UCRL-11753

UNIVERSITY OF CALIFORNIA

Lawrence Radiation Laboratory
Berkeley, California

AEC Contract No. W-7405-eng-48

FIELD GUIDE TO COMPUTERS, THEIR HABITS & HABITATS

Jim Baker

January 1965

UCRL-11753

Field Guide to Computers, Their Habits & Habitats
Part I: The Nature of the Beast

LRL mathematician Jim Baker, author of the series which begins here, is currently acting head of Berkeley's Mathematics and Computing Group. Baker majored in mathematics at UCLA and Pomona College, later did graduate work at UC. In 1952, he joined the Laboratory to work on applications of computers to experimental and theoretical physics problems. Since 1960, he has assisted Kent Curtis in the administration of the Mathematics and Computing Group. He was named acting group leader in August, 1963, when Curtis left for a year's assignment with the AEC in Washington.

Back in 1940—which is not so very long ago as the history of science is measured—an LRL research scientist with a problem in mathematics on his mind could take his questions to any one of the five computing machines which the UC physics department proudly maintained and made available to the faculty. Of course, one couldn't be sure of getting an electric machine (there were, after all, only two of those in the whole department); most likely, our scientist would settle for a hand-operated model and count himself lucky that he wasn't living back in the days when scientists did their own calculations with pencil, paper, and a "head for figures."

By 1950, this picture had changed but little. The hand-operated machines had, no doubt, been replaced by electric models, and the first generation of electronic digital computers—ENIAC and its relatives—had already appeared on the national scene. But the working scientist at LRL still did most of his calculations with the help of computing machines not very different in principle from the ones he had been using ten years earlier.

The Machines Multiply

By 1960 all this had changed. Today, this Laboratory ranks as one of the greatest concentrations of automatic computing power to be found anywhere on earth. Within the past twelve years, automatic digital computing machines and their associated systems have become central to LRL research programs, and their uses have been extended to fields as diverse as high energy physics and payroll accounting, nuclear device design and library documentation.

This is the first of a series of MAGNET articles which will explore the design and uses of computers, with particular attention to their pertinence to LRL research programs. Among the questions which we shall seek to answer are these:

1. What is an automatic digital computer?
2. How does a computer work?
3. What sorts of problems can you do with a computer?
4. What kinds of computers does the Laboratory have?
5. What will be done with computers in the future?

Definitions

The things we want to talk about in this series are called Automatic Stored-Program Digital Computing Machines. Let us start, then, with a word-by-word analysis of this rather sonorous title.

A *computing machine* is any device which allows us to deduce, from certain numbers that we know, certain other numbers that we want to know.

All computing machines may be classified either as analogue machines or digital machines. *Analogue machines* compute by measuring; *digital machines* compute by counting. An example of an analogue computer is the ordinary slide rule. The operation of the slide rule depends on the principle that says that if you lay two sticks end-to-end, then the distance from the left end of the first stick to the right end of the second stick is the sum of the lengths of the sticks; slide rules work by adding and subtracting distances.

The Abacus

An example of a digital computer is the abacus. This is an ancient oriental device which allows one to compute by moving beads on wires. A typical abacus may have eight vertical wires and seven beads on each wire. The right-most wire is the units wire, the second right-most wire is the tens wire, the third right-most wire is the hundredths wire, and so on; this is a decimal computing machine. On a given wire, the five bottom-most beads are worth one unit apiece, while the two top-most beads are worth five units apiece; this is called the biquinary encoding system. One operates the abacus by moving the beads up and down on their wire. If a bead is pushed up as far as it can go, then it counts its whole value. If it is down as far as it can go, then it counts zero. One essentially operates the abacus by counting.

Manual vs. Automatic

Digital computers may again be divided into two classes: manual and automatic. *Manual digital computers* require the presence and intervention of a human operator at each step of the calculation which is being performed; *automatic digital computers*, on the other hand, can perform a large number of calculations without any human intervention.

An example of a manual digital computing machine is an ordinary electric desk calculator. This machine requires its operator to enter each number by hand and then to press the appropriate button indicating to the machine the operation that it is to perform. An example of an automatic digital computing machine is the IBM type 602A electronic calculator; in this machine, the user indicates the sequence of arithmetic operations to be performed in advance by inserting wires in a plug board. This plug board is then mounted on the machine, input data in the form of tabulating cards are placed in a hopper belonging to the machine, the operator presses the START button, and the machine then proceeds automatically to perform a fairly lengthy series of calculations on the input data without further operator intervention.

Fixed vs. Stored Program

All automatic digital computing machines may again be subdivided into two classes: fixed and stored program.

The fixed program machines employ a fixed sequence of operations for each problem that they do. This sequence of operations is either permanently built into the hardware of the machine (such a machine is called a *single-purpose machine*) or it is specified by a medium such as a plug board or sequence of tabulating cards which the machine itself cannot alter. In a *stored program machine*, on the other hand, the sequence of operations to be performed is specified by "instructions" which are stored in a part of the machine called the memory which the machine itself can alter.

The distinction between a fixed program and stored program automatic computer is much more fuzzy than the distinction between a manual digital computer and an automatic digital computer; *the essence of this distinction is, however, that the sequence of operations in a stored program computer is much more easily altered than the sequence of operations in a fixed program computer, and that it is very much quicker and easier to insert a new sequence of instructions in a stored program computer than it is in a fixed program computer.*

An example of a fixed program automatic digital computer is the IBM 602A Electronic calculator cited above. An ex-

ample of a stored program automatic digital computer is the IBM type 1401 computer.

The sequence of "instructions" which specifies the operations that the computer is to perform is called a *Program*.

Parts of a Computer

Every automatic stored-programmed digital computer may be divided into four functional units. These units are called:

1. *The memory*
2. *The arithmetic unit*
3. *The control unit*
4. *The input-output section*

The *memory* of a digital computer is used to hold data which are being processed, intermediate results, and the instructions that tell the machine which operation to perform next.

The basic information unit in any computer is called the *word*. A word is nothing but a string of numeric digits or alphanumeric characters with an algebraic sign (+ or −) in front of it. Associated with each computer is a definite *word size;* one computer might have a word size of five decimal digits together with an algebraic sign. A typical word in such a computer might be −21376. In another computer, the word size might be 11 binary digits (we will talk about the binary number system later) and an algebraic sign. An example of a word in such a computer would be +10110010-110. A computer whose words contain decimal digits is said to be a *decimal computer*. A computer whose words contain binary digits only is said to be a *binary computer*. A computer whose words contain decimal digits and alphabetic characters is said to be an *alphanumeric computer. The important thing is that associated with each computer is exactly one word size*. For example, the IBM 650 is a decimal computer and its word consists of an algebraic sign and ten decimal digits. The IBM 7094 is a binary computer whose word size is an algebraic sign and 35 binary digits. The IBM 1401 is an alphanumeric computer whose word size is one alphanumeric character.

Computer memories are divided up into cubbyholes — much like postoffice boxes—each one of which is large enough to hold exactly one word. So that we can keep track of the cubbyholes in memory where we are storing particular pieces of data or particular instructions for the computer, we give each cubbyhole a name or label. This label is called the *address* of that cubbyhole or the address of that memory cell. The label is usually a number; so if, for example, we have a machine with a one-thousand word memory, the addresses of cells in that memory would run from 000 to 999.

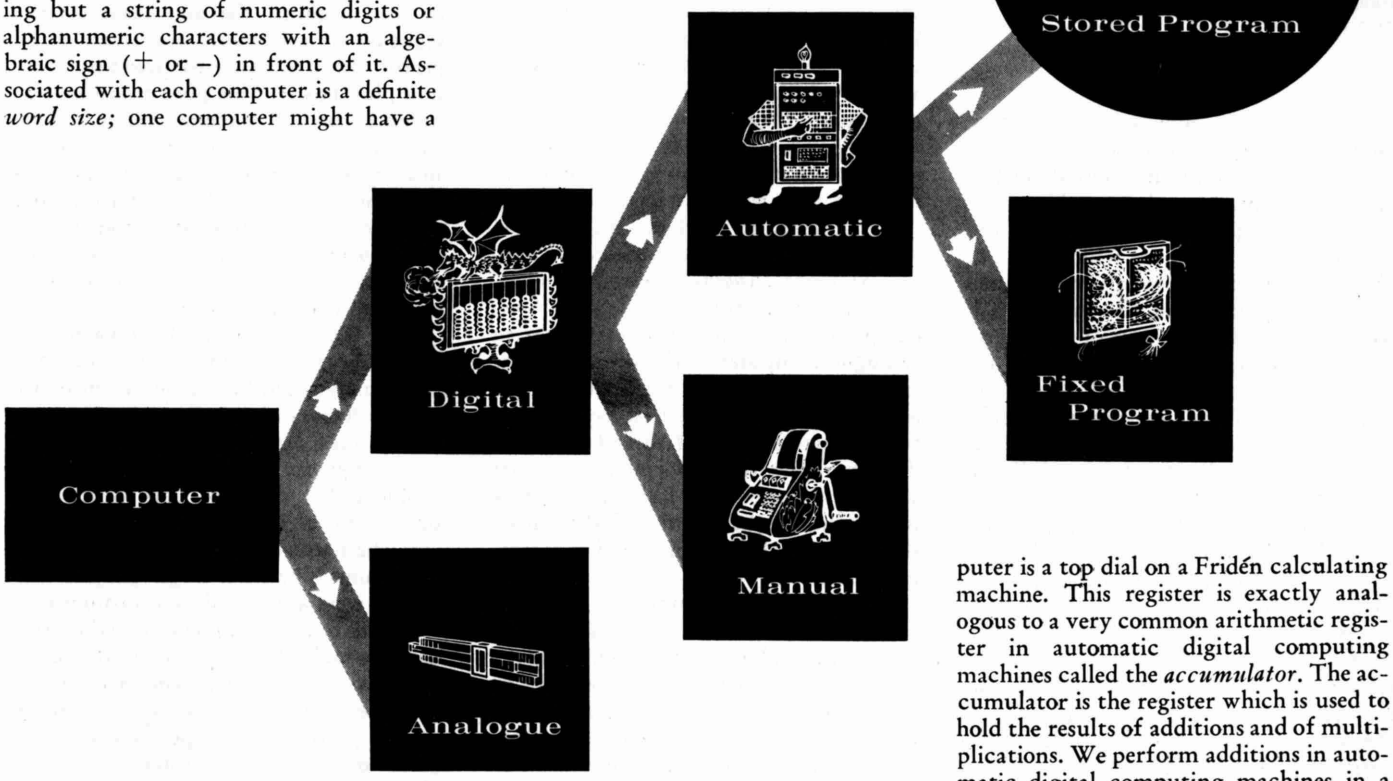
By the *cycle time* of a computer memory, we mean the time that is required to retrieve one word from the memory and be ready to retrieve another word. Hence, if we have a computer whose memory has a cycle time of 10 microseconds, we will need 100 microseconds to retrieve 10 words from that memory.

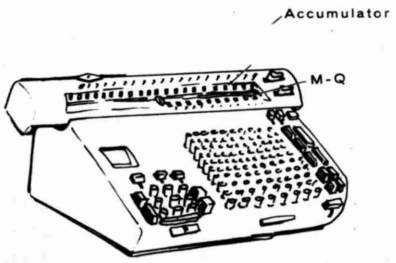
Arithmetic Unit

The *arithmetic unit* (pronounced with the accent on the "met") of a digital computer is the part where all of the work gets done. This unit contains hardware which enables the computer to add, subtract, multiply, divide and perform certain other logical functions. As users, we need not be concerned with the mechanisms which are used to perform these arithmetic operations; the thing that we need to be concerned with is where the results of these operations end up.

The results of arithmetic operations in many digital computers end up in devices which are called *registers*. A register is simply a bin or cell or cubbyhole in the arithmetic unit which is large enough to hold one or more words. A good example of an arithmetic register in a digital computer is a top dial on a Fridén calculating machine. This register is exactly analogous to a very common arithmetic register in automatic digital computing machines called the *accumulator*. The accumulator is the register which is used to hold the results of additions and of multiplications. We perform additions in automatic digital computing machines in a fashion quite similar to the way in which we perform them on an ordinary adding machine (the Fridén, for example); we first set the accumulator to zero, then we place in it a number which up until now has been stored in a cell in memory; we then add to it a second number which was perhaps stored in another cell in memory. At the conclusion of this operation, the sum of these two numbers remains in the accumulator register. Often, to save space, we refer to the accumulator as the *A register* or, starkly, as *A*.

Another arithmetic register which occurs in quite a large number of digital computers is called the *Multiplier-Quotient* register or for short, the "M-Q" register. This register, as its name implies,





is used to hold one of the factors in a multiplication and to hold the result of a division. It corresponds to the second dial from the top on the Fridén.

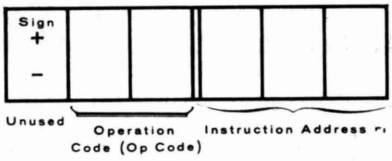
All computers have an arithmetic section, but not all computers have arithmetic registers. In some machines, the IBM 1401 for example, results of arithmetic operations are stored directly in the memory without stopping in an arithmetic register along the way.

The Control Unit

The *control unit* of an automatic digital computer is that portion of the machine that interprets the instructions that the machine is to execute and, in general, tells the machine what to do next. The control unit almost always contains two registers, called the *instruction register* and the *control counter* (IR and CC, for short). The instruction register, which is usually one word long, contains the command which is *currently being executed*. The control counter, which is just large enough to hold one address, holds the address of *the next instruction to be executed*.

It should be clear by now that the instructions or commands which tell the machine which function to perform next are nothing but machine words (which is to say, numbers of a certain length); thus, there is no effective way of distinguishing an instruction word from a data word. In fact, we occasionally inadvertently try to execute as an instruction a data word—this usually leads to absurd results. On the other hand, we often intentionally do arithmetic with instruction words, as we shall see later.

An instruction word, in a digital computer whose word length was algebraic sign and five decimal digits, might look something like this:



The sign in this particular machine is not interpreted as part of the instruction word. The *operation code*, sometimes abbreviated op code, is composed of two decimal digits. This operation code tells the machine what function it is to perform next; for example, an operation code of 01 might tell the machine to add, an operation code of 02 might tell the machine to subtract, and so on. The last three digits of the instruction word in this machine are to be used for an address. We will use the letter "m" to designate this instruction address in future discussions. The *address part of the instruction* tells the machine the location of the data on which it is to perform the operation specified by the operation code. So, for example, the instruction word +01256 might mean to add (operation code 01), the number that is stored in address number 256 to the number that is already in the accumulator, and leave the result in the accumulator.

Some machines have registers called "B registers" or "index registers" in their control sections. These registers are used in a special way to modify the address parts of instructions. We will discuss them in more detail later.

Input-Output Section

The *input-output section* of an automatic computer is a very important one from the viewpoint of the user. It is this section which allows the user to communicate with the computer (this is called input) and the computer to communicate with the user (this is called output).

Input-output may be accomplished through a very wide variety of media. The simplest, least expensive, and slowest input device is a set of *switches* on the console of the computer. By throwing these switches appropriately, the operator may feed information into the computer. While almost every computer has facilities for this type of input, these facilities are seldom used except by engineers who are trying to repair the machine.

The simplest kind of output device is a set of *display lights* on the console of the machine. These lights may display the contents of the various arithmetic and control registers and may also have the capability of displaying the contents of various cells in the memory. Obviously, the machine must be stopped before we can read the contents of the lights; hence. these devices are not frequently used either.

A step ahead of console switches and lights as an input-output device is the *typewriter*. This device may be used for input by its keyboard or for output onto a piece of paper. Because of its slowness, it should not, however, be used for large volumes of either input or output.

Now we come to a class of input-output media which includes *paper tape*, *tabulating cards*, and *magnetic tape*—all of which make input-output a two-or three-step process. Consider, for example, paper tape, which is a very common medium on low-cost computers. If we wish to prepare some data for input to our computer, we may punch these data onto a paper tape using a typewriter-like device such as a Flexowriter or teletype machine. We then take this paper tape containing our data over to our computer and read it in. The important thing here is that the slow-speed human process of punching the paper tape is separated from the relatively high-speed automatic computer processes of reading paper tape and computing. The situation on the output side is very similar. The computer punches out our answers on paper tape at relatively high speed. Since we cannot read this paper tape directly, we take it over to a printing device which looks something like a typewriter, and insert the paper tape into this device and our answers are then printed out on a sheet of paper.

Paper tape and cards are generally used as input-output media on low-or medium-price machines. Magnetic tape, which is very much faster, is used as the basic input-output medium on all high speed computers today.

Part II: Simple-Minded Computer

These days, digital computers can solve calculus problems at about the level of a college sophomore; they can translate from Russian into English at perhaps the college freshman level; they can produce numerical solutions to very difficult mathematical problems.

We have already described the four principal parts of an automatic digital computer. We saw that the computer must perform all its complicated tasks by doing a few relatively simple arithmetic and logical operations at a very high rate of speed.

At this time we are going to give an example which will illustrate all the features and functions described in last month's article. This example computer, called the SMAC (or Simple Minded Automatic Computer), is a realistic machine; it is more complicated than some computers which are presently installed at the Laboratory. If you can understand how SMAC works, then you should be able to understand how almost any digital computer works.

Type of Arithmetic

The SMAC is a *decimal machine* — which is to say that it uses the same number system that you and I use. SMAC's word length is five decimal digits and an algebraic sign. It represents negative numbers the same way that you and I represent negative numbers. For example, in SMAC the number −14 would be represented as −00014. This is an advantage over most hand calculators, which would tend to represent −14 as 9999999986 (as you can easily verify by subtracting 14 from zero on one of them). The SMAC's way of representing numbers is called the *sign and magnitude system.* The hand calculator's way of representing negative numbers is called the *tens complement system.*

Anatomy of SMAC

Memory. SMAC has a one-thousand-word memory. Each word in the memory has an address consisting of three decimal digits; thus, addresses run from 000 up to 999. Each of the one thousand memory cells is, of course, just big enough to contain one word consisting of five decimal digits and an algebraic sign.

Arithmetic Section. SMAC has a very simple arithmetic section indeed. It consists of exactly one arithmetic register: the accumulator. We will often refer to the accumulator as the A register or simply as A. SMAC's accumulator is just one word long — five decimal digits and an algebraic sign. It is the register that will hold the results of all arithmetic operations performed in SMAC.

Control Section. The control section of SMAC contains three registers. The *instruction register,* or IR, is a one-word (five decimal digits and sign) register; it holds the command which is currently being executed. The *control counter,* or CC, is a three-digit register; it contains the address in memory where the next command to be executed is located. The *index register* (which we will call the B register or B) is another three-digit register. It is used to modify the address portions of certain commands. Its use will be illustrated later.

Input-Output Section

SMAC has a paper tape reader for input and a paper tape punch for output. It can read one word in a forward direction from the paper tape which is currently in the tape reader, and then place this word in a memory location specified by the command which is being executed. Similarly, it can punch out onto the paper tape which is currently in the tape-punching mechanism one word from an address designated by the current command.

Paper tapes which are to be read into the computer must first be prepared in the proper form. Sometimes, they are punched on a Flexowriter or Teletype machine away from the computer; sometimes they have been punched by the computer itself in a previous run. When we wish to find out what information is on an output tape that has been punched by the computer, we must take that tape over to a Flexowriter, which then prints the contents of the tape on a piece of ordinary paper.

Instruction Format

Every computer word that is brought from the memory into the instruction register is interpreted by SMAC as an instruction word. The two decimal digits at the extreme left (which we sometimes call the two most significant digits) are interpreted as an *operation code,* or OP code. This operation code tells the machine what function it is to perform next. The three digits at the extreme right of the instruction word are interpreted by the machine as an *address.* This address, in general, gives the machine the location of the data upon which the function specified by the OP code is to be performed. So, for example, if the operation code for addition is 01 and the word +01223 comes into the instruction register, the computer will add the word which is presently stored in memory location 223 to the number which is currently in the accumulator, leaving the sum in the accumulator.

The sign of the instruction word tells the computer what to do with the B register. If the sign of the current instruction word is +, then the B register is ignored. If the sign of the current instruction word is −, then the contents of the B register (a three-digit number) is subtracted from the address part of the instruction word (the three digits on the right) before the instruction is executed. We often refer to the address part of the current instruction as m.

SMAC EXECUTES A COMMAND
(Assume that the execution of a command has just been completed, then start at Step 1.)

1. C(C(CC))→IR
2. 1+ C(CC)→CC
3. If the sign of IR is +, go to Step 5; if the sign of IR is −, go to Step 4.
4. Subtract C(B) from m (the rightmost 3 digits of IR) and replace m by the result.
5. Execute the instruction in IR.
6. Go to Step 1.

Figure 1

Notation

In order to be able to describe more concisely how the computer works, we must make some notational conventions. Let us agree that $C(s)$ is to mean the *contents of s* if s is the name of either *a memory location, an arithmetic register, or a control register;* similarly, $C(m)$ will mean the word that is currently stored in memory location m; $C(A)$ will mean the contents of the accumulator; $C(CC)$ will mean the contents of the control counter (remember that the contents of the control counter is a three-digit number).

We use the arrow (→) to indicate "goes to"; C(m)→A, for example, means that the contents of the memory register m goes to the accumulator (that is, that the contents of m will replace the current contents of the accumulator). C(A)→m means that the current contents of the accumulator will replace the current contents of m.

How the Machine Works

Each time that SMAC executes an instruction, it goes through five or six steps under the direction of the hardware in its control section. These steps are summarized succinctly in Figure 1.

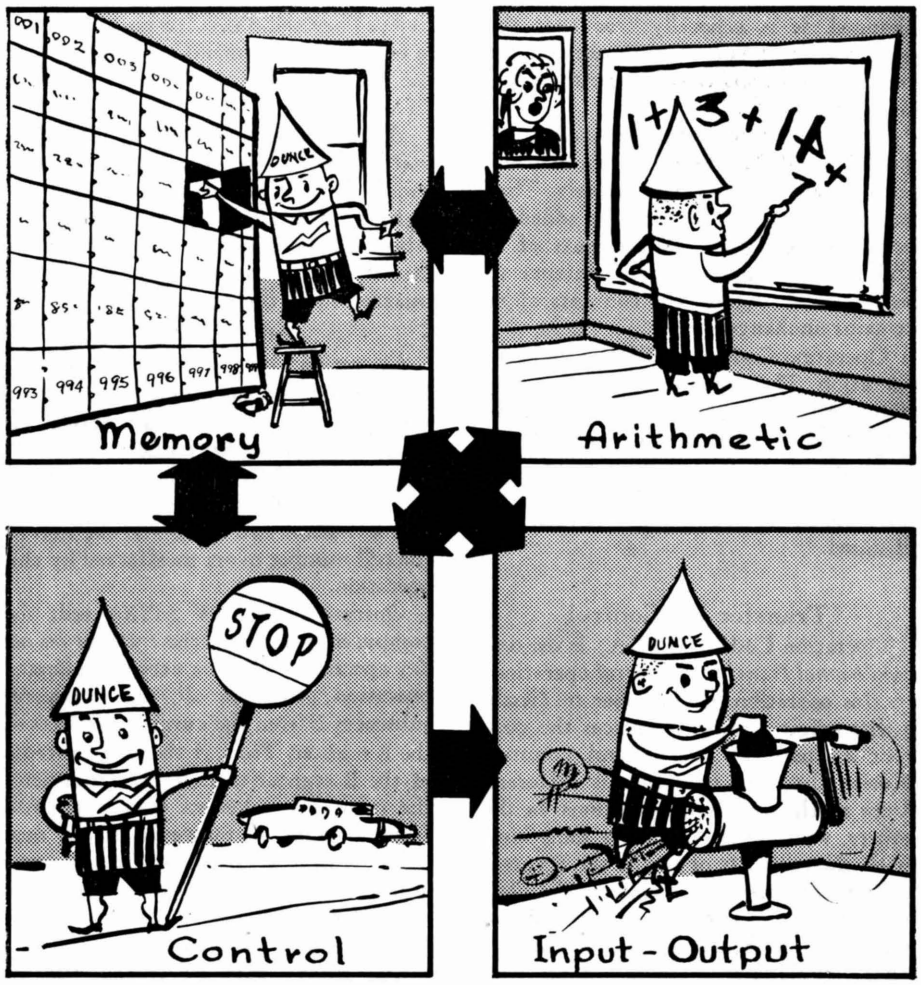
1. $C(C(CC)) \rightarrow IR$. The purpose of this step is to take the next command from the location specified by the *control counter* and load it into the *instruction register*. (The command actually tells the contents of the contents of the control counter to "go to" the instruction register.) Remember that the control counter is a three-digit register — just big enough to contain one address. So, in fact, C(CC), the contents of the control counter, is an address in SMAC's memory, and C[C(CC)] is the *contents* of that address in the memory. In fact, the contents of that address is to be used as the next instruction to be executed by SMAC.

2. $1+C(CC) \rightarrow CC$. The effect of Step 2 is to add *one* to the control counter. For example, if during Step 1 the control counter contained 173, after Step 2 it would contain 174. In performing this operation, SMAC takes successive commands from successive locations in memory.

3. *If the sign of IR is +, go to Step 5; if the sign of IR is −, go to Step* 4; and,
4. *Subtract C(B) from m (the last three digits of IR), and replace m by the result.* Steps 3 and 4 have to do with the *index register* (the B register). They say that if the sign of the next instruction is minus, we should subtract the contents of the B register from the address part of the next instruction before executing that instruction. On the other hand, if the sign of the next instruction is positive, then we should simply ignore the B register.

(It may be somewhat reassuring for you to know that you are not yet supposed to understand the purpose of the B register; this will be explained when we start to do a programming example.)

5. *Execute the instruction in IR.* Step 5 says that we are now to execute the instruction as it currently appears in the instruction register. At this stage of the game, the memory address portion of the instruction, m, may have been altered by having the contents of the B register subtracted from it. However, we will continue to call this altered address m.



SMAC
(Simple Minded Automatic Computer)

After we have finished executing the current instruction, we return to Step 1 and start the whole process over again.

Command Repertoire

In Figure 2, the SMAC's repertoire of instructions is listed in concise form. The first column gives a three-letter mnemonic abbreviation which is supposed to remind us what each operation code does. The second column lists the actual operation codes; these codes are the ones that will actually appear in the two left-hand digits of instruction words. The third column in Figure 2 lists exactly what each operation code tells SMAC to do.

The OP Codes

The first operation in the table has operation code 00. Its mnemonic is CLA, which stands for *clear and add*. The effect of this instruction is that the contents of the address designated by m (the address portion of this instruction), should be placed in the accumulator. Whatever was in the accumulator before is lost. However, the contents of the memory address number m remains un-

SMAC'S INSTRUCTION REPERTOIRE

Mnemonic	OP Code	Explanation
CLA	00	C(m)→A. Clear and add.
ADD	01	C(A)+C(m)→A. Add.
SUB	02	C(A)−C(m)→A. Subtract.
STO	03	C(A)→m. Store.
TRA	04	m→CC. Unconditional transfer.
TMI	05	If C(A) <0, then m→CC; otherwise proceed normally. Transfer on minus.
TZE	06	If C(A) $=0$, then m→CC; otherwise proceed normally. Transfer on zero.
LXA	07	Rightmost 3 digits of C(m)→B. Load index from address.
SXA	08	C(B)→Three rightmost digits of m. Store index in address.
TIX	09	C(B)−1→B; if C(B) $\neq 0$, then m→CC; otherwise, proceed normally. Transfer on index.
INP	10	Read the next word from the tape in the paper tape reader into address m.
OUT	11	Punch onto paper tape the word in address m.
HLT	12	STOP. Halt.

Figure 2

changed. It is generally true that the only way in which we can destroy the contents of a register is by storing something on top of it.

The next two operation codes, 01 (ADD) and 02 (SUB), are the codes for *add* and *subtract*, respectively. They cause the contents of m to be added to or subtracted from the contents of A and the resulting sum or difference to be left in A. Again, the contents of m remains unchanged.

Operation Code 03 (STO) is the *store* operation. The execution of an instruction word which has this operation code causes the contents of the accumulator to be placed in address number m. The previous contents of m are lost and the contents of the accumulator remain unchanged.

Transfer of Control

Operation Code 04 (TRA) is the *unconditional transfer of control* operation. In this operation, the number m (itself a three-digit number) replaces the current contents of the control counter. Notice that this operation is different from all the preceding operations. In the others, we were always doing something with the *contents* of m. In the transfer of control operation, however, the contents of m are undisturbed. Instead, we are simply instructing SMAC that *its next command is to be picked up from address number m* instead of from the usual spot.

Operation Code Numbers 05 (TMI) and 06 (TZE) are the *transfer on minus* and *transfer on zero* operations, respectively. They tell SMAC to go to location m for its next command in the event that the accumulator is either negative (TMI) or zero (TZE).

Operation Code 07 (LXA) is the *load index from address* operation code. This code tells the computer to take the three right-hand digits from the memory cell whose address is m, and place those digits into the B register. The contents of m is unaffected by this operation.

Store Index

Operation Code 08 (SXA) is the *store index in address* operation. It is the reverse of the LXA operation above. Its effect is to place the contents of the B register into the three right-hand digits of the memory cell whose address is m. The left-hand two digits and the sign of the cell number m are unaffected by this operation.

Operation Code 09 (TIX) tells the computer to perform the *transfer on index* operation. This is a very complicated operation; first of all, the computer subtracts 1 from the current contents of the B register. Then, if the new contents of the B register is different from zero, the computer takes its next command from location m. However, if the new contents of B is equal to zero the computer goes ahead and takes its next command from the normal location designated by the control counter.

Operation Codes 10 (INP) and 11 (OUT) are the input and output operations. They tell the computer to read the next word from paper tape into location m or to punch the next word from location m onto paper tape, respectively.

The final operation code is number 12 (HLT). It is the code for Halt. When SMAC executes this command, it stops computing and turns on a light on its console which says PROGRAM STOP.

Getting Started

A question that very often worries beginners in computing is "How does the machine get started?" On the SMAC the starting operation is accomplished through the use of a very convenient button which is labeled "LOAD PAPER TAPE." The depression of this button causes the following sequence of actions to take place:

1. The computer reads the next four words from the paper tape in the tape reader into memory cells 000, 001, 002, and 003.
2. The control counter is set to 000.
3. The computer proceeds to operate automatically, taking its first command (of course) from location 000. Location 000 contains one of the four words that have just been read in from paper tape.

At the beginning of his paper tape, the programmer will have punched a "Loader" program which will load itself into the computer's memory and then in turn load his program in. In the next chapter we will see an example of such a loader.

What's more, we will write down three programs to do the same simple calculation. We will see how we can economize on the use of memory by using different sections of our program a number of times, how a program can modify its own commands, and (at last) what useful purpose the B register serves.

Part III: A Problem in Addition

The problem we want to do is a very easy one, but its solution will illustrate many important programming techniques.

The problem is, simply, to add up 50 numbers (we shall call these numbers, $X_1, X_2, \ldots X_{50}$) and to place their sum (which we shall call S) in SMAC's memory cell number 700. We shall assume, for the sake of simplicity, that (1) each of these 50 numbers is a five-digit number, (2) their sum is also a five-digit number, and (3) in the course of adding them up, we shall never encounter a number with more than five digits in it. (The purpose of all these assumptions is to avoid having to worry about exceeding the word size of SMAC).

Subtotals

Thus far, we have labeled our *numbers* (X_1, etc.) and our *final answer*, S. From time to time in the course of our computing, however, we may have occasion to refer to a subtotal, or partial sum, of the numbers added thus far. Let us call these subtotals S_1, S_2, S_3 and so on. (You will note that S_1 will be the same as X_1, and S_{50} will be the same as S.) When we wish to refer to a subtotal without being specific, we shall call it S_i.

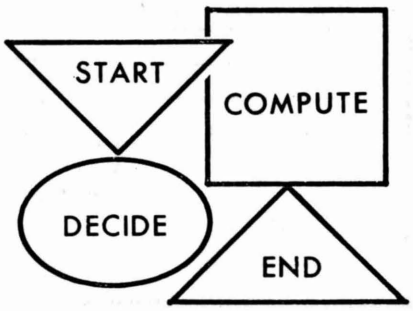
A few more assumptions, and we will be ready to begin. Let us agree that, by the time our program takes control of the computer, all of the numbers that are to be added up will already have been read into SMAC's memory from the paper tape. We assume that the first number, X_1, is in memory cell number 101, X_2 is in 102, etc., up to our last number, X_{50}, which is in cell number 150.

Let us also agree not to worry about getting our own programs into SMAC's memory. We shall suppose that our program has been punched on paper tape and has been placed in the appropriate memory cells by a "loader" program. Since memory cell numbers 101-200 have been pretty well taken up by our numbers X_1 and so forth, let us put our program into the memory cells numbered 201-300.

Finally, let us agree that the location of the first command in our program (Location 201) has already been placed in the control counter by the loader program. We are now ready to solve our problem in addition.

Flow Charts

Actually, we are going to solve our problem in three different ways, using three different programs. Each program presents a different attack on the problem, and each (except possibly Number 2) has its advantages and its disadvantages. Each program is first described by a *Flow Chart*. This chart, an essential of any program, is simply a sequence of



boxes, each of which describes in more or less detail a function to be performed by the computer. These boxes are connected to one another by arrows. The arrows describe the *flow of control* from box to box in the computer.

Differently shaped boxes represent different functions to be performed by the computer: an inverted triangle signifies the start of the program; a rectangular box signifies a computation to be performed; an oval box signifies that a decision is to be made (this is the only kind of box which can have more than one arrow going out of it); a right-side-up triangle designates the end of the program.

We give each important box on the Flow Chart a number; this number will be referred to on our programming sheet and will tell us what part of the program we are working on currently.

Programming Forms

On the next page, you will see illustrated the actual programming forms on which we write down our program. The sort of programming that we are doing here is called *machine language programming* (i. e., programming that is done in the computer's own simple-minded language). Machine language programming is harder than other kinds of programming, and the forms we use in connection with it are more complicated than the forms used for other systems. Machine language is worth your attention however, since other methods (which we shall describe next month) are based ultimately on this approach.

Our machine language coding form has seven columns. The first column, headed Box Number, designates the number of the flow chart box that we are currently working on. The second column, headed Location, tells us the address in SMAC's memory where this command is to reside as the program is being executed. The third column, headed Op Code Mnemonic, will contain the three-letter abbreviations which remind us just what the operation code in this command does. The fourth, fifth, and sixth columns contain the actual program words which are to be loaded into memory. (The fourth column gives the sign, plus or minus, the fifth column gives the Op Code, and the sixth column gives the address, m. For a review of the meaning of these terms, see Part I, page 3, column 2.) Column 7 of the programming form is headed Remarks. In this column, we write down what we are trying to accomplish with each command. It is a very important column.

A Simple-Minded Approach

In Program 1, which is based on Flow Chart 1, we solve our problem in the most simple-minded manner imaginable. First, we execute a *clear and add* (CLA) command, which brings the first number of our series (X_1) into the accumulator. Then we execute a sequence of ADD commands which successively add to the sum already in the accumulator the subsequent numbers X_2, X_3, and so on, until we reach X_{50}. At the conclusion of this sequence of 50 commands, we have the sum, S, in the accumulator. We then execute a *store* (STO) command, instructing the machine to put this answer into location 700. The total number of commands executed is 51. The total amount of memory required for our program is 51 cells.

We shall see, in due time, that Program 1 is the most efficient of the three programs from the viewpoint of time; it executes many fewer commands than either of the other two codes. However, Program 1 uses 51 memory cells to add up 50 numbers. If our problem had been to add up 100 numbers and we had written a program to do this using the techniques of Program 1, that program would have required 101 memory cells. In fact, when we use this technique the number of words in the program *always* depends on the number of numbers to be added together. Let us see whether we

can write a code which will avoid some of these problems.

A Complicated Approach

In Program 2, we use a programming technique known as *looping*. This means that various sections of the program are used more than once while we are doing our problem. For example, one ADD command (in Location 206 of Program 2) does all of the work accomplished by the 49 ADD commands of Program 1. Since Program 2's ADD command has to add numbers residing in different memory cells, the address part of the command must be modified during the program.

Before we turn to the program itself, let us get its logic firmly in mind by studying its Flow Chart, pictured on the opposite page. In boxes 1 and 2, the variables of our "loop"—i and S_{i-1}, are given their starting values of 1 and 0, respectively. Box 3 (the ADD command) is where the assigned task is actually performed; boxes 4 and 5 perform *loop control* functions, deciding that the loop is to be performed again or that the program is to proceed to the next step. Boxes 6 and 7 describe the terminal portion of the program, where we store our result (box 6) and halt (box 7).

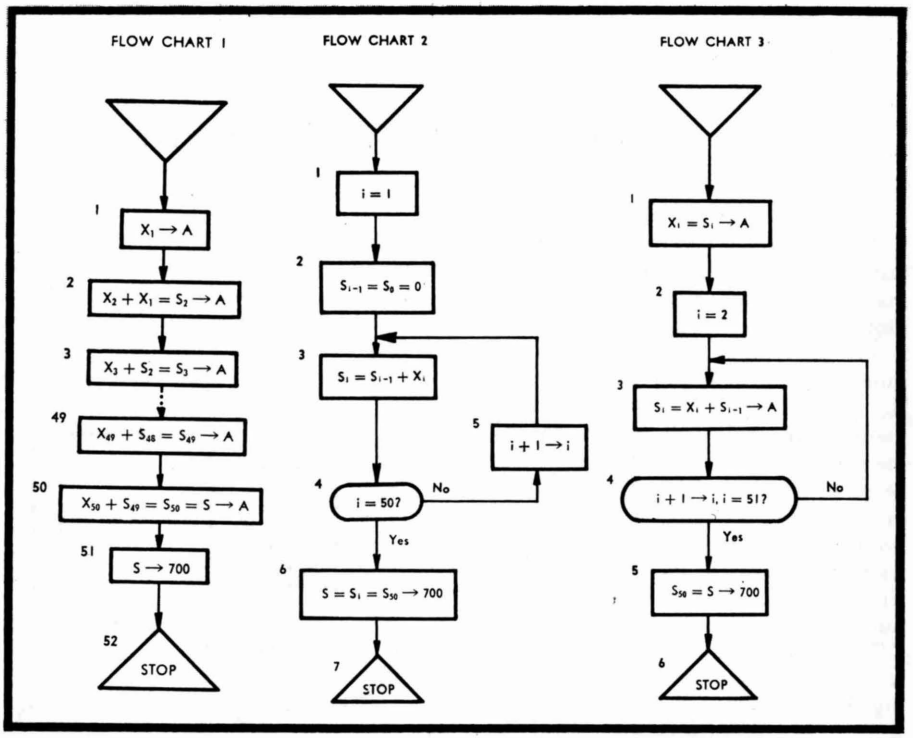
The Program

Now let us look at the program which carries out this logic. Its first moves (i.e., the commands in memory cells 201 and 202) are concerned with the ADD command in Location 206 which will be used throughout the program. What these two commands do is to set this ADD command to the first value that it must have: that is, +01 101. (So that it will be readily available to us, we have previously stored this number in Location 216—the first empty memory cell following those devoted to the program proper.)

In cells 203 and 204 of Program 2, we set the partial sum S_{i-1} to its initial value of zero. S_i, you will remember, is the sum of the first i X's. S_{i-1}, then, will be the subtotal which *precedes* S_i. The quantity i starts out with a value of 1, and we first compute S_1 (which is equal to X_1) by adding X_1 to S_0 (which is equal to 0).

In cell 205, we have a *clear and add* (CLA) command which places the current value of S_{i-1} in the accumulator. The ADD command in cell 206 adds X_i to S_{i-1}, yielding S_i, our subtotal. The *store* (STO) command in memory cell 207 places that value of S_i back into Location 700.

Now we are ready to see whether we are finished. *If i is equal to 50, we have just computed S_{50}, which is equal to the sum of the first 50 X's and, hence, is equal to* S, *our answer.* In that event, we



PROGRAM 1

Box No.	Location	Op Code Mnemonic	Sign	OP Code	m	Remarks
1	201	CLA	+	00	101	$C(101) = X_1 = S_1 \rightarrow A$
2	202	ADD	+	01	102	$C(102) + C(A) = X_2 + S_1 \rightarrow A$
3	203	ADD	+	01	103	$C(103) + C(A) = X_3 + S_2 \rightarrow A$
..	.....	.....	....	....	.....	....
..	.....	.....	....	....	.....	....
49	249	ADD	+	01	149	$C(149) + C(A) = X_{49} + S_{48} \rightarrow A$
50	250	ADD	+	01	150	$C(150) + C(A) = X_{50} + S_{49} = S \rightarrow A$
51	251	STO	+	03	700	$C(A) = S \rightarrow 700$
52	252	HLT	+	12	999	STOP

PROGRAM 2

Box No.	Location	Op Code Mnemonic	Sign	OP Code	m	Remarks
1	201	CLA	+	00	216	$C(216) = +01\ 101 \rightarrow A$
1	202	STO	+	03	206	$C(A) = +01\ 101 = +01\ L(X_1) \rightarrow 206$
2	203	CLA	+	00	217	$C(217) = 0 = S_{i-1} = S_0 \rightarrow A$
2	204	STO	+	03	700	$C(A) = 0 = S_{i-1} = S_0 \rightarrow 700$
3	205	CLA	+	00	700	$C(700) = S_{i-1} \rightarrow A$
3	206	ADD	+	01	101	$C(100+i) + C(A) = X_i + S_{i-1} = S_i \rightarrow A$
3	207	STO	+	03	700	$C(A) = S_i \rightarrow 700$
4	208	CLA	+	00	218	$C(218) = +01\ 150 = +01\ L(X_{50}) \rightarrow A$
4	209	SUB	+	02	206	$C(A) - C(206) = C(A) - +01\ (100+i) = C(A) - (+01\ L(X_i)) = 50 - i \rightarrow A$
4	210	TZE	+	06	215	If i = 50 take the next command from 215, otherwise take the next command from 211.
5	211	CLA	+	00	206	$i \neq 50$. $C(206) = +01\ (100+i) = +01\ L(X_i) \rightarrow A$
5	212	ADD	+	01	219	$C(A) + C(219) = +01\ (100+i) + 1 = +01\ (100 + i + 1) = +01\ L(X_{i+1}) \rightarrow A$
5	213	STO	+	03	206	$C(A) = +01\ L(X_{i+1}) \rightarrow 206$
5	214	TRA	+	04	205	Take next command from 205
7	215	HLT	+	12	999	STOP
	216		+	01	101	$+01\ L(X_1)$
	217		+	00	000	0
	218		+	01	150	$+01\ L(X_{50})$
	219		+	00	001	1

are through, since we have already placed this answer in Location 700, which is just where we want it. *However, if i is not equal to 50, we are not yet through. Instead, we must add 1 to the current value of i and perform more additions.*

Commands which permit us to find out what the current value of i is are contained in cells 208, 209, and 210.

PROGRAM 3

Box No.	Location	Op Code Mnemonic	Sign	OP Code	m	Remarks
1	201	CLA	+	00	101	$C(101) = X_1 = S_1 \rightarrow A$
2	202	LXA	+	07	207	Rightmost 3 digits of $C(207) = 049 \rightarrow B$
3	203	ADD	−	01	151	$C(151 - C(B)) + C(A) = X_i + S_{i-1} = S_i \rightarrow A$
4	204	TIX	+	09	203	$C(B) - 1 \rightarrow B$. If $C(B) \neq 0$ go to 203, If $C(B) = 0$ go to 205.
5	205	STO	+	03	700	$C(A) = S \rightarrow 700$.
6	206	HLT	+	12	999	STOP
	207		+	00	049	49

We find this value by looking at the address part of the command in Location 206. This is the address of X_i, and we know that X_i is stored in Location 100 + i. So, if cell 206 currently contains +01 150, then we know that i must be equal to 50. We check this by subtracting the current contents of cell 206 from +01 150 (which we have thoughtfully placed in cell 218 in advance). If the result of this subtraction is not zero, then we know that i is not yet 50.

In the event that i *is* equal to 50, then, of course, the problem is over and we go to Location 215, which contains a HLT, for our next command. In the event that i *is not* yet equal to 50, we go to Location 211, where we increase i by 1. Since the only command in our program which refers to i directly is in Location 206 (this is the command in which we add X_i to S_{i-1}), we must add 1 to the address part of 206 in order to increase i. This is done in Locations 211, 212, and 213. At Location 214, we go back to Location 205 and repeat this whole process.

Self-Improvement

If you have followed us closely, you will already have noted that the contents of cell 206 are changed several times during the execution of Program 2. This is important because cell 206 contains one of the commands in the program. We observe, thus, that a program may alter itself. This is very important.

Now let us see how our Program 2 compares with our first, more simple-minded, Program 1. We see that Program 2 and the constants associated with it require 19 spaces in memory as compared with the 51 spaces that were needed by Program 1. However, as we work our way through the program we see that those commands located in cells 205 through 210 are each executed exactly once; the commands located in cells 205 through 210 are each executed 50 times; the commands located in cells 211 through 214 are each executed 49 times, and the command in cell 215 is executed once. When we add all these executions up, we observe that 501 commands are executed by Program 2 in the course of adding up our 50 numbers. This means that Program 2 will require ten times as long to execute as Program 1. So, by saving a little over a factor of two in memory requirements, we have sacrificed a factor of about ten in speed. It is interesting to note, however, that by changing the constant in cell 218 we can make Program 2 add up any number of numbers that we may require.

A Balanced Approach

In our third and last program, we do the same problem in addition, achieve even better savings in space, and make a substantial improvement over Program 2 in time.

Program 3 employs the B register, or *index register*. We hope that the importance of this register will be illustrated in this example.

Program 3 is similar to Program 1 in that we are not required to store intermediate partial sums as we go through the program. We begin, in Location 201, with a *clear and add* (CLA) command which places our first number (X_1) in the accumulator. Next, in Location 202, we place the constant 49 in the B register. In Location 203, we have an ADD command—an ADD command, however, whose sign is *minus*. Let us see what happens the first time we execute the command in Location 203. The control counter now contains 203, so we place the contents of 203 (namely, −01 151) in the instruction register. We then add 1 to the control counter, so that it now contains 204. Next, we examine the sign of the instruction register and observe that it is minus. This tells us that before executing the command we must subtract the contents of the B register from the address part of the instruction register. The address part of the instruction register contains 151; the B register contains 49. After we perform the required subtraction, the instruction register contains 01 102; we then proceed to execute this command. It says: *Add the contents of 102 to the number that is already in the accumulator.* The number that is already in the accumulator is X_1 (or S_1, as we sometimes call it). The contents of 102 is X_2. We complete this addition, leaving $X_1 + X_2 = S_2$ in the accumulator.

In Location 204, we have a *transfer on index* (TIX) command. This command tells the machine to subtract 1 from the B register and then, if the new contents of B is different from 0, to take its next command from Location 203 again. The first time we went through this piece of code, the B register contained 49; after we perform this subtraction, it will contain 48. Since 48 is indeed different from 0, we must go back to 203 for our next command. And since 203 contains the command −01 151, we must subtract 48 (the contents of B) from 151 before executing the command. The command that we actually execute is 01 103, which says: *Add X_3 to the number that is already in the accumulator.*

HLT at Zero

We keep repeating this two-step loop until we have added X_{50} into the accumulator. At that time, the B register will contain 1. This time, when we execute the command in 204 and subtract 1 from the B register, our result is 0. Now, for the first time, the B register *does* contain 0, and we are sent to 205 for our next command. Here, we are told to *store* (STO) the result, S, in Location 700. This completes the program, and we go to Location 206 where we are told to HLT.

The total memory required for this code is 7 cells. The total number of commands executed is 102, since those in Locations 201, 202, 205, and 206 are each executed once, and the ones in 203 and 204 are executed 49 times each.

This code, then, requires one-seventh the memory required by Program 1, and takes only twice as long to execute. If we were doing this problem in real life, this is undoubtedly the code that we would use.

Part IV: Programming Languages

We have just written three programs for the SMAC Computer. These programs were written down in the language of the computer itself. This type of programming is called *Machine Language Programming*.

When we program in machine language we write down the actual address in memory where each command is to be stored; we write the actual numeric operation code for each command and we write the numeric address of the operand which this command refers to. If we are writing short programs for a decimal or alphanumeric computer, machine language programming can be relatively satisfactory. However, when we try to write long programs or when we try to write programs for binary computers, the drawbacks of machine language coding soon become obvious.

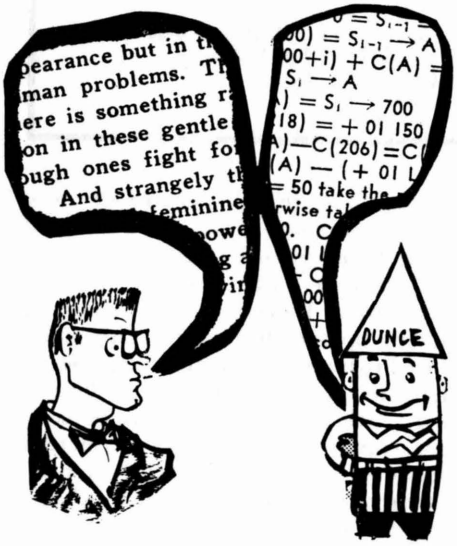
The Human Factor

One of the major difficulties is associated with the fact that people make mistakes when they write programs. If we have made a mistake in a machine language program and we wish to correct that mistake by inserting an extra command in the middle of our program, then it is necessary for us to go through the whole program and advance by one the location in which each command following the inserted command is to be placed in memory; at the same time, we must be careful to modify appropriately any operand addresses which refer to that section of coding. If our code were several thousand words long this would obviously be an impermissibly tedious task.

$2 + 1 = 11$?

If we try to write programs in machine language for a binary computer, the situation immediately becomes even worse. Now the probability of our making an error is compounded by the fact that we are working in a number system which is unfamiliar to us and by the necessity for our writing down many more symbols than were necessary for a decimal or alphanumeric machine. For example, the word length on the 7094 Computer is 36 binary digits and the address length on that computer is 15 binary digits. This means that if we were to write in machine language for the 7094, then for each instruction, we would have to write a location consisting of 15 binary digits and a command consisting of 36 binary digits. Even if we were able to write down a sequence of such commands without making any errors, we would be almost certain to make some mistakes when we tried to punch those commands into tabulating cards.

Quite early in the game it began to occur to people that the computers which were the source of all these difficulties might also offer solutions for some of them. For example, it proved very



easy to write a program called an Octal Loader which allowed the programmer to write his program in the octal number system instead of in the binary system. It turns out that it is almost trivial to translate back and forth between the octal (or base 8) number system and the binary (or base 2) system; however, if one writes in the octal system, he needs to write down only one-third as many symbols as he would if he were writing in the binary system. The programmer who wrote his program down in octal would also punch it into cards in octal. Then, instead of loading his program directly into the machine as he would have done had it been a machine language program, he would first load the octal loader program into the machine. This octal loader program would then treat his code as data; it would read in the octal cards, translate them into regular binary machine language, and output this binary machine language program on a separate card deck. Now, whenever the programmer wishes to run his problem, he loads this binary machine language deck into the computer followed by whatever data he wants to process.

The process explained above is typical of the sorts of things that happen in connection with all automatic programming systems: a program is written in some language other than machine language; it is then input to some sort of a translating or processing routine which converts it into a machine language program. This machine language program is the output of the automatic programming system. This translation need occur only once, provided the programmer has not made a mistake.

Assembly Programs

After the invention of the octal loader, developments in automatic programming took place very rapidly. It became clear that it would be very easy to have the computer translate mnemonic operation codes such as CLA, ADD, TRA, etc., into the corresponding numeric machine-language operation codes. It also became clear that it would be possible for the programmer to write his program using *symbolic* addresses instead of absolute numeric addresses, and that the computer could then translate these symbolic addresses into the required absolute numeric addresses. So, for example, the programmer instead of writing 101, might write X and the computer would be given the task of translating this symbolic address, X, into the absolute address 101. The programs which perform the translation from mnemonic operation codes to numeric operation codes and from symbolic addresses to numeric addresses are called *Assembly Programs* or *Symbolic Assembly Programs*.

Let SMAC Do It

These symbolic assembly programs still have the property that each line of code that the programmer writes down generates only one machine-language instruction; the programmer is still thinking in the way that the machine thinks. He must still break down his program into basic machine instructions. He has only delegated to the machine certain tedious bookkeeping tasks.

The language that the programmer uses when writing a program which is to be processed by an Assembler is called an *Assembly Language*. These days no one writes in absolute machine language. When we say that a program is written in machine language, we mean that it is written in assembly language.

Problem Oriented Languages

Because of the essential identity between assembly language and machine language, the programming of a problem

in assembly language was still a fairly tedious task. However, it wasn't long before several people got the idea that it might be possible to develop a programming language which resembled more closely the language we ordinarily use when describing a procedure for solving a problem on a computer. The idea was, of course, that the programmer would write his program in this "Procedure Oriented Language" and that the computer, using a program called a *Compiler*, would translate this code into a machine language code.

Source Programs

Let us examine some of the terminology associated with automatic programming systems. The programming languages we are talking about are called *Procedure Oriented Languages* or *Problem Oriented Languages* or sometimes, for short, POL's. The computer programs which translate programs written in these POL's into machine language are called *Compilers*, or *Processors*, or *Translators*. Programs written in POL's are called *Source Programs;* the machine language programs into which they are translated are called Object Programs.

Problem Oriented Languages have developed in two principal fields: engineering-scientific computing and commercial data processing. When trying to solve problems in engineering or science, one is usually concerned with doing a lot of fairly complicated arithmetic. Often, it is difficult to tell in advance when doing such problems what the sizes of the numbers involved will be. These factors must be taken into account when one is designing a language which is to be used to describe procedures for solving this class of problems. On the other hand, in solving problems in commercial data processing, one has to do only a small amount of arithmetic in general and the sizes of the quantities involved are relatively well known in advance; however, when doing such problems one is often faced with very much more input-output than is required in most scientific calculations. Languages to be used in describing solutions of business problems must reflect these properties.

FORTRAN

By far the most popular engineering and scientific POL in use today is FORTRAN. FORTRAN is an acronym for Formula Translation. This language was developed by the International Business Machines Corporation in cooperation with certain computer users; Robert Hughes of LRL's Livermore Laboratory is one of the designers of this language.

The first machine for which a FORTRAN Compiler was written was the IBM 704. That Compiler was completed in 1956 at a cost of around half a million dollars. Since that time, FORTRAN Compilers have been written for many other computers, and their unit cost has gone down substantially.

In Figure 1, an example of a FORTRAN program is shown. This program does the same problem that we did last month. The problem is simply to add up 50 numbers. In FORTRAN, we call these numbers X(1), X(2), X(3), . . . X(50). The first statement in the program DIMENSION X(50) simply tells the FORTRAN Compiler that there are to be at most 50 X's. The next statement sets the partial sum S equal to 0. Incidentally, the equals sign in FORTRAN has a somewhat different meaning than it does in ordinary algebra. In FORTRAN, the equals sign means "is to be replaced by." So, for example, S=0 means that the current value of the variable whose name is S is to be replaced by 0. The next statement, DO 23 I=1,50, means that the computer is to execute all

```
FORTRAN
      DIMENSION X(50)
      S=0
      DO 23 I=1,50
23    S=S+X(I)
```

the statements following this one down through and including statement number 23, and it is to do this while the variable I successively takes on the values 1,2,3,...,50. Statement 23, as a matter of fact, is the very next statement. It says S=S+X(I) which means that the variable whose name is S is to be replaced by the current value of S+X(I). This statement is executed first with I=1 and S=0, so that the new S is equal to 0+X(1). The next time through, I is equal to 2, and the new S is equal to X (1) + X (2). The next time we obtain X (1) + X (2) + X(3) and so on. Finally, when I=50, we obtain the sum of all 50 X's. The FORTRAN Compiler will translate this source program into an object program which looks very much like Program No. 3 of last month's article.

ALGOL

ALGOL is another engineering and scientific language. ALGOL, which is an acronym for Algorithmic Language, was designed in 1958 by a committee of European and American mathematics and computer experts. The ALGOL language is more elegant than FORTRAN and is much more precisely specified. ALGOL Compilers have been written for the CDC 1604, the Burroughs B5000, and the IBM 7090 computers.

In Figure 2, an ALGOL program to solve our addition program is exhibited. Although ALGOL is a more precise language than FORTRAN, it is somewhat farther away from our usual algebraic

```
ALGOL
begin real array X[1:50];
      real S; integer I;
S := 0. ; for I := 1 step 1 until 50
do S:=S + X(I); end;
```

language than FORTRAN is and hence has less intuitive appeal to many of us.

In the United States, ALGOL is used at certain university computer centers, such as the Stanford Center. It is also used very extensively in Burroughs Computer Installations.

Examples of ALGOL programming can be seen in the Algorithm section of the Communications of the Association for Computing Machinery every month.

Business Languages

Late in 1958, the Department of Defense organized a committee of computer users and manufacturers to design and specify a common business-oriented computer language. The language which this group designed is now known as COBOL. It is, in fact, a common business-oriented language and is very much more popular than any other such language. COBOL Compilers have been written for all principal computers on which business applications are performed.

The COBOL language is designed to deal effectively with the large input-output problem associated with commercial data processing, and with the logical

```
COBOL
MOVE 0. TO S
PERFORM SUMMING
VARYING I FROM 1 BY 1
UNTIL I EQUALS 50; END RUN.
SUMMING. ADD X(I) TO S.
```

complexity of business computer applications. It is, in fact, true that business applications are quite often very much more complex logically than are engineering and scientific applications.

For example, consider the straightforward problem of computing a soldier's pay. If the soldier goes AWOL for one day on the 28th of February, 1963, he loses three days' pay; however, should he go AWOL for one day on the 31st of March, 1963, he does not lose any pay at all. If we write a computer program to compute military payroll, the program must cope somehow with this

rather anomalous situation. In Figure 3, a segment of a COBOL program to add up 50 numbers is illustrated. This segment of the COBOL program is called the procedure section. There are other sections called the environment section and the data section.

Programming Made Easy

In the eight years since the invention of Problem Oriented Languages, their use has increased phenomenally. At the Laboratory, more than 50% of the programs currently being written are being written in FORTRAN. In other large computer installations, almost 90% of all programs are now written in FORTRAN. Many business installations write only in COBOL.

The reasons for this revolution are fairly clear. It is possible to learn a language like FORTRAN in about a week, whereas to learn machine language programming on a large binary computer would require two or three months. Once a program has been written in a problem oriented language, it is possible to compile and run it on any computer for which a Compiler for that language has been constructed.

A remarkable application of FORTRAN occurred at Livermore recently when, under the direction of Hans Bruijnes, a FORTRAN Compiler for the CDC 3600 was written in the FORTRAN language. The construction of this Compiler required less than one man year; this is to be contrasted with the 20 man years that were required for the construction of the first FORTRAN Compiler for the IBM 704.

Part V: Computers at LRL

In the first part of this article, we asserted that the Radiation Laboratory contained "one of the greatest concentrations of automatic computing power to be found anywhere on earth." In this chapter we hope to make this strong assertion seem a little more believable by reviewing the history of computers at LRL and describing the models currently in use.

The Laboratory acquired its first automatic digital computer in 1952; this machine, which was installed at the Livermore Laboratory, was the famous UNIVAC I. It was manufactured by Remington-Rand and was the first stored-program automatic digital computer to be offered commercially in the United States. The Laboratory's machine was the fifth to be manufactured.

Dawn of an Age

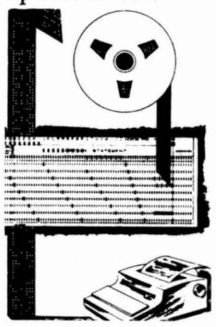
Although the UNIVAC I was slow by modern standards (its memory cycle time was about 242 microseconds), it had many sophisticated features. Its only input-output medium, other than a typewriter for operator instructions, was magnetic tape; magnetic tape input-output was *buffered*, which means that the machine could compute at the same time as it was reading or writing on tape.

However, the fact that the only input medium to the machine (for both programs and data) was magnetic tape created certain problems. It was not possible, at the time when the UNIVAC I was installed, to punch data on cards and then transfer them to magnetic tape; one had to use a machine called a "Unityper" to transfer data directly from a keyboard to magnetic tape. The Unitype Operator could not conveniently see what she had put on the tape, and had no convenient method for verifying this information. The great success of the UNIVAC I operation is a tribute to Livermore unitypist Cecilia Larsen, who during her entire career at the Laboratory has made only 17 mistakes.

Binary Machines

The next machine to be installed, an IBM 701, was the first of a long sequence of IBM scientific computers at the Livermore and Berkeley Laboratories. The 701 was a binary machine—unlike the UNIVAC I, which was alphanumeric. It had a word length of 36 binary digits.

The 701 employed a Williams Tube memory. In this memory system, information was stored in the form of dots on the surface of a cathode ray tube. It was probably the least reliable memory system ever invented. Information would capriciously disappear from or appear in the memory; occasionally when someone opened a door or window and allowed a beam of sunlight to strike one of the tubes' surfaces, the machine would pick up several bits.



When the 701 was released by the Livermore Laboratory in 1956, it was moved to UC's Berkeley campus, where it became the first computer to be installed at the campus Computer Center. At the time of this move, its Williams Tube memory was replaced by a magnetic core memory, which proved to be very much more reliable.

A Compatible Family

The next machines to be installed at Livermore were IBM 704's. They replaced the 701. The introduction of the 704 in 1956 marked the beginning of a period in the history of scientific computation which is just now coming to an end. The 704 was the first of a sequence of IBM-produced scientific computers which were to be more or less compatible; that is, programs written for one member of the sequence could be run on subsequent members of the sequence. Members of this series of computers were the 704, the 709, the 7090, the 7094-II, the 7040, and the 7044. One or more of each of these types of computers has been installed at the Laboratory at one time or another.

Like the 701, all of these machines were binary 36-bit word computers. They employed sign and magnitude arithmetic, used single-address logic, and had one command per word. All of these machines, with the exception of the 704, had buffered input-output. The 704 and 709 were vacuum-tube machines, and all the rest are solid-state.

Another family of scientific computers represented at the Laboratory includes the CDC (Control Data Corporation) 1604 and 3600. These are large-scale scientific machines; they have a 48-bit word, use single-address logic, and (like the UNIVAC I) have two commands per word.

There are currently two 3600 computers installed at Livermore. It is for these machines that the FORTRAN Compiler written in the FORTRAN language was devised.

Few-of-a-Kind Machines

From a very early time in its history, the Livermore Laboratory has supported the development of advanced computer hardware. The Laboratory was motivated to do this because it had computing requirements which continuously outpaced the capacities of commercially available computers.

The first machine whose development the Laboratory sponsored was the LARC (Livermore Advanced Research Computer). It was designed and manufactured by Remington-Rand to the Laboratory's specifications, and was finally delivered to the Laboratory in 1960. One additional LARC was manufactured for the Navy's Bureau of Ships.

The LARC is a decimal machine with a 12-decimal digit word. Its design includes many important innovations. It was the first machine to incorporate an independent computer as an input-output processor. It was also the first machine to employ large-scale mass storage devices for input and output buffering. Another innovation which it included was the use of a cathode ray tube as a primary output medium.

The STRETCH

The development of the IBM STRETCH computer was initially sponsored by the Los Alamos Scientific Laboratory. It is another large-scale binary, scientific computer. Approximately seven of these machines were manufactured; machine number two is at the Livermore Laboratory.

Notable features of the STRETCH system are a very high-speed mass storage system and ingenious hardware provisions for setting up subsequent commands for execution while processing the current command.

Although the CDC 6600 Computer is now part of the Control Data Corporation's standard product line, the first machine was developed under a contract with the Livermore Laboratory. This machine, which is the most powerful of all the computers mentioned so far, has just been delivered to the Laboratory.

The 6600 incorporates a radical design. Because of this design, it can perform several arithmetic or logical functions simultaneously. Its input and output are handled by ten independent com-

puters or peripheral processors which are built into the main machine. The main memory consists of 130,000 words of 60 binary digits each.

Support Computers

The problem of preparing information to be processed by computers, and the problem of presenting output from these computers in a meaningful form, are very difficult ones. The Berkeley and Livermore Laboratories have arrived at somewhat different solutions to these problems.

These solutions depend on the nature of the problems done at the two sites. At Berkeley, the principal computation requirement is in the area of scientific data reduction. Here, very large volumes of input are involved and only moderate volumes of output result. At Livermore, the situation is just reversed. There, the principal interest is in solving large problems of hydrodynamics; a small volume of input may result in a very large volume of output.

In Berkeley, the large volume of data arising from the measurement of the position of tracks on bubble chamber film is either fed to computers directly from automatic measuring machines (like the Flying-Spot Digitizer) or is written by measuring machines on a medium such as paper tape, cards, or magnetic tape, which may then be read by a computer. The primary input medium for programs and for other types of data is cards. These cards are not read directly into large-scale computers but are first transcribed onto magnetic tape. The transcription is performed by small peripheral computers; IBM 1401's are used for this purpose at both Berkeley and Livermore.

At Berkeley, the peripheral computers also serve as printers. Output which is to be printed is first written by the large scientific computers on magnetic tape, and the information on the tapes is later printed through the facilities of one of the peripheral 1401's.

The Versatile 1401

The IBM 1401 (which, by the way, is the most popular computer ever manufactured) is an alphanumeric, variable-word-length, two-address machine. A variety of input-output equipment may be attached to the 1401, including magnetic tapes, card readers, card punches, printers, plotters, paper tape readers, etc. The 1401 can read cards at the rate of 800 cards per minute, and it can print at the rate of 600 lines per minute. There are currently four 1401's in Berkeley and two in Livermore.

Printing at Livermore

At the Livermore Lab, the printing problem is much larger than it is in Berkeley. Historically, Livermore has solved this problem by procuring very high-speed printers. Printers currently installed at Livermore include two 5000-line-per-minute SC 5000 machines, manufactured by General Dynamics, and the recently delivered 30,000-line-per-minute Radiation Inc. printer.

Storage Systems

It is perfectly obvious that even if the entire scientific staff of the Laboratory spent all of its time examining computer output, it would still be possible to look at only a small fraction of the output of these very high-speed printers. The difficulty is that although we may want to see only a few numbers which result from a given computer run, we generally do not know in advance just what numbers these are; so, instead of printing out only a few numbers and then rerunning the problem whenever we want to print out a few more numbers (which would be very expensive), we quite often print out a very comprehensive selection from the output of a given computer run. We can then store this output on our shelf and whenever we want to look at a few more numbers we can find them in the course of five or ten minutes without asking for an

Computing Machines at the Laboratory

(brought up to date as of September 1965)

Machine Name	Manufacturer	Word Size	Memory Size	Memory Cycle Time	Number	Site	Power Relative to 7094
1401	IBM	1 Alphanumeric Character	4000-12000 Characters	11.5 Micro-seconds	2 4	L B	—
7040	IBM	36 Binary Digits	32,768 Words	8 Micro-seconds	1	B	0.2
7044	IBM	36 Binary Digits	32,768 Words	2 Micro-seconds	1	B	0.5
7094	IBM	36 Binary Digits	32,768 and 65,536 Words	2.0 Micro-seconds	2 1	L B	1
3600	CDC	48 Binary Digits	65,536 Words	1.5 Micro-seconds	2	L	1.5
LARC	Sperry-Rand	12 Decimal Digits	30,000 Words	4 Micro-seconds	1	L	1.5
STRETCH	IBM	64 Binary Digits	96,000 Words	2.2 Micro-seconds	1	L	3
6600	CDC	60 Binary Digits	131,092 Words	1 Micro-second	1 1	L B	16

additional computer run.

This system works quite effectively. However, it has obvious disadvantages in that it creates a large demand for storage space and the paper costs associated with it are very large.

In order to eliminate these difficulties and to create a generally more effective man-machine relationship, a new input-output handling system is presently being designed and implemented at Livermore. This system is called *Octopus*. It substitutes for the present paper storage of computer output a large machine-readable memory. This memory is to hold all output from the principal Livermore computers for as long as this output is of interest. The user will retrieve information from this mass memory either via conventional printers or via one of several remote input-output stations. Current versions of these input-output stations include a teletype machine and a small incremental plotter; ultimately the stations will include CRT-type display devices.

When the user wants some information he approaches the teletype keyboard, types in the serial number of his problem and specifications about the data that he wants. This data will then be retrieved from the mass memory and presented to him on the input-output device of his choice. A Digital Equipment Corporation PDP-6 Computer is being acquired to control the Octopus complex.

Livermore people who are working on the Octopus project include George ing, Bob Abbott, Bob Wyman, J. Carver Michael, Norman Hardy, Bud Wirsch-Hill, Ed Lafranchi, and Jerry Russell.

The chart on page 7 lists some vital statistics about computers currently installed at the Laboratory. The column headed "Power Relative to 7094" requires some explanation. When a 2 appears in this column, it means that the computer in question could do in one hour a set of problems that the 7094 would require two hours to do, and so on.

Of course, different computers are better at some problems than others; the figures given reflect a typical mix of Livermore problems.

Part VI: Applications at LRL

So far in this series, we have described the way in which digital computers work; we have told about the automatic programming systems which make the use of digital computers easier; and we have described the more important computers which are currently installed at the Laboratory. In this chapter we will tell about some of the problems which are solved on these computers.

Hydrodynamics

A large proportion of the calculations which are done at the Livermore Laboratory are associated with hydrodynamics problems. These problems have to do with the motion of gases or liquids; they arise very frequently in connection with programmatic research conducted at Livermore.

A mathematical model of a new nuclear warhead can be put on a computer to determine whether the model has a prospect of success or failure. Many models can be tested inexpensively, and many failures eliminated without actually constructing prototypes. In developing mathematical models, the larger the number of factors taken into account by the model, the more accurate the evaluation by the machine.

Simulated A-Tests

Computers, although no substitute for the testing of the final product, are in a sense an unlimited "pretesting" range. The only alternative to the use of computers would be crude models, much trial-and-error field testing, many failures, greater cost, and slower progress.

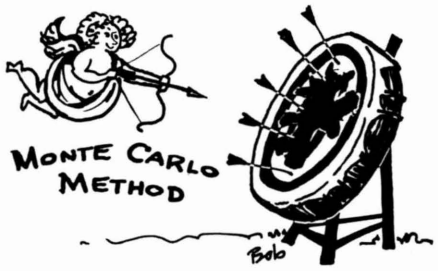
Computers are essential in other research and development at Livermore. They were critical in the design of the reactors for Project Pluto, the program to develop a nuclear ramjet engine.

Project Plowshare, the program to develop a technology for the peaceful use of nuclear explosives, is also dependent on computers. The design of special nuclear devices for a variety of special purposes in Plowshare is done in much the same way as is the design of nuclear weapons. Computers are also coming into use in the analysis of plasmas, the extremely hot gases produced in experimental machines in Project Sherwood, the program to develop controlled thermonuclear reactions.

Mathematically speaking, a hydrodynamics problem involves the solution of partial differential equations. In order to solve these equations on a digital computer, one generally substitutes for them difference equations, whose solutions are, one hopes, close to the solutions of the original equations.

These problems require very large amounts of computer time for their solution. They constitute an important reason for the Livermore Laboratory's acquisition of very large-scale machines, such as the new Control Data 6600.

In a rather interesting instance of a hydrodynamics application, Livermore physicist Chuck Leith is using the LARC



computer to study the global movement of air masses. His programs are capable of computing the movement of air in an entire hemisphere. It is hoped that this work will ultimately lead to moderately long-range weather forecasting systems.

Monte Carlo

A large class of problems which are of interest at Livermore are attacked most conveniently by so-called Monte Carlo techniques. These techniques involve devising a probabilistic game whose distributions of outcomes corresponds to the solution of a certain mathematical or physical problem, and then of playing that game with a digital computer and observing this distribution of outcomes.

Suppose, for example, that the problem that we are faced with is determining the area of a certain figure which we have inscribed on a square sheet of fiberboard. There are a number of ways in which we could attack the problem. We could, for example, cover the sheet of fiberboard with a network of grid lines which divide it up into small squares, and then count the number of such small squares which lie inside our figure. Alternatively, we could take a bandsaw and cut out the figure and weigh it (its weight would presumably be proportional to its area). Or, using numerical techniques in a computer, we could compute the integrals of the functions which define the bounding curves of our figure. These methods are all *deterministic* ways of solving our problem. Let us consider a Monte Carlo technique for solving the same problem.

Suppose we have a dart-throwing machine with which it is possible to hit our piece of fiberboard at each throw, but which has the property that successive throws of the dart will hit any section of the board with equal probability. This means that the probability of hitting any figure inscribed on our rectangle is proportional to the area of that figure. We now proceed empirically to determine the probability of hitting the figure inscribed on our piece of fiberboard by throwing darts; we count as successes throws on which we hit our figure and as failures throws on which we miss our figure. We can then estimate the probability of hitting the figure by dividing our total number of successes by our total number of throws. Then, to determine the area of the figure, we need only to multiply this estimated probability by the total area of the board.

We can play this game inside of a digital computer by substituting for a dart throw the selection of two random numbers, which represent the coordinates of the point on our board.

By using the Monte Carlo method, it is possible to attack large classes of physical problems which do not readily yield to more sophisticated mathematical techniques.

Orbits

For a number of years, Joe Brady of the Livermore Computation Group has been carrying out a program of research in celestial mechanics. One of the earliest programs for computing the orbits of the near-earth satellites was written by Brady and his co-workers; in fact, at the time the first Sputnik was launched, Brady's program was the only operational one in the country. Since that time, Brady has been working on a definitive orbit for the planet Mars. This work resulted in the publication, last year, of a set of Mars coordinates extending from the year 1800 to the year 2000.

Astronomers, especially those interested in celestial mechanics, were among the earliest users of automatic computing equipment. The advent of artificial earth satellites greatly increased the computing requirements of this group. As much computer time is required to compute one orbit of an artificial satellite about the earth as is required to compute one revolution of the earth about the sun. The difficulty is that the satellite requires only about 90 minutes to go around the earth, whereas the earth takes a year to

go around the sun. Brady's pioneering work at Livermore has exercised a significant influence in this important area of computer applications.

Berkeley Computing

The nature of the computing load at Berkeley is influenced by the goal of the Berkeley Laboratory, which is to conduct basic research in the physical and biological sciences. Most problems that come into the Berkeley Computing Center have to do with the reduction of experimental data, with the design of experiments, with theoretical physics calculations, or with the design of particle accelerators.

By far the largest applications area at Berkeley is the reduction of data from bubble chamber experiments. This process takes place in several stages, not all of which are directly associated with computing machines.

The first stage consists in the exposure and processing of pictures of bubble chamber events.

The second stage involves the manual scanning of these pictures. In this stage a person called a scanner looks at each picture to see whether it contains an "interesting" nuclear event. If it contains such an event, he notes the roll and frame number of the picture on which this event is recorded and perhaps a rough location of the event on the film.

Automatic Measuring

The third stage consists of accurately measuring the locations of points along the tracks on the film which make up the events of interest. Several automatic and semi-automatic devices are available at the Laboratory to perform this measuring function. These devices include the Franckenstein, which was developed at the Laboratory by Jack Franck. This machine is a projection microscope which automatically follows bubble chamber tracks, measures the locations of points on the tracks, and writes the coordinates of these positions out on magnetic tape for later use on a digital computing machine. Another measuring device developed at the Laboratory is the Spiral Reader. This machine, which is very much faster than the Franckenstein, operates under the control of a small digital computer, and again places its output on magnetic tape for later analysis on a large computer.

The Scanning-Measuring Projector, invented by Luis Alvarez, operates under the control of a medium-scale computer. One 7040 Computer is currently controlling five of these devices simultaneously.

The Flying-Spot Digitizer, or FSD, developed at the Laboratory under the supervision of Howard White, is the most automatic of all the measuring devices mentioned so far. It operates under the direct control of a 7094-II computer. In this system, the computer, under the control of a magnetic tape produced when the film was scanned, turns the film to the frame numbers which contain interesting events, locates the fiducial marks on these frames, and digitizes bubble locations. The computer program which controls the FSD device also selects coordinates of points along the tracks in events of interest, and discards measurements of uninteresting points on the film. In the time left over after these control and filtering functions are performed, the computer analyzes either events that have just been measured or events that have been measured on some previous occasion.

Data Analysis

The output from all of the measuring devices just described serves as input for data analysis programs which are designed to identify the observed nuclear events and to study distributions of various properties of these events. The first step in this analysis process is to reconstruct the event in three-dimensional space; at least two pictures are taken of each event, so one may perform this reconstruction quite easily. After the tracks have been located in space, their curvature is computed. From this curvature and from the intensity of the magnetic field in the neighborhood of the track, one can deduce the momentum of the particle which made the track.

After these calculations have been made, all the tracks making up the event are considered simultaneously. Now, by using the constraints of conservation of energy and momentum, one can often deduce the mass of the particles involved and, hence, their identity. This step is called kinematic analysis.

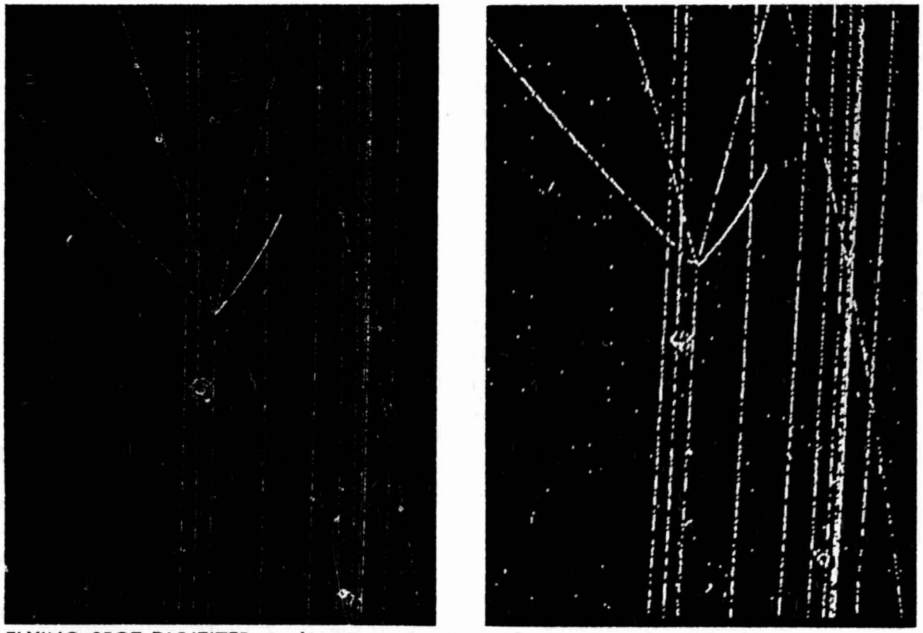
When all events making up an experiment have been analyzed, the information just computed is stored on a magnetic tape. This tape serves, in turn, as input for a series of statistical analysis programs.

Two sets of bubble chamber data analysis programs are used at the Berkeley Laboratory: the PANAL–PACKAGE–EXAMIN system developed in the Alvarez Group, and the FOG–CLOUDY–FAIR system developed by Howard White's Data Handling Group. Each of these systems involves well over 100,000 computer instructions, and together they consume about 60% of the computation time used on Berkeley computers.

The Berkeley Laboratory pioneered the development of automatic measurement and analysis systems for bubble chamber data. Programs and hardware developed at the Berkeley Labortatory are used in high energy physics laboratories all over the world.

Accelerator Design

Computers are also playing an increasingly important role at Berkeley in the design of particle accelerators. They are used to compute the magnetic fields generated by given magnet designs and to trace the orbits of particles through these fields. Al Garren, of the Theoretical Physics Group, has developed an accelerator-oriented language which allows its user to evaluate very quickly a particular accelerator configuration.



FLYING-SPOT DIGITIZER, working in conjunction with computers, has helped to automate the bubble-chamber film analysis task at Berkeley. Picture at left shows a typical pi-minus interaction in a bubble chamber. The negative of this photograph is scanned by the FSD, and the digitized information is sent directly into a computer. The picture at right was "computed" from the digitized information supplied by the FSD.

Part VII: The World of the Future

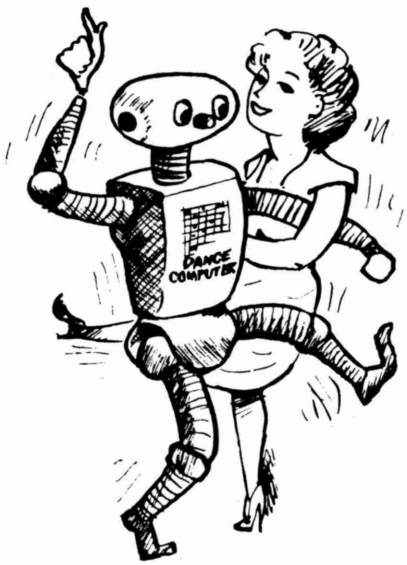
The computing machines which will be installed in the United States starting in 1965 (an early forerunner of these was the Control Data 6600, delivered to the Livermore Lab two months ago) will be quite different from the machines that have been installed to date. They will be faster by a factor of ten or so; their memories will be up to four times as large; and they will be equipped with much faster and more sophisticated input-output devices than were their predecessors. These hardware changes will have a profound effect on the ways in which computers are used.

At computer installations where the problem load involves a fair amount of input-output (such as LRL's Berkeley Laboratory), the installation of a computer whose arithmetic unit is ten times faster does not necessarily mean that one will now be able to do ten times as many problems in the same time period. If we attempt to run our problems in the conventional way, we will spend most of the time on our computer waiting for input-output tasks to be accomplished; the central processing unit will be idle for a large share of the time. One way to remedy this situation would be to acquire input-output devices which were faster than existing ones by a factor of ten. Unfortunately, such devices are not readily available, so we must seek another solution to this problem.

Multiprogramming

The solution that will be adopted at most large computer installations is known as Multiprogramming. This concept involves having the programs for several problems in memory at once. These problems all share the computer's central processing unit, but each one is assigned its own peripheral input-output devices—tape drives, for example. If we have a reasonable mix of problems, and if we are clever in writing a program to allocate our computer components among these problems, we can expect to keep a reasonable number of machine components busy at the same time; one code may be using the central processing unit, for example, while three others are simultaneously performing tape input-output functions. Under such a system, the elapsed time for doing a set of problems will be substantially reduced.

Most of the machines which are scheduled for delivery starting in 1965 have hardware capabilities which allow them to be multiprogrammed. Such hardware capabilities include large memories (we must have enough space to get several codes into memory at once), several input-output channels (we must allow several programs to be performing input-output tasks at the same time), and memory protection devices (we must prevent the program which has control of the central processing unit from destroying other programs which share the memory).



The programs which control the scheduling of problems on the computer and the allocation of computer hardware are called Monitor Systems. A limited multiprogramming monitor system, called Diprogramming, has been in operation on a 7094 computer at the Berkeley Laboratory for about six months. It was described in the August issue of the MAGNET. A more sophisticated Multiprogramming Monitor System for the Control Data 6600 is currently under development at the Livermore Laboratory.

In the future, Multiprogramming Monitor Systems will be provided by the manufacturers of all major computers. Such systems are already available for the Honeywell 800 and UNIVAC 1107 Computers.

An important consequence of the availability of multiprogramming systems on new computers involves the way in which card input and printer output will be handled. The user who now wishes to submit a problem for solution on a large-scale computer turns in a deck of cards with his program on it; this deck of cards is then combined with several other decks and taken to a small auxiliary computer such as an IBM 1401, where the cards are transcribed onto a reel of magnetic tape. This tape is then transferred from the 1401 computer to the large-scale computer, and the problems on the tape are done in sequence. The large-scale computer, instead of printing the answers on its on-line printer, writes them out on another reel of magnetic tape. When all the problems in this sequence have been completed, this printer tape is transferred to a 1401 computer, which performs the actual printing. The reason for all this tape manipulation lies in the sequential way in which problems are done on current large-scale computers. If one were to read cards on-line or print on-line, one would idle all other components of the machine—an intolerable condition.

SPOOLing

When a mulitprogramming system is available, this whole situation changes; it now becomes perfectly feasible to print on-line and read cards on-line and, at the same time, to utilize many other components of the computer. This type of activity is called SPOOLing, which stands for Simultaneous Peripheral Output On-Line. In the bright world of the future, the user will simply insert his problem deck into a card-reader attached to a large-scale computer; the computer will then read his cards in and store them on an auxiliary storage device, such as a magnetic disk. When the problem reaches the head of the queue and when memory space becomes available for it, the Monitor System will load it into the memory and begin executing it, probably in conjunction with several other programs. Printer output will be written on a disk; as soon as an on-line printer becomes available, the output will be printed. The availability of this SPOOLing capability should dramatically decrease the elapsed time between the submission of a program deck and the return of printed output.

The possibility of more intimate man-machine relationships is another important consquence of multiprogramming systems. Under ideal conditions, the user of current hardware and systems may expect a two-hour delay between the time he submits a problem to the computer and the time he gets his output

back. Quite often, when he looks at his output, he discovers that he has omitted a comma in a FORTRAN statement, or that there has been a keypunching error in the preparation of his data, or that the hypothesis he was testing in his program is false and that he now wishes to try another one. In any of these cases, the user may require only a few minutes of looking at his output before he is prepared to submit another program. After submitting the program again he must, of course, wait another two hours. This is a very inefficient way of using people. One possible solution to this problem would be to provide each user with his own computer; he could sit at the console of his machine and obtain immediate turn-around on his problem, since there would be no people queuing up in front of him. This, of course, is not a very efficient way of using computing machines—even if one had enough money to buy that many machines.

User Consoles

A compromise solution will be found in a variety of user display and inquiry consoles that will be available with the next generation of computing machines. These consoles, which may be installed at remote locations, will be connected directly to the central computer system. The simplest of these consoles is a typewriter or teletype machine — supplemented, perhaps, by a card reader. More complicated consoles include a cathode ray tube display system, which can be used for either graphical or alphanumeric output. The user will employ these consoles as debugging devices or as quick-response mechanisms in the solution of problems.

The Livermore OCTOPUS System, which was described in a previous MAGNET article (September '64), will employ several consoles, each consisting of a teletype machine and a small pen-and-ink plotter. In Berkeley, consoles of two types will be used: typewriter and card reader stations will be used for program debugging, and CRT plotters equipped with keyboards and light pencils will be used as on-line physics consoles.

New Problems

The increased computing power of future computer generations, and the increasing sophistication of programming systems available for them, will allow the computer user to attack larger problems than ever before. For example, Livermore scientists will be able to undertake much more detailed simulation of weapons systems, and Berkeley scientists will tackle the analysis of larger experiments than in the past. Moreover, it will now be possible to solve effectively some entirely new problems. One of these which has been much discussed is the pattern-recognition problem for bubble chamber events. Over the past few years the Berkeley Laboratory has developed some extremely speedy devices for measuring the location of points on bubble chamber film. However, present systems demand that all film be scanned by a human scanner in advance of measurement. This human scanner is the pattern-recognition element in the system. In the future, it is hoped that this pattern-recognition role can be taken over by computers; such systems are currently under development by Howard White and his co-workers and several other groups at the Berkeley Laboratory.

At present, when a computer user wants to solve a specific mathematical problem he is forced to spend a great deal of time matching his particular version of that problem with the available numerical techniques, and with computer programs which implement those techniques. He must match the appropriate code to his problem. This is often a fairly time-consuming process. In the future, it is hoped that much of this work can be taken over by the computer; for example, the user may someday merely submit his differential equation for solution, specifying the range over which the solution is needed and the required accuracy. The computer would then analyze the differential equation, select an appropriate solution technique, and pick out of its library a code which implements that technique. Here, the computer would take over some of the functions not only of the programmer, but also of the numerical analyst.

Advances in list-processing and in the manipulation of symbols by computers should make possible the creation of routines capable of performing many of the algebraic manipulations which now constitute something of a national sport among theoretical physicists. The least we may confidently expect of such routines is that they will be capable of the formal manipulation of infinite series and of the simplification of algebraic expressions. These manipulations will probably be accomplished with the assistance of the display consoles mentioned earlier.

Exotic Applications

Many new and exciting computer applications can be expected outside of the Laboratory's research areas during the coming years as well. For example, computer-assisted engineering design applications, already being written, will permit a mechanical engineer to interact very closely with a computer. In this application, the computer will play the role of a sort of high-speed calculator and draftsman.

Somewhat farther off is the development of fully-automatic, high-quality mechanical translation from one natural language to another. This development will surely be stimulated by the increased speed and memory size of the computers now becoming available.

Effective computer-assisted weather forecasting is probably somewhat closer than effective mechanical translation. The availability of high-speed computers, together with the presence of increasing amounts of data from weather satellites, should make this development possible in the near future.

Finally, the use of computers in teaching, to extend existing programmed learning techniques, will be extremely important.