al+i_b) $<$ 0.

**When executing this instruction, a counter letter, if any, is not considered part of the address section of the instruction in register al+i_b.

†An integer may be added to an instruction or vice-versa, but an integer may be subtracted only <u>from</u> an instruction. Two instructions with identical operations sections may be subtracted to get an integer.

††A multiplication or division instruction in which the register referred to (register al+i_b) is 290+p where $0 \leq p \leq 8$ (so that the resulting operation is a decimal shift) takes 0.1 (p+1) milliseconds.

†††Time for magnetic tape instructions: 10 ms. per block read or written, plus 10 ms. each time tape stops. Searching time is the same as the time to read all intervening blocks, but computer operation may occur simultaneously.

‡Longer time applies to direct printing, shorter time to delayed printing (via special magnetic tape).

PROGRAMMING MISTAKES which cause a POST-MORTEM

1. Result is an integer of magnitude $\geq 2^{54}_{27}$
2. Result is an integer of magnitude $\geq 2^{27}$
3. C(AC) and C(al+i_b) are both instructions
4. Result is instruction with address > 499
5. al+$i_b \geq 290$ or al+$i_b = 0$
6. $|C(AC)| \geq 1024$ or $|C(al+i_b)| \geq 1024$
7. C(AC) is a positive integer > 499 (or instruction with address < 300)
8. C(AC) is an instruction
9. C(AC) is undefined
10. C(AC) is not an integer
11. C(al+i_b) = 0
12. C(AC) and C(al+i_b) are not both integers
13. C(al+i_b) is not an integer, or instruction with same operation section as C(AC)
14. C(al+i_b) is undefined
15. C(al+i_b) is an instruction
16. C(al+i_b) is not an instruction
17. C(al+i_b) is not an instruction and the jump takes effect (the Post-Mortem will occur after the jump is executed)
18. Resulting magnitude of $i_b \geq 512$
19. $m > 499$
20. $m > 77$ or m corresponds to an illegal Flexo character
21. m=10 or m=20 or $m \geq 30$
22. First character on new tape is not f. (Will cause WWI alarm) (Reader stops after last t in start. If new program follows, next character must be f.)
23. $m \neq 1, 2, 3,$ or 4
24. no = 00 and block 99 has just been used
25. al+$i_b \geq 281$ or al+i_b = 0
26. al+$i_b \geq 290$
27. If tape is positioned after block 99

Contents of special registers

C(0) = +0

C(290) = $+10^0_1$ = +1
C(291) = $+10^1_2$ = +10
C(292) = $+10^2_3$ = +100
C(293) = $+10^3$ = +1000
C(294) = $+10^4$ = +10,000
C(295) = $+10^5_6$ = 100,000
C(296) = $+10^6_7$ = 1,000,000
C(297) = $+10^7_8$ = 10,000,000
C(298) = $+10^8$ = 100,000,000
C(299) = undefined

DEFINITIONS OF SYMBOLS

$\longrightarrow$	becomes the new contents of
AC	Accumulator
C(al)	Contents of register al. al represents any floating address, i.e., any letter except o or l, followed by any non-negative decimal integer < 1000.
C(al+i_b)	Contents of the register whose address is obtained by adding to al the value of i_b.
i_b	The index associated with counter b, where b represents any of the 7 counters a, b, c, d, e, f or g. Except for the 6 instructions rst, jii, jic, inc, dec, cii, a counter letter need not be specified all.
n_b	The criterion associated with counter b.
RR	Remainder Register, which holds the remainder after dhr and is not changed by any other instruction.
PETR	Photo-electric Tape Reader into which is inserted a punched Flexo tape to be read in under control of the computer.

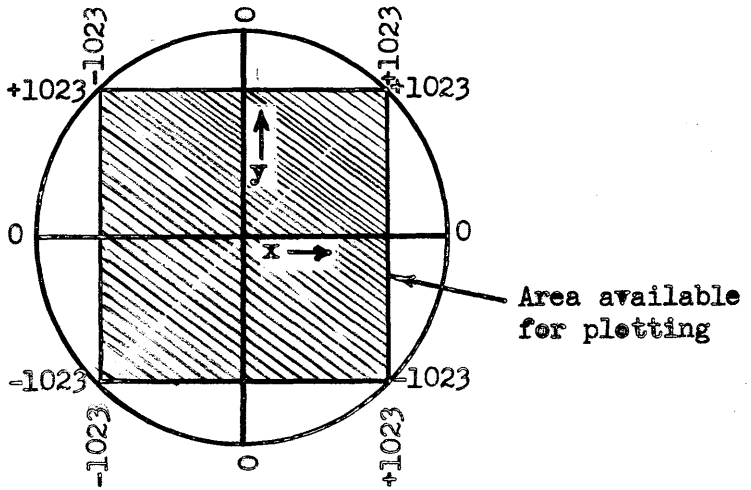
Tabulation of m values for use with tyn

m	initial zeros	no. of digits printed = d	total space pos.	total space neg.	zero prints as	Post-Mortem if
0	ignored	$1 \leq d \leq 9$	d	d + 1	0	
1-9	printed	d = m	m	m + 1	m 0's	$\|C(AC)\| \geq 10^m$
11-19	spaced over	$1 \leq d \leq (m-10)$	m-10	m - 9	see examples	$\|C(AC)\| \geq 10^{m-10}$

Examples	C(AC) = 1234	C(AC) = -789	C(AC) = 0	Delayed/Direct
tyn0	1234	-789	0	Delayed
tyn103	Post-Mortem	-789	000	Direct
tyn5	01234	-00789	00000	Delayed
tyn116	**1234	**-789	*****0	Direct

* represents space on printed copy

Calibration of Scope Face for pat Instruction



A Selected Bibliography of Material Relevant to Business Applications of Computers

PROCEEDINGS OF CONFERENCES AND SYMPOSIA

Joint AIEE-IRE-ACM Computer Conference Reports

(Available from the Association for Computing Machinery, 2 East 63rd St., New York, N.Y.)

1. Trends in Computers: Automatic Control and Data Processing (Western Computer Conference, Los Angeles, Calif., Feb. 11-12, 1954, $3.00)

   Keynote and Luncheon Addresses
   a. Will Electronic Principles Make Possible a Business Revolution W. W. McDowell, p.9
   b. Trends in Electronic Business Data Systems Development, Dean E. Wooldridge, p. 16

   Session II-Data Processing Systems
   c. The Automatic Handling of Business Data, Oliver Whitby , p. 75
   d. Introduction, Richard G. Canning, p. 80
   e. Ready-to-Wear Unit Control Procedure, S. J. Shaffer, p. 82
   f. Unit Control Systems Engineering, Raymond Davis, p. 89
   g. A Solution for Automatic Unit Control, Harry D. Huskey, p. 96
   h. The System in Operation, Myron J. Mendelson, p. 98

   Session IV-Data Processing Equipment
   i. The IBM Magnetic Drum Calculator Type 650-Engineering and Design Considerations, E. S. Hughes, Jr., p. 140
   j. Design Features of Remington Rand Speed Tally, John L. Hill, p. 155
   k. Production Control With the Elecom 125, Norman Grieser, p. 163
   l. A Centralized Data Processing System, Jerome J. Dover, p. 172
   m. A Merchandise Control System, William L. Martin, p. 184

   .....and 10 other articles

2. Proceedings of the Western Computer Conference (Los Angeles, Calif., Feb. 4-6, 1953, $3.50)

   Session I
   a. Commercial Applications -- The Implication of Census Experience J. L. McPherson, p. 49
   b. Payroll Accounting with Elecom 120 Computer, R. F. Shaw, p. 54
   c. Automatic Data Processing in Larger Manufacturing Plants M. E. Salveson and R. G. Canning, p. 65
   d. Requirements of the Bureau of Old-Age and Survivors Insurance for Electronic Data Processing Equipment, E. E. Stickell, p. 74
   e. The Processing of Information-Containing Documents, G. W. Brown and L. N. Ridenour, p. 80
   .....and 18 other articles

3. Proceedings of the Eastern Joint Computer Conference (Washington, D.C., Dec. 8-10, 1953, $3.00)

   Use of Electronic Data-Processing Systems in the Life Insurance Business, M. E. Davis, p. 11

   .....and 23 other articles

PROCEEDINGS OF CONFERENCES AND SYMPOSIA - Cont.

4. Review of Input and Output Equipment Used in Computing Systems (New York, N.Y., Dec. 10-12, 1952, $4.00) 27 articles

5. Review of Electronic Digital Computers (Philadelphia, Pa., Dec. 10-12, 1951, $3.50)

Proceedings of the Association for Computing Machinery

(Available from the Association for Computing Machinery, 2 E. 63rd St, New York, N.Y.)

6. Meeting at Toronto, Ont., Sept. 8-10, 1952 - 35 articles ($3.00)

7. Meeting at Pittsburgh, Pa., May 2 and 3, 1952 - 41 articles ($4.00)

See also Journal of the ACM, #23 below

American Management Association

(Available from American Management Assn., 330 W. 42nd St., New York 36, N.Y.)

8. A.M.A. Special Conference - Integrating the Office for Electronics (Convention Workbook, Feb. 25-26, 1954, New York, N.Y., Not for sale) Contains good bibliography of references.

9. A New Approach to Office Mechanization: Integrated Data Processing Through Common Language Machines - The U.S. Steel Corp. Program $2.50 to non-members

Navy Mathematical Computing Advisory Panel Meetings

(Published by the Office of Naval Research, Dept. of the Navy, Washington, D. C.)

10. Symposium on Managerial Aspects of Digital Computer Installations 30 March 1953

11. A Symposium on Commercially Available General-Purpose Electronic Digital Computers of Moderate Price - 14 May 1952

Life Office Management Association

12. Electronics Seminar, Papers presented at Spring Conference, Swampscott, Mass., May 25, 1953 (Not for sale)

Midwest Research Institute (Kansas City, Mo.)

13. A Symposium on Industrial Applications of Automatic Computing Equipment, Jan., 1953

Railway Systems and Procedures Assn.

14. Proceedings 1954 Spring Meeting: Inventory Management and Data Processing (Chicago, Ill., April 20-22, 1954, $4.50, Copies obtainable from: Mr. J. W. Milliken, Secretary-Treasurer, Railway Systems and Procedures Assn., P. O. Box 514, New York 8, N.Y.)

Manchester University Computer

15. Inaugural Conference, July, 1951

SURVEYS

16. A Survey of Automatic Digital Computers - Office of Naval Research-1953 (Available from US Dept. of Commerce, Office of Technical Services, Washington 25, D. C., $2.00)

17. Electronic Digital Computer Survey - Jan. 1953
(Vitro Corp. of America, 233 Broadway, New York 7, N.Y.)

GLOSSARIES

18. Report to the Association for Computing Machinery: First Glossary of Programming Terminology - June 1954
(Available from the ACM, 2 East 63rd St. New York 36, N.Y., $.25)

19. Standards on Electronic Computers: Definitions of Terms, 1950
(Available from IRE, 1 East 79 St., New York 21, N.Y., $.75)

REPORTS

20. Electronics - New Horizon in Retailing
Research Report prepared by a group of students at the Harvard Grad. School of Business Administration
(Available from AER Associates, 6450 Cecil Ave., Clayton 5, Mo.-$10.00)

21. Electronic Business Machines - A New Tool for Management
Report of a group of Harvard Business School students for second-year course in Manufacturing

22. Report of Committee on New Recording Means and Computing Devices
Society of Actuaries, Sept., 1952

COMPUTER JOURNALS

23. Journal of the Association for Computing Machinery (Published Quarterly since Jan. 1954, subscriptions $5.00 for members; $10.00 for non-members; membership, including subscription, $6.00)

24. Computers and Automation (Edmund C. Berkeley and Associates, 36 West 11th St., New York 11, N.Y., $4.50/yr., published periodically; now to be published monthly)

BOOKS

25. Berkely, Edmund C., Giant Brains or Machines that Think
John Wiley and Sons Inc., 440 Fourth Ave., New York, N.Y. (1949)

26. Booth and Booth, Automatic Digital Calculators
Academic Press Inc., 125 East 23rd St., New York 10, N.Y. (1953)

27. Bowden, B. V., Faster Than Thought
Pitman Publishing Corp., 2 West 45 St., New York, N.Y. (1953)

28. Diebold, John, Automation
D Van Nostrand Co. Inc., New York, N.Y. (1952)

29. Engineering Research Associates, High Speed Computing Devices
McGraw-Hill Book Co., New York, N.Y. (1950)

30. Hartree, Douglas R., Calculating Instruments and Machines
The University of Illinois Press, Urbana, Illinois (1949)

31. Wiener, Norbert, The Human Use of Human Beings
Houghton Miffling Co., Boston (1950)

32. Wilkes, Wheeler and Gill The Preparation of Programs for an Electronic Digital Computer
Addison Wesley Press, Inc., Cambridge 42, Mass. (1951)

ARTICLES FROM JOURNALS AND MAGAZINES

33. Business Week
Tomorrow's Management Aug 15, 1953

Harvard Business Review
34. Electronics Down to Earth, J. A. Higgins and J. S. Glickauf, March-April 1954
35. GE and UNIVAC, R. F. Osborn, July-August 1954

36. Office Management Association Journal
Automatic Calculating Machines and Their Potential Application in the Office, D. R. Hartree, August 1952

37. Journal of the Instutute of Actuaries
Large-Scale Electronic Digital Computing Machines, R. L. Michaelson December 1953

38. Journal of Accountancy
Accountant's Responsibility for Making Punched-Card Installations Successful, Leon E. Vannais Oct. 1949

39. Journal of the Royal Society of Arts
Automatic Calculating Machines, M. V. Wilkes, 14 December 1951

40. Proceedings of the Institute of Radio Engineers, Computer Issue, October 1953
a. Computing Bit by Bit or Digital Computers Made Easy
A. L. Samuel, p. 1223
b. Can Machines Think?, M. V. Wilkes, p. 1230
c. Computers and Automata, C. E. Shannon, p. 1234
d. Electronic Computers and Telephone Switching, W. D. Lewis, p. 1242
e. Fundamentals of Digital Computer Programming, W. H. Thomas, p. 1245
.....and 39 other articles

41. Philosophical Magazine
Programming a Digital Computer to Learn, A. G. Oettinger, Dec. 1952

42. Stores
Retailing with Electronics, Joseph B. Jeming

Fortune
43. Office Robots, January, 1952, p. 82
44. The Automatic Factory, October, 1953, p. 168
45. The Information Theory, December, 1953, p. 136 Francis Bello
46. Push-Button Labor, August, 1954, p. 50

Scientific American
47. Mathematical Machines, H. M. Davis, April 1949, p. 29
48. The Strange Life of Charles Babbage, Philip and Emily Morrison, April 1952, p. 66
49. Computers in Business, L. P. Lessing, January 1954, p. 21
50. Linear Programming, W.W.Cooper and A. Charnes, August 1954, p. 21

51. Time
The Thinking Machine, Jan. 23, 1950, p. 54

COMMERCIAL COMPUTER MATERIAL

International Business Machines Corp., 590 Madison Ave., New York, N. Y.

52. Light on the Future (1953) (General information on computers.)
53. IBM Electronic Data Processing Machine Type 702 Manual of Instruction
54. IBM Type 650 Operating Principles
55. Principles of Operation Type 701 and Associated Equipment (1953)
56. IBM Speedcoding System for the Type 701 Electronic Data Processing Machines (1953)

Also case studies on the 650 and brochures on commercial applications of other IBM machines.

Remington Rand Inc., 315 Fourth Ave., New York 10, N.Y.

57. Programming Univac Fac-tronic Systems - Instruction Manual I (1953) ($18.50)
58. How Univac Predicted the Election for CBS-TV (1952)
59. The A-2 Compiler Systems Operation Manual (1953)
60. The Editing Generator (1952)
61. Univac Short Code (1952)
62. The Programmer - a periodical
63. Catalogue of Courses in Electronic Computers (1953-54)

Computer Research Corp., 3348 W. El Segundo Blvd., Hawthorne, Calif.

64. An Explanation for the Layman of "Electronic Brains", Everett A. Emerson (1953)
65. Comparison of the Card-Programmed Computer with the General-Purpose Computer Model CRC 102-A (1953)
66. Accounting with Electronics, J. S. Warshauer (1953)
67. Sorting and Collating with the CRC 107 or CRC 102A General Purpose Computers

The British Tabulating Machine Co., Ltd., 17 Park Lane, London, W.1.

68. Rambles Through Binland and Electronia, R. Michaelson

MIT PUBLICATIONS

Course Notes

69. Notes on Digital Computers and their Applications, Summer 1953 ($5.00)
70. Digital Computers - Advanced Coding Techniques, Summer 1954 ($1.00)
71. Notes from MIT Summer Course on Operations Research, June 16-July 3, 1953 ($3.50+ .15 postage from Technology Press)

Theses

72. Electronic Digital Machines for High-Speed Information Searching, Philip R. Bagley (1951)
73. Applications of Self-Checking and Self-Correcting Codes to Digital Computers, F. E. Heart (1952)
74. A Survey of Automatic Coding Techniques for Digital Computers John L. Jones, (1954)
75. Department Store Information Processing Techniques B. E. Morriss (1952)
76. Information Sorting in the Application of Electronic Digital Computers to Business Operations, H. H. Seward (1954)

MIT PUBLICATIONS, Cont.

Digital Computer Laboratory Reports

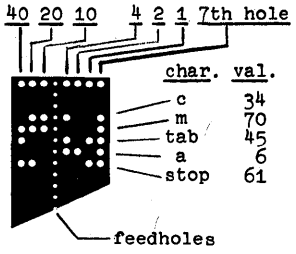
77. The Programmed Synthesis of Digital Computers Within Digital Computers, F. E. Heart (1954)
78. The M.I.T. Systems of Automatic Coding: Comprehensive, Summer Session, and Algebraic, C. W. Adams (1954)
79. Digital Computers As Information Processing Systems, J. W. Forrester (1949)
80. Charles Babbage -- Scientist and Philosopher, Edited by R. R. Rathbone (1952)
81. The Difference Engines of Pehr Georg and Edvard Scheutz, Edited by R. R. Rathbone (1952)
82. Summary Report No. 37 - First Quarter 1954
83. Summary Report No. 36 - Fourth Quarter 1953
84. Summary Report No. 35 - Third Quarter 1953
85. Summary Report No. 34 - Second Quarter 1953

THE FLEXOWRITER CODE

Alphanumerical Sequence

Low. Case	Up. Case	Code Value	Low. Case	Up. Case	Code Value
a	A	6	0	0	76
b	B	62	1	1	25
c	C	34	2	2	17
d	D	22	3	3	7
e	E	2	4	4	13
f	F	32	5	5	23
g	G	64	6	6	33
h	H	50	7	7	27
i	I	14	8	8	3
j	J	26	9	9	66
k	K	36	+	/	15
l	L	44	-	-	35
m	M	70	.	)	21
n	N	30	,	(	31
o	O	60	=	:	11
p	P	54	|	_	5
q	Q	56	space bar		10
r	R	24	carr. ret.		51
s	S	12	tab.		45
t	T	40	up. case		71
u	U	16	low. case		75
v	V	74	nullify		77
w	W	46	stop code		61
x	X	72	back space		43
y	Y	52	color sft.		20
z	Z	42			

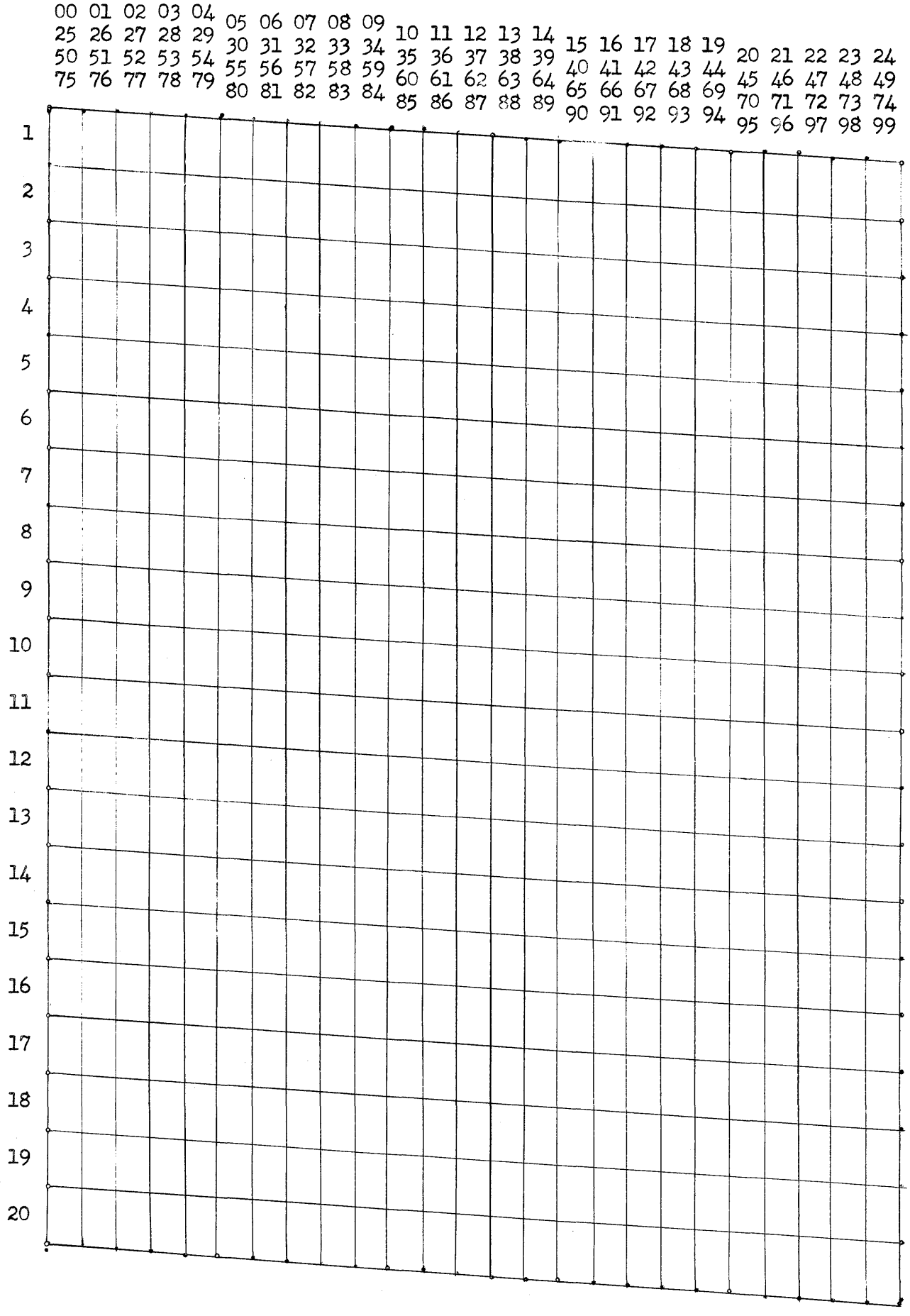
The values of such characters are equal to the sum of the weights appearing at the head of each column in which a hole is punched. The seventh hole, which comprises the right-most column in the sketch, is used only for control purposes. The seventh hole must be punched as part of each character which is to be read in by the computer.

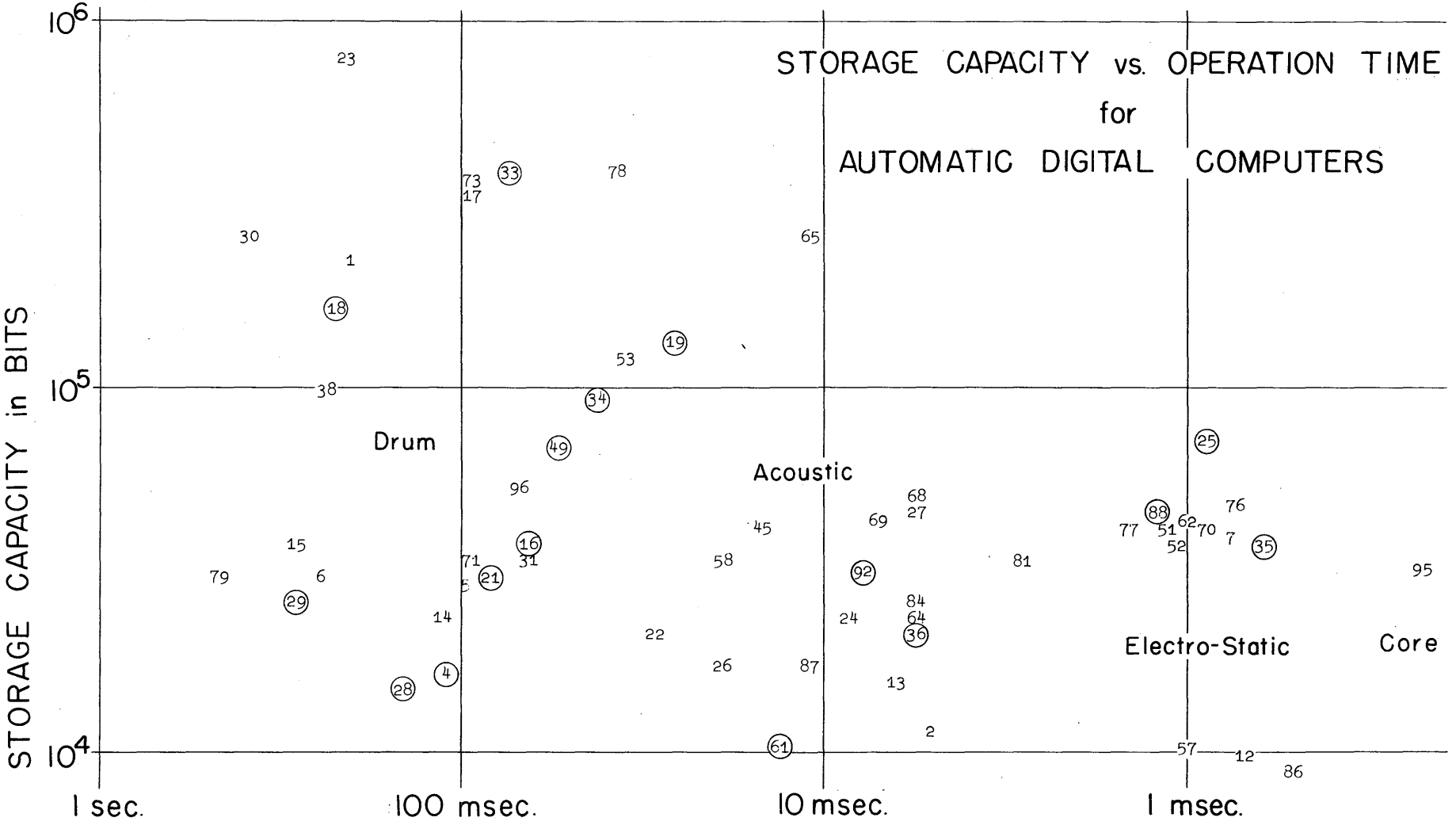


Coded Value Sequence

Code Val.	Low. Case	Up. Case	Code Val.	Low. Case	Up. Case
2	e	E	35	-	-
3	8	8	36	k	K
5	|		40	t	T
6	a	A	42	z	Z
7	3	3	43	back space	
10	space bar		44	l	L
11	=	:	45	tabulation	
12	s	S	46	w	W
13	4	4	50	h	H
14	i	I	51	carr. ret.	
15	+	/	52	y	Y
16	u	U	54	p	P
17	2	2	56	q	Q
20	color shift		60	o	O
21	.	)	61	stop code	
22	d	D	62	b	B
23	5	5	64	g	G
24	r	R	66	9	9
25	1	1	70	m	M
26	j	J	71	up. case	
27	7	7	72	x	X
30	n	N	74	v	V
31	,	(	75	low. case	
32	f	F	76	0	0
33	6	6	77	nullify	
34	c	C			

TAC Time Chart





◯ = commercially available

1	ABC	13	BINAC	21	CRC 107	30	ELECOM 200	51	ILLIAC	68	MSAC	81	RAYDAC
2	ACE	14	Burroughs	22	CSIRO Mark I	31	ELLIOTT-NRDC	52	IAS	69	MOSAIC	84	SEAC
4	APE(R)C	15	CADAC 102	23	CUBA	33	ERA 1101	53	IRSIA-FNRS	70	NAREC	86	SWAC
5	APE(X)C	16	CADAC 102A	24	DYSEAC	34	ERA 1102	57	JOHNNIAC	71	NICHOLAS	87	TAC
6	ARRA	17	CALDIC	25	EDPM 701	35	ERA 1103	58	LEO	73	OARAC	88	TC-1
7	AVIDAC	18	Circle	26	EDSAC I	36	FLAC	61	Manchester	76	ORACLE	92	UNIVAC
12	BESK	19	CE 36-101	27	EDVAC	38	G 2	62	MANIAC	77	ORDVAC	95	Whirlwind I
				28	ELECOM 100	45	Hughes Air	64	MIDAC	78	PERM	96	WISC
				29	ELECOM 120	49	IBM 650	65	MINAC	79	PTERA		