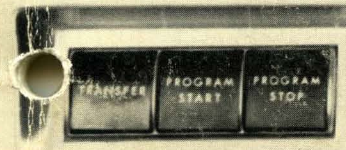
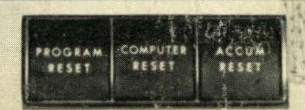
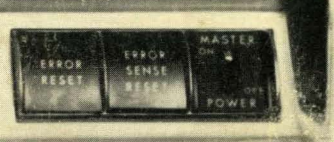
RICHARD V. ANDREE
ASSOCIATE PROFESSOR OF MATHEMATICS
UNIVERSITY OF OKLAHOMA

Programming the IBM 650 MAGNETIC DRUM COMPUTER AND DATA-PROCESSING MACHINE







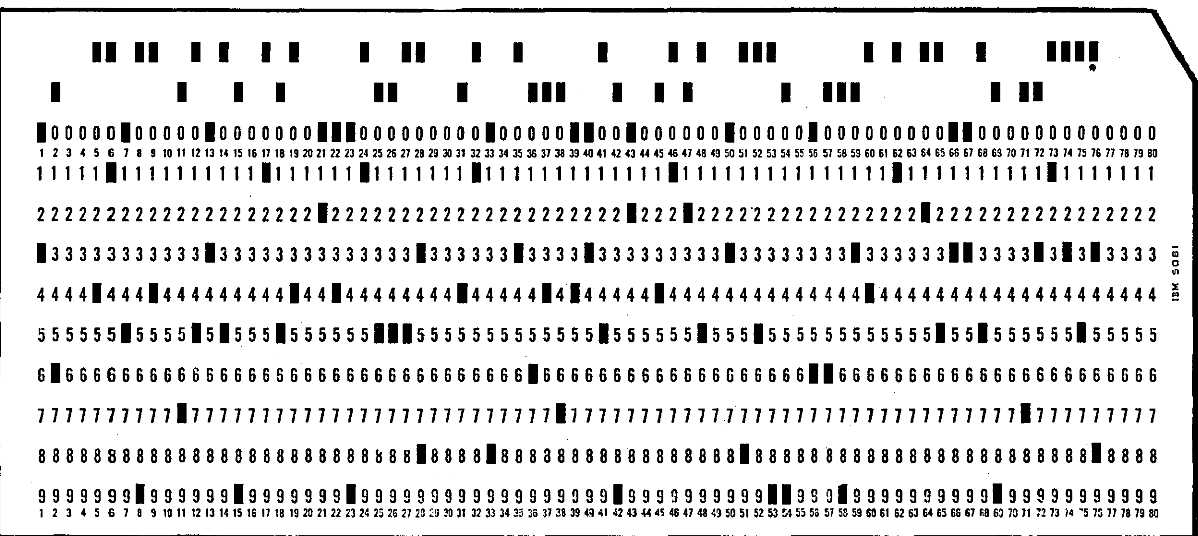
HOLT-RINEHART-WINSTON
PUBLISHERS · NEW YORK

RICHARD V. ANDREE
ASSOCIATE PROFESSOR OF MATHEMATICS
UNIVERSITY OF OKLAHOMA

Programming the IBM 650 MAGNETIC DRUM COMPUTER AND DATA-PROCESSING MACHINE

HOLT, RINEHART AND WINSTON, INC.
383 MADISON AVENUE, NEW YORK 17, N. Y.

Dedication



Copyright © 1958 by Richard V. Andree

Library of Congress Catalogue Card Number: 58:12522

Printed in the United States of America

20416-0218

PREFACE

This book is a first introduction to the programming of the IBM 650 computer. Most of the techniques involved are applicable to any medium-speed or high-speed computer. There are important advantages in programming for a specific machine, available to the reader, rather than with a mythical machine on which it is impossible to check out programs once they are written. The student learns much while debugging his program, and enjoys the thrill of finally seeing it work. These notes can be used whether or not a 650 is available, and they have been used both ways with marked success. Students use flow charts and write actual programs from the very beginning. After half a dozen class sessions, they write simple programs with surprising ease.

These notes are designed to take the neophyte and advance him to the stage where he can use the several available IBM programmer's manuals. Those manuals, are, after all, designed as reference manuals, not as textbooks. These notes provide a text, with suitable problems, and are not intended as a reference manual. The first four chapters are self-contained. Chapters 5 to 9 begin to lean slowly more and more upon the published IBM manuals, which the student is encouraged to consult.

The main purpose of these notes then, is to provide a teachable text on introductory programming, usable either for self-study or for classroom work. They have been used in intensive one-week courses, in three- and four-week courses, and in semester-long courses. They have been used by graduate engineers and by high-school students.

After the first four chapters are completed, the student can be turned loose on his own, although the author much prefers to have the student finish Chapter 5 on the use of SOAP rather than forcing him to write in machine language using five-ten optimization.

Chapter 6, which emphasizes overall principles stressed elsewhere in the book, concludes with a fairly extensive set of real-life problems, the programming of which may develop considerable sophistication on the part of the reader. Chapter 7 discusses the advantages, and also the disadvantages, of interpretive systems and uses the Bell Interpretive System (FLOPS) as an example. Chapter 8 is devoted to compilers, with particular mention of IT and For transit. Since excellent manuals are now (1958) available on the Perlis compiler, IT, and on For transit, no attempt is made to discuss these two all-important techniques in complete detail. Instead, the reader is given enough insight into For transit to overcome his first fear and to convince him that learning For transit is worthwhile; he is then encouraged to program, turning to the For transit manual as additional help is needed. Chapter 9 is devoted to several odds and ends with which the student should become familiar if he is seriously interested in programming.

A convenient teaching device is to make a series of "bug cards" for a program with which the students are familiar (say Problem 9, Set 1-15). These typical errors can be punched on one-per-card load cards. It is convenient to punch the words "Bug Card" and the problem number in the remarks columns for future use. If a single bug card is placed at the end of the program deck, the false instruction will be loaded over the top of the correct instruction. The bug card can then be removed without disturbing the main body of the program.

The author will sincerely welcome discussion from users of this book concerning errors, or portions that can be improved in the next edition.

It is my sincere hope that *you* will find this brief text both helpful and enjoyable.

ACKNOWLEDGMENTS

This list of acknowledgments should include the names of hundreds of students and dozens of colleagues who used these notes in various preliminary forms. That seems impractical. It would, however, be ungracious not to mention individually those people who have contributed most.

Mr. Frank E. McFarlin, formerly of Oklahoma State University and now with IBM Computer Research and Design Group, Endicott, has helped with the planning of these notes from the beginning and has read the entire manuscript, offering numerous excellent suggestions.

Dr. John Hamblen formerly Director of the 650 Computer Laboratory at Oklahoma State University (currently at the University of Kentucky) and Dr. William Viavant, Director of Scientific Computations at the University of Oklahoma have each given encouragement and assistance far beyond the call of duty.

Dr. Melvin Shader, Manager, IBM University and Research Institute Program has graciously read the entire manuscript. Many improvements have resulted from his thought-provoking suggestions.

Dr. H. A. Meyer, University of Florida and Dr. G. E. Forsythe, U. C. L. A., each made verbal suggestions for which the author is sincerely grateful.

The award for efficiency and elegance in real-life programming goes to the author's wife, who never goes into a loop when her instructions are modified in unanticipated ways. Without her understanding cooperation this book would have been impossible.

To each of the above, and to other colleagues and students, the author expresses sincere thanks.

Norman, Oklahoma
September, 1958

R. V. A.

CONTENTS

CHAPTER 1 — SIMPLE PROGRAMMING 1

1–1 Introduction · 1–2 What a Computer Is, and Is Not · 1–3 The IBM 650 · 1–4 The IBM Card · 1–5 Word · 1–6 Drum · 1–7 Accumulator · 1–8 Distributor · 1–9 Instructions · 1–10 A Problem · 1–11 OPeration Codes · 1–12 Logical Test – NZU · 1–13 Multiplication · 1–14 New Operation Codes · 1–15 Flow Charts.

CHAPTER 2 — THE 650 CONSOLE 24

2–1 Read and Punch · 2–2 Power Control Switch · 2–3 Display Lights · 2–4 Flow of Instructions · 2–5 Internal Checking of the 650 · 2–6 Load Cards · 2–7 Board Wiring for Load Cards · 2–8 Operating and Checking Lights · 2–9 Address Stop · 2–10 Program Start · 2–11 Cookbook Directions.

CHAPTER 3 — MORE ADVANCED PROGRAMMING 38

3–1 Improving Speed · 3–2 Stepping Instructions · 3–3 Looping and Stepping · 3–4 The Count Box · 3–5 Another Terminating Technique · 3–6 Branching on a Code Number (Optional) · 3–7 Scaling · 3–8 Scaling in Multiplication · 3–9 Division · 3–10 $\frac{(A + B) \cdot C}{D} = T$ · 3–11 Optimum Programming · 3–12 Index Registers (Optional).

CHAPTER 4 — EASY PROGRAMMING VIA SUBROUTINES 60

4–1 Accuracy of Computed Results · 4–2 Subroutines · 4–3 Partial List of Available Programs and Subroutines · 4–4 A Problem · 4–5 A Word to Those Who Will Have Others Write Their Programs · 4–6 Floating-point Arithmetic.

CHAPTER 5 — SOAP 72

5–1 SOAP (Symbolic Optimum Assembly Program)

CHAPTER 6 — A RE-EXAMINATION OF PRINCIPLES 76

6–1 Sophistication in Programming · 6–2 Flow Charting · 6–3 Loops · 6–4 Free Data · 6–5 Economy in Loading the Program · 6–6 Program Errors – Debugging · 6–7 Special Traces and Debugging Routine · 6–8 Experience.

CHAPTER 7 — **INTERPRETIVE SYSTEMS** 89

7 – 1 Interpretive Systems · 7 – 2 Bell System (FLOPS) · 7 – 3 Read and Punch · 7 – 4 Additional Bell Instructions · 7 – 5 Loop Operations · 7 – 6 Example 1 · 7 – 7 Example 2 · 7 – 8 Example 3 · 7 – 9 Summary of Bell Operations · 7 – 10 Remarks on Interpretive Systems.

CHAPTER 8 — **COMPILERS** 99

8 – 1 Easy Programming · 8 – 2 IT · 8 – 3 For Transit · 8 – 4 Programming in For Transit.

CHAPTER 9 — **ODDS AND ENDS** 104

9 – 1 Common Courtesy · 9 – 2 Sources of Information · 9 – 3 Organization of a 650 Laboratory · 9 – 4 The 650 Library · 9 – 5 Suggested Group Demonstrations · 9 – 6 Subroutines and Programs for Your Library · 9 – 7 Well.

INDEX 108

SAMPLE PROGRAMMING SHEETS 110

CHAPTER 1

SIMPLE PROGRAMMING

1-1. INTRODUCTION.

There is probably no easier way to lose the friendship of a mathematician than to hand him a pencil and paper along with the remark, "You are a mathematician, you keep score." Mathematics is the art of *avoiding* computation, and it is partly because digital computers help avoid large amounts of computation that they interest mathematicians.

Instead of making fairly general remarks about computers, these notes will examine one particular computer, namely, the IBM 650 Magnetic Drum Computer, in considerable detail. To a large extent, the techniques illustrated are applicable to other computers.

After one week of classroom course work, or approximately three hours of study from this book, you should be able to write simple programs for the IBM 650! At the end of a month's study, you should be able to do fairly complicated programs on your own. It will, of course, be desirable to have the guidance of a more experienced programmer for some time, if you are to learn most efficiently. One important thing to learn is to think in a manner compatible with the machine's optimum abilities. In evaluating $X^4 + 3X^3 - 5X + 2$ for example, it is much quicker for the machine to compute

$$X\{X[X(X+3)]-5\}+2$$

than for it to compute

$$(X^4)+3(X^3)-5(X)+2.$$

Many persons will never do a great deal of programming themselves, but must understand the machine well enough to direct intelligently those who will be doing the actual programming for their problems. Chapters 1 to 6 will suffice for such purposes, but experience shows that even readers who do not plan to do their own programming will probably read the entire book once they get started. The 650 is like a virus which gets into the blood stream—it is hard to shake loose once you are inoculated. (FAIR WARNING!) These notes may be used either for a short intensive course, or for a semester-long course. They are also excellent for self-study. It is much easier to study computers if you have a computer available; but it is quite possible to have an elementary course in programming in the complete absence of an actual computer. For those who do have a 650 available, a series of programs will be given, along

with certain "bug cards" which will make the type of mistakes in the program which you, as a beginning programmer, are most apt to make. Your problem will be to find, with the aid of your instructor, just where the mistakes are. The debugging process takes a good deal of machine time in many programs, particularly in the type of one-time-through programs which occur often in research applications.

1-2. WHAT A COMPUTER IS, AND IS NOT.

A computer is *not* a giant brain, in spite of what some of the Sunday supplements and science fiction writers would have you believe. It is a remarkably fast and phenomenally accurate moron. It will do exactly what you tell it to do—no more, no less. There are about 60 different instructions you can give an IBM 650. It can add, subtract, multiply, and divide. It can determine whether or not a given 10-digit number is zero, and it can determine whether or not a 10-digit number is negative. It can also determine whether or not any given digit in a 10-digit number is an 8 or a 9.

Actually, it is somewhat misleading to say that it can add, subtract, multiply, and divide, since it does its multiplication by repeated addition and it does its division by repeated subtraction, just as one does on a desk calculator. Actually, it does not even subtract. When you tell the machine to subtract, it first takes the complement of the number and then adds the complement; so, in the last analysis, a computer adds!

Let us see what it is that distinguishes a computer from a desk calculator. In the first place, in the desk calculator, an auxiliary piece of paper is usually used to store initial data or numbers and intermediate results, and either paper or the operator's memory is used to keep track of the sequence of steps to be performed. On a computer, both data and the sequence of instructions are stored in the storage or memory of the machine, although not necessarily in the same type of storage. For example, data might be stored on a magnetic drum and the instructions punched on paper tape. In a "stored-program" machine, both data and instructions are stored in the same memory, and a given memory location may be used to store either data or instructions. The 650 is a stored-program machine and, as will be illustrated later, such a machine can be programmed to generate its own instructions. This is possibly the most important identifying characteristic of a stored-program computer. The main difference between a computer and a desk calculator is that once the computer is started it will follow a list of instructions without further guidance, since it is merely necessary to change data. Another advantage of most of the large and medium-scale computers is their speed of calculation. For example, the IBM 650 can add two 10-digit numbers in $\frac{1}{125,000}$ of a minute, and can multiply two 10-digit factors, obtaining a 20-digit product, in $\frac{1}{6000}$ of a minute. A more meaningful example is to note that the 650 can extract the square root of a 10-digit number, accurate to 9 or 10 significant digits, in about 0.152 seconds. In so doing, the 650 does about 60 program steps ranging from storing a 10-digit number to dividing a 20-digit number by a 10-digit number.

1-3. THE IBM 650.

This book contains an introduction to the programming of the IBM 650 Magnetic Drum Computer (data processing machine). No previous knowledge of computing machines, nor of IBM equipment, is assumed.

Figure 1.1 shows the three units which comprise the basic 650. The center is the 650 *Console,* which houses the computing units. On the right is the 533 input-output unit, through

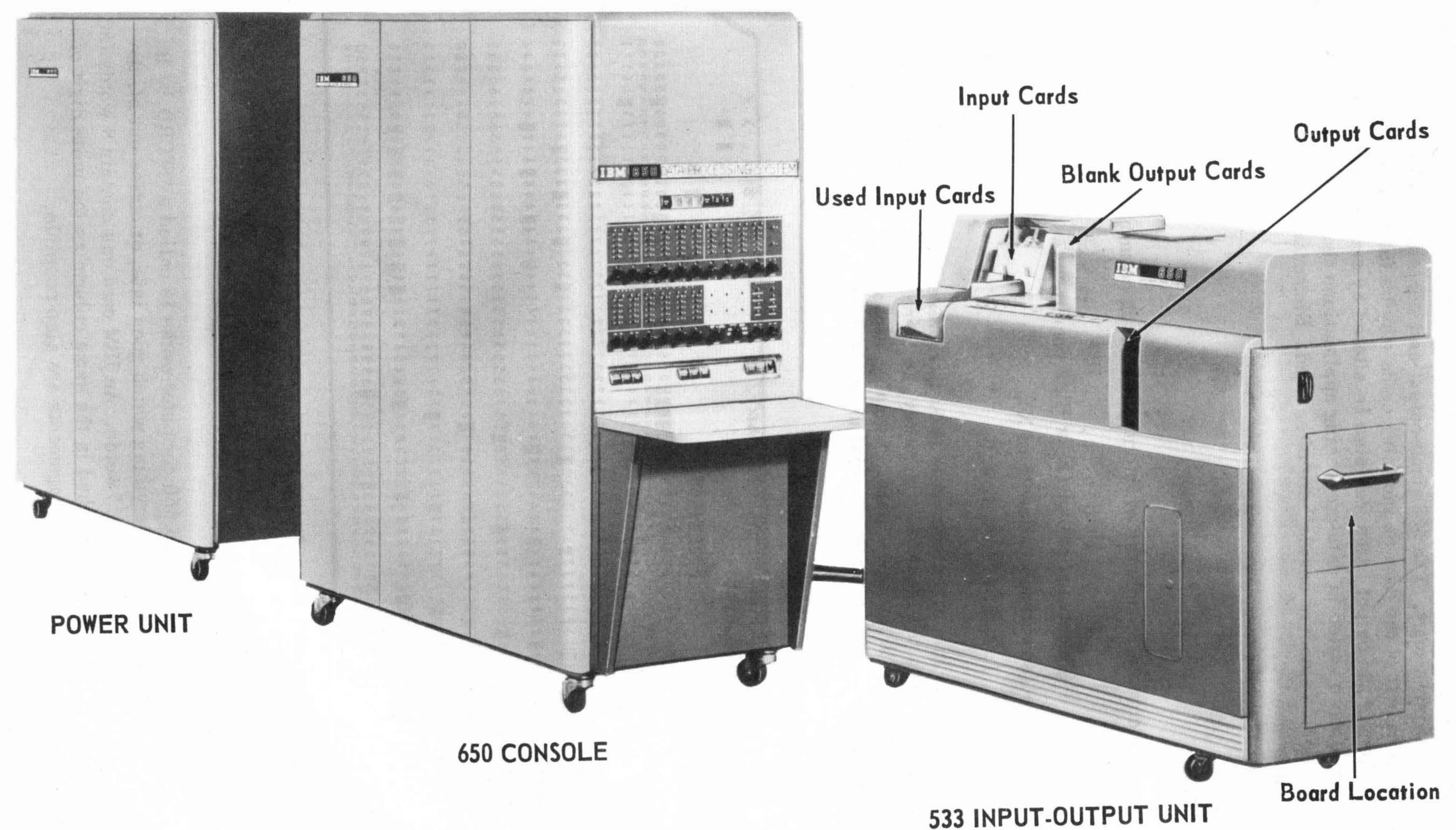


FIGURE 1-1
MAGNETIC DRUM DATA-PROCESSING MACHINE
TYPE 650

which punched cards are read into the 650 and from which results are punched out. The unit in the background is the power unit. The 533 input-output unit contains a wired control panel, by means of which all or part of the data on incoming cards may be fed into the 650 in various combinations. The form of the output cards may also be altered by control-panel wiring. Control-panel wiring is more varied in data-processing applications than in scientific computation, where two or three "standard boards" are often used. Elementary comments on board wiring will be given as needed.

1-4. THE IBM CARD.

Each IBM card may contain 80 columns of information. Each column may be punched in any of 12 positions:

+ (sometimes called 12, or High punch),

– (sometimes called 11 or X),

0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

A combination of two punches in a given column is used to represent alphabetic data. A glance at the sample card gives the alphabetic code key, so there is no reason to describe it verbally. The 650 computes with numerical data. Normally, alphabetic information may be transferred from input cards to output cards. Special usage is described under the SOAP programming system (Chapter 5).

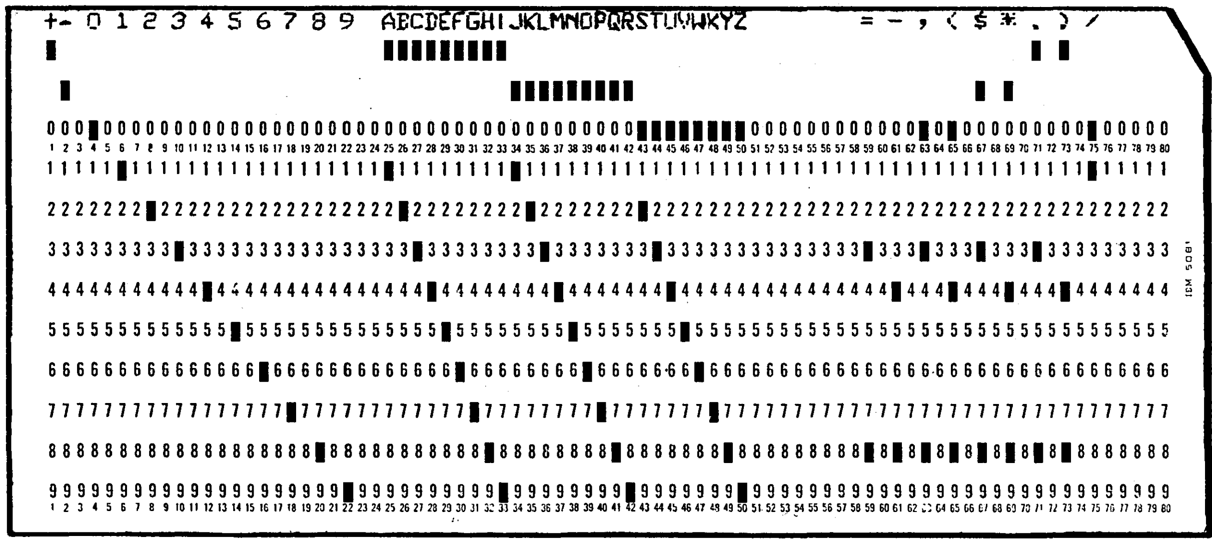


FIGURE 1-2

1-5. WORD.

The basic numerical unit with which the 650 computer works is called a "WORD." It consists of 10 decimal digits and a sign. Everything which goes into, or comes out of, the 650 will be in the form of "10-digit and a sign" words. An IBM card can contain 8 words (80 digits). The sign of each word is usually punched in the same column as the units digit of the word (a double-punched column). Other arrangements are also possible.

1-6. DRUM.

The storage (formerly called "memory") unit of the 650 is a cobalt-nickel-plated cylinder about 4 inches in diameter and 16 inches long called a *Magnetic Drum* or just *Drum*. The drum revolves at about 12,500 rpm. Information is stored on the drum in the form of magnetized spots. The drum is divided into 2000 word locations, each word containing 10 digits and a sign. It is convenient to think of the surface of the drum as divided into 2000 rectangles, each containing 10 digits and a sign. Each rectangle is then a "drum location." Figure 1.3 shows a layout of a portion of the drum. Each location is given a "street address" or "drum location address" which is a 4-digit number running from 0000 to 1999. Words may be thought of as stored in 40 bands of 50 words each, running around the drum. Since the drum rotates at

	0000	0050	0100	0150	0200	0250	0300	0350	0400	0450	0500	0550	0600
READING	[illegible]	51	01	51	01	51	01	51	01	51	01	51	01
	02	52	02	52	02	52	02	52	02	52	02	52	02
	03	53	03	53	03	53	03	53	03	53	03	53	03
	04	54	04	54	04	54	04	54	04	54	04	54	04
	05	55	05	55	05	55	05	55	05	55	05	55	05
	06	56	06	56	06	56	06	56	06	56	06	56	06
	07	57	07	57	07	57	07	57	07	57	07	57	07
	08	58	08	58	08	58	08	58	08	58	08	58	08
	09	59	09	59	09	59	09	59	09	59	09	59	09
	10	60	10	60	10	60	10	60	10	60	10	60	10
	11	61	11	61	11	61	11	61	11	61	11	61	11
	12	62	12	62	12	62	12	62	12	62	12	62	12
	13	63	13	63	13	63	13	63	13	63	13	63	13
	14	64	14	64	14	64	14	64	14	64	14	64	14
	15	65	15	65	15	65	15	65	15	65	15	65	15
	16	66	16	66	16	66	16	66	16	66	16	66	16
	17	67	17	67	17	67	17	67	17	67	17	67	17
	18	68	18	68	18	68	18	68	18	68	18	68	18
	19	69	19	69	19	69	19	69	19	69	19	69	1
	20	70	20	70	20	70	20	70	20	70	20	70	2
	21	71	21	71	21	71	21	71	21	71	21	71	
	22	72	22	72	22	72	22	72	22	72	22	72	
	23	73	23	73	23	73	23	73	23	73	23	73	
	24	74	24	74	24	74	24	74	24	74	24	74	
	25	75	25	75	25	75	25	75	25	75	25	75	
	26	76	26	76	26	76	26	76	26	76	26	76	
PUNCHING	27	77	27	77	27	77	27	77	27	77	27	77	
	28	78	28	78	28	78	28	78	28	78	28	78	
	29	79	29	79	29	79	29	79	29	79	29	79	
	30	80	30	80	30	80	30	80	30	80	30	80	
	31	81	31	81	31	81	31	81	31	81	31	81	
	32	82	32	82	32	82	32	82	32	82	32	82	
	33	83	33	83	33	83	33	83	33	83	33	83	
	34	84	34	84	34	84	34	84	34	84	34	84	
	35	85	35	85	35	85	35	85	35	85	35	85	
	36	86	36	86	36	86	36	86	36	86	36	86	
	37	87	37	87	37	87	37	87	37	87	37	87	
	38	88	38	88	38	88	38	88	38	88	38	88	
	39	89	39	89	39	89	39	89	39	89	39	89	
	40	90	40	90	40	90	40	90	40	90	40	90	
	41	91	41	91	41	91	41	91	41	91	41	91	
	42	92	42	92	42	92	42	92	42	92	42	92	
	43	93	43	93	43	93	43	93	43	93	43	93	
	44	94	44	94	44	94	44	94	44	94	44	94	
	45	95	45	95	45	95	45	95	45	95	45	95	
	46	96	46	96	46	96	46	96	46	96	46	96	
	47	97	47	97	47	97	47	97	47	97	47	97	
NOT USED FOR TABLES	48	98	48	98	48	98	48	98	48	98	48	98	
	49	99	49	99	49	99	49	99	49	99	49	99	

FIGURE 1-3
MAGNETIC DRUM LAYOUT

12,500 rpm, the largest possible access time—i.e., time to read out one word of information from a specific location—would be .0048 second, with the average time being much less if a sensible programming procedure is followed. A word stored on the drum remains there until another word is stored on top of it, at which time the old word is first automatically canceled by the new word. Words remain on the drum even if the power is cut off.

1-7. ACCUMULATOR.

The arithmetical unit is a 20-digit accumulator (like a 20-digit desk calculator). For convenience, it is thought of as divided into three parts, the upper half of the accumulator (10 digits), the lower half of the accumulator (10 digits), and a sign. In some operations it is

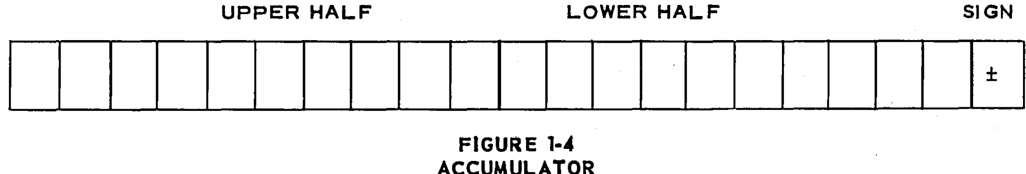


FIGURE 1-4
ACCUMULATOR

desirable to use only the upper half of the accumulator; in some, to use only the lower half; and, in some, to use both parts. However, the machine *always* behaves as if the entire 20-digit accumulator were used, with zeros usually being placed on the unused section. A result that is in the accumulator will remain there until it is erased by a "reset" operation, or altered by an arithmetical operation, or until the power is shut off. (Capacitor storage which is used in the accumulator does *not* "hold"—i.e., remain indefinitely in the absence of regenerative power—as does the magnetic storage of the drum.)

1-8. DISTRIBUTOR.

The distributor has a capacity of one word (10 digits and a sign). Any word transferred between the drum and the accumulator passes through the distributor and *remains there until it is replaced by* another word. A word located in the distributor is available more quickly

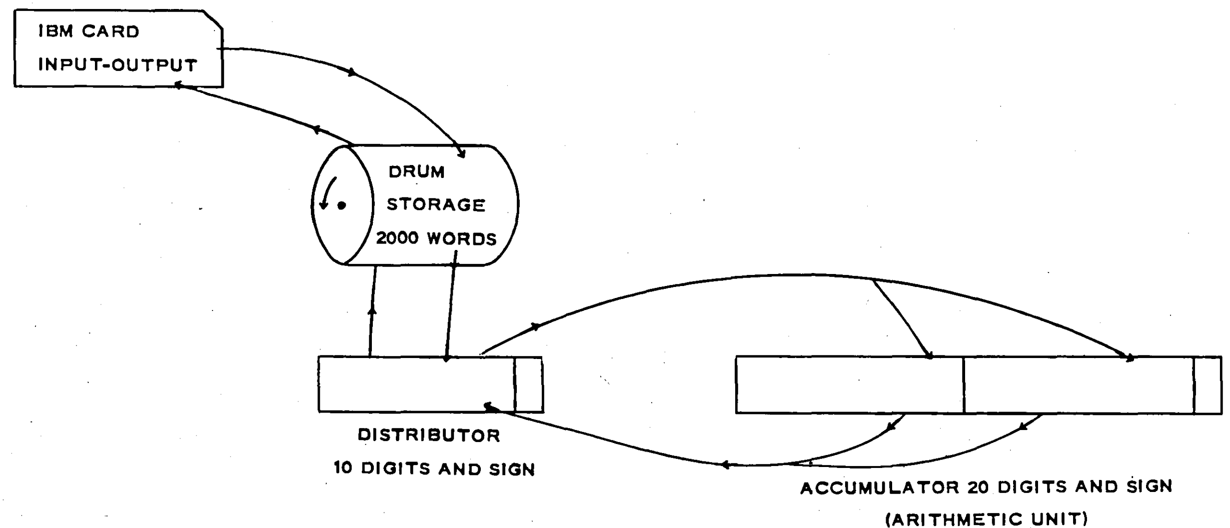


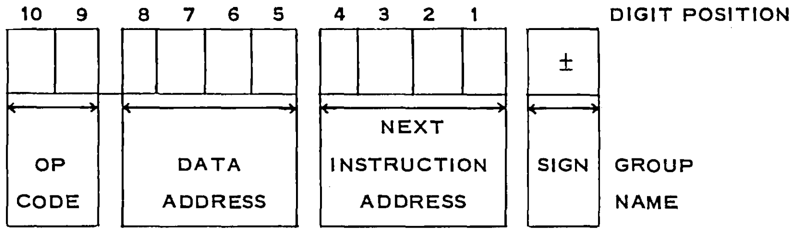
FIGURE 1-5

than one on the drum, and should be used when appropriate. A word will remain in the distributor until another word is transferred from the drum to the accumulator, or from the accumulator to the drum, or until the power is shut off.

The schematic diagram in Figure 1.5 will help fix this important "routing pattern" in your mind.

1-9. INSTRUCTIONS.

The IBM 650 is a stored-program machine. The instructions are stored on the drum in the same form as data. Since the only thing which the drum can store is a set of 10-digit signed numbers, the instructions to the machine are coded as 10-digit numbers. The 10 digits of an instruction are sectioned into three groups as shown:



In general:

OPeration: The two digits on the left (positions 10,9) tell the machine what operation to perform and where to perform it. (Example: 65 means "Reset the entire 20-digit accumulator to zero, and then add the contents of that storage location specified by the data address to the lower half of the accumulator.")

Data **A**ddress: The four digits in positions 8–5 (**D**ata **A**ddress) usually give the storage location needed in performing the **OP**eration code of the instruction.

Instruction **A**ddress: The four right-hand digits (positions 4–1) usually give the storage location of the next instruction,—i.e., tell the machine where to go for its next instruction after the **OP**eration has been performed.

Sign: The sign of the instruction (±) is quite useful in connection with subroutines, but will not be discussed until needed. A sign *must appear* on each instruction in storage, or the machine will stop, but it is of no importance here what sign appears. For the present we shall make them all +. The sign has no effect on the operation code.

1-10. A PROBLEM.

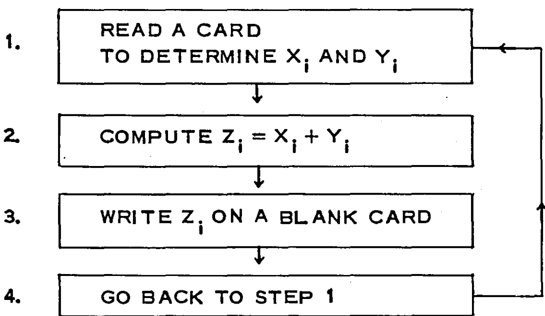
Let us assume that a deck of data cards, data deck 1, contains 600 IBM cards. The first card word (Cols. 1–10) consists of a 10-digit signed number, of which the first three digits are always zero; hence, the first card word contains a 7-digit number, X_i. The second card word (Cols. 11–20) contains a 10-digit word of which the six high-order digits are all zeros; hence, the second word contains a 4-digit number, Y_i, with its sign. The other six words on the card will not be discussed currently, since only the first two words are used in this example.

Although the 650, itself, receives its instructions through numerical codes, it is convenient to have a symbolic letter code (mnemonic) which suggests the operations implied. There are two of these code systems in common use. One is the "old code" found in the old 650 Manual, and the second is the SOAP (Symbolic Optimum Assembly Program) code. Since

WORD 1	WORD 2	WORD 3	WORD 4	WORD 5	WORD 6	WORD 7	WORD 8
000xxxxxxx	000000yyyy						

the reader will almost certainly wish eventually to learn to program using SOAP, we shall adopt these symbols throughout. The main difference between the two codes is that each SOAP operation code contains exactly three letters, while the old code has from two to five letters.

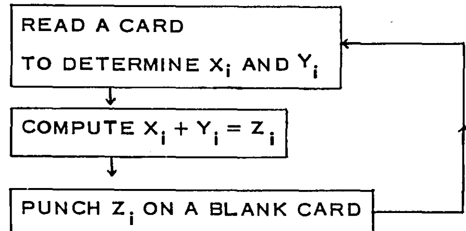
EXAMPLE 1.
Compute the sum $Z_i = X_i + Y_i$ for each of the 600 pairs of X_i, Y_i which are given in deck 1, words one and two. How would you explain to someone what you wish him to do, using a desk calculator, if you were giving him this assignment? Roughly, you would probably tell him to read the card and determine X_i and Y_i. Next, he should compute $X_i + Y_i = Z_i$ and mark Z_i on a blank card, then go to the next card and repeat the instructions.



This might be adequate if the person knew how to add on a desk calculator.

A more detailed set of instructions might be: Read a card for X_i and Y_i; clear (reset) the desk calculator (accumulator) to all zeros; add X_i into the accumulator; add Y_i into the accumulator; read out the answer Z_i from the accumulator and remember the answer long enough to write it in the desired columns of a blank card. Repeat these instructions with the next card.

This detailed sequence of instructions is approximately what one does on a 650 computer. The flow chart giving the general overall idea of the program is similar.



When writing a program, in addition to writing down instructions (10-digit words), it is also necessary to keep track of the drum location in which the instruction is stored. A form similar to that which the student has on his IBM 650 Planning Chart (Form 22-6151-2) is used for this. We shall arbitrarily choose to locate our first instruction in drum location 0500 and each successive instruction in a location 10 larger than its predecessor, e.g., 0500, 0510, 0520, 0530....

DRUM LOCATION OF INSTRUCTION	INSTRUCTION			COMMENTS
	OPERATION	DATA ADDRESS	NEXT INSTRUCTION ADDRESS	
0500	RCD 70	1851	0510	

This first instruction RCD (Read **C**ar**D**, or, sometimes, RD1—**R**ea**D** input one) 70 1851 0510 is to be located in drum location 0500. It tells the machine to read a card (8 words, 80 digits) and to store the contents of these 8 words in drum locations 1851, 1852,..., 1858; then to go to drum location 0510 for its next instruction. Note that both X_i and Y_i are now on the drum. X_i is in location 1851, while Y_i is in 1852. Drum locations 1853–1858 also contain data, but they will not be used in this problem. Drum locations 1859, 1860 have zeros stored in them through the board wiring used on the 533-input-output unit. (*Caution*: Unless an experienced person has wired the board, these zeros may not have a sign with the word. Do not try to use them unless you are certain they also have a sign. The program will stop if a number without a sign is used.)

The next instruction is to **R**eset (clear) the entire accumulator and **A**dd X_i (now in location 1851) into the **L**ower half of the accumulator (*Note*: The upper half of the accumulator might have been used. The choice was arbitrary.)

DRUM LOCATION OF INSTRUCTION	INSTRUCTION			COMMENTS
	OPERATION	DATA ADDRESS	NEXT INSTRUCTION ADDRESS	
0500	RCD 70	1851	0510	
0510	RAL 65	1851	0520	

After performing these two operations, the machine now has X_i in its lower accumulator (and distributor) and, according to the instruction address, is seeking its next instruction in drum location 0520.

The program continues:

DRUM LOCATION OF INSTRUCTION	INSTRUCTION			COMMENTS
	OPERATION	DATA ADDRESS	NEXT INSTRUCTION ADDRESS	
0500	RCD 70	1851	0510	
0510	RAL 65	1851	0520	
0520	ALO 15	1852	0530	

The last instruction (in 0520) tells the machine to **A**dd (with**o**ut reset) the contents of drum location 1852 (namely, Y_i) to whatever is in the **LO**wer accumulator (it contains X_i), thus forming $Z_i = X_i + Y_i$ there, and then to go to drum location 0530 for its next instruction.

We now wish to punch out Z_i (the contents of the lower accumulator). To do so, it is necessary to store Z_i on the drum, since it is impossible to punch directly from the accumulator. Furthermore, there are only certain drum locations from which data may be punched. (See shaded portion of Figure 1.3) Quite arbitrarily we select 1530 (one such location) for this purpose. The program now reads:

DRUM LOCATION OF INSTRUCTION	INSTRUCTION		
	OPERATION	DATA ADDRESS	NEXT INSTRUCTION ADDRESS
0500	RCD 70	1851	0510
0510	RAL 65	1851	0520
0520	ALO 15	1852	0530
0530	STL 20	1530	0540

which tells the machine to store the contents of the lower half of the accumulator in drum location 1530 and to go to drum location 0540 for its next instruction. (*Note*: The number Z_i is now in *three* locations in the 650: in drum location 1530; in the lower accumulator; and, since it just passed from accumulator to drum, it also appears in the distributor.) We now tell the 650 to punch out the contents of drum location 1530 and to go back to location 0500 for its next instruction, which is (RCD) 70 1851 0510—i.e., read another card and store it in drum locations 1851 to 1858, then go to location 0510 for the next instruction.

DRUM LOCATION OF INSTRUCTION	INSTRUCTION		
	OPERATION	DATA ADDRESS	NEXT INSTRUCTION ADDRESS
0500	RCD 70	1851	0510
0510	RAL 65	1851	0520
0520	ALO 15	1852	0530
0530	STL 20	1530	0540
0540	PCH 71	1530	0500

If this 5-step program is loaded onto the drum by some means (actually this, too, is done through punched cards) and the set of 600 X_iY_i data cards placed in the read hopper of the 533-read-punch unit, the machine will read the X_i and Y_i from the first card when the program is started, then reset (clear) the lower accumulator and add in X_i; add Y_i to form $Z_i = X_i + Y_i$; store Z_i in location 1530; punch Z_i on a blank card in the output hopper of the 533 read-punch

unit; read the X_i and Y_i from next card, etc., etc. When all 600 cards have been processed, the 650 will receive the final instruction 70 1851 0510, but there will be no card to read, so the machine will stop. The total time to read, compute, and punch these 600 cards will be about 6 minutes. However, this is not to be interpreted that it takes the machine 6 minutes to compute and store these 600 sums. Most of the time the computer is merely sitting idle, waiting for the punch unit which has a maximum output capacity of 100 cards per minute. Actually, the 650 could have computed and punched not only $X_i + Y_i$, but also several other functions, such as $V_i = X_i^2 - 4X_iY_i - X_i$ and $W_i = X_i^3Y_i^5 - 3X_i + Y_i - 17$, in addition to $Z_i = X_i + Y_i$, in the same 6-minute period. It makes no difference, in time, whether only one word, Z_i, or all eight words of the output card are punched! It would, of course, require a longer program to compute all three, Z_i, V_i, and W_i, but the 650 would be able to compute and punch all of them in the same 6 minutes it would take to compute and punch the Z_i's alone. This may give you a clue as to why you must learn a little about how to program the 650, even if you only plan to use it by having someone else write your programs. Often you can secure much more output information for the same "price" *if you plan for it in advance*.

If the above program is used, the results will be a series of 600 cards, each of which will contain a Z_i in the fourth word (Col. 31-40) of the card, and zeros elsewhere. The output card contains the contents of the eight drum locations 1527-1534 in words 1 to 8 of the card, respectively, if the "standard" 8- to 10-digit word 533 control board is used. The kth output card will contain the sum of the two numbers on the kth input card.

If someone should happen to drop the answer pack of cards, there would be no way to get them back into order. This is embarrassing and it is also costly. To avoid this difficulty, you, the student, are asked (Problem 3, Set 1-11) to reprogram this problem in such a manner that not only Z_i but also X_i and Y_i are punched on each output card. Store the X_i in drum location 1527 and the Y_i in location 1528. Your punch instruction can still be 71 1530 0500, although it will be located in a different drum location. Using one of the "standard control boards," which we assume our 533 read-punch unit contains, any data address between 1500 and 1549 used with OP code 71 will cause all eight drum locations 1527 to 1534 to be punched out in words 1 to 8, respectively, of the output card.

1-11. OPERATION CODES.

The operation codes needed in the following problem set are described here.

70 RCD (**Read CarD**). You may use RD1 in place of RCD: This operation code causes the machine to read the 80 columns of an IBM card into the 10 read words (xx01 to xx10, or xx51 to xx60; see Fig. 1.3) in the band selected by the Data Address. The xx means any valid number or address—for example, 0001 to 0010 or 1951 to 1960. With the "standard 533 board," we are assuming that card-word 1 (Cols. 1-10) becomes the first word in the chosen read section, card-word 2 (Cols. 11-20) becomes the second word in the chosen read section . . . card-word 8 (Cols. 71-80) becomes the eighth word (i.e., xx08 or xx58) in the chosen read section and 0000000000 is automatically placed in words 9 and 10. Anything previously stored in these read locations is lost when the contents of a new card are read in. It takes 300 milliseconds (.3 second) to read a card into the machine, but of these 300 milliseconds, 270 milliseconds are available for computing as a result of an ingenious read buffer storage discussed in Chapter 2.

71 PCH (**PunCH**) You may use WR1 (**WR**ite output one) in place of PCH: This causes 80 of the 100 digits in 10 words of the punching section (xx27 to xx36, or xx77 to xx86; see Fig.

1.3) to be punched into an IBM card. Board wiring selects the digits and order in which they are punched. Our ''standard board'' takes the first eight words of the punching section of the drum (xx27 to xx34, or xx77 to xx84) into the eight words of the card, ignoring the last two words of the punching section.

24 STD (**ST**ore **D**istributor): The word (10 digits and sign) in the distributor is stored into the drum location specified by the Data Address. The word also remains in the distributor.

15 ALO (**A**dd to **LO**wer): The 10-digit-and-sign word in the location specified by the Data Address is added to the lower 10 digits of the 20-digit accumulator. The word is found in the distributor, and also in its former location, after the operation is complete.

16 SLO (**S**ubtract from **LO**wer): The word in the location specified by the Data Address is subtracted (algebraic subtraction) from the lower 10 digits of the 20-digit accumulator. The word is, of course, in the distributor and still in its former (drum) location after the operation is complete.

65 RAL (**R**eset and **A**dd into **L**ower): Reset (clear) the entire 20-digit accumulator to zero, and add the word specified in the Data Address to the lower 10-digits of the accumulator. The word is on the drum, in the distributor, and in the lower half of the accumulator after the 65 operation is completed.

20 STL (**ST**ore **L**ower onto drum): The lower half of the accumulator (10 digits and sign) is stored into the drum location specified by the Data Address, replacing whatever was already in the drum location. The contents of the lower half of the accumulator are undisturbed, and the stored word is in the distributor after the operation is complete.

69 LDD (**L**oa**D D**istributor): The word in the location specified in the Data Address is brought into the distributor. It also remains, unchanged, on the drum.

These eight operation codes will enable you to program a number of problems. In doing so, you will not only gain familiarity with 650 technique, but certain properties of the operations themselves will be brought out in the problems. The problems in this manual are designed to help you learn how the computer works; and *if you are to achieve maximum learning in the shortest possible time, each problem should be attempted and the results discussed with fellow students before the next class session.*

Problem Set 1-11

1. The numbers given with each problem show the contents of the accumulator, distributor, and of a specific drum location, before the instruction is given. Write the contents of each after the command has been executed.

	OP Code	Data Address	Accumulator Upper (8003)	Accumulator Lower (8002)	Distributor (8001)	Drum Storage Loc	Drum Storage Contents
(a)	LDD 69	1978	+0000098765	0000605678	−0000000099	1978	+1234500000
(b)	STD 24	0030	+0456558903	0000000000	−0000089897	0030	+0987898765
(c)	RAL 65	1944	−9999999999	4444444444	+9999999955	1944	+8888888888
(d)	ALO 15	1776	−1111111111	8888888888	+098765432	1776	+5555555555
(e)	SLO 16	1492	+1492177622	0000000000	−9999988875	1492	+3333333333
(f)	SLO 16	1588	+0000010000	0000088888	+0000000009	1588	+9000000009

	OP Code	Data Address	Accumulator Upper (8003)	Accumulator Lower (8002)	Distributor (8001)	Drum Storage Loc	Drum Storage Contents
(g)	ALO 15	0024	+0000000080	0000000777	+0000000777	0024	−0000033333
(h)	RAL 65	0945	+9999999999	9999977786	+0908070706	0945	−0001213140
(i)	STL 20	1973	−0008800001	5555555555	−3456723456	1973	+2345645376
(j)	STL 20	1000	+0000000000	0000000043	−0987223434	1000	+3333343334

2. (a) Take a 650 Planning Chart (Form 22-6151-2) and write the 5-step program given in Example 1. As you write each step, also note the contents of the upper accumulator, lower accumulator, and distributor for the first card which contains X_1 = 0005555555 +, Y_1 = 0000003333 +. Assume that after the first operation, 70 1851 0510, the upper half of the accumulator contains the digits of your telephone number, the lower half contains the digits of your girl's telephone number with a (+) sign, and that the distributor contains 1492177657 −, all of which are left over from a previous problem. The execution of a Read instruction does *not* alter the contents of the distributor or accumulator.

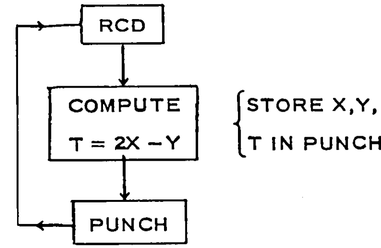
(b) Show the contents of the upper accumulator, lower accumulator, and distributor after each step of the program on the second time through, assuming that the second card read contains X_2 = 0005769121+, Y_2 = 0000009342 −.

3. Reprogram the problem of Example 1, Section 1-10, so that X_i and Y_i as well as Z_i will be punched in the output card, as suggested in Section 1-10.

4. Show the contents of the upper accumulator, lower accumulator, and distributor after each step of the program you wrote in Problem 3. Do not use specific numbers, but fill in known as zeros as zeros, and write 000⟵X_i⟶, 000000⟵Y⟶, 00⟵$X_i + Y_i$⟶, etc., to show the general nature of contents as well as the range of digits which it will involve. For example:

INSTRUCTION OP	DA	IA	UPPER ACCUMULATOR	LOWER ACCUMULATOR	DISTRIBUTOR
RAL 65	1851	0510	0000000000	000⟵X_i⟶	000⟵X_i⟶

5. Write a program which will use the same input X_i, Y_i cards as in Example 1, but which will compute $T_i = 2X_i - Y_i$ and which will punch X_i in word 1, Y_i in word 2, and T_i in word 6 of the output card.



6. Fill in the Accumulator and Distributor columns of your planning chart for the program of Problem 5, using the type of notation described in Problem 4.

7. If the X_i's were 2-place decimals, 000xxxxx.xx, and Y_i's were also 2-place decimals, 000000yy.yy, what, if any, difference would it make in your program for: (a) Problem 3 (b) Problem 5?

8. If the X_i's were 2-place decimals, as in Problem 7, but the Y_i's were whole numbers (integers), what directions would you give your key-punch operator so that she could punch the Y_i's in such a form that you could use the program of Problem 3 to obtain the correct $Z_i = X_i + Y_i$?

9. Alter the situation in Problem 3 by requiring that additional steps be written into the program so that drum location 1534 will contain the sum of the Z_i's from the first card to the present card, in each case. What will appear on the output cards? (*Hint*: Remember 1534 is part of the punchout band.)

10. In addition to placing $\sum_{i=1}^{k} Z_i$ in drum location 1534 on the kth card, also alter your program to compute $\sum_{i=1}^{k} X_i$ in location 1532 and $\sum_{i=1}^{k} Y_i$ in location 1533. What will the 7th output card contain? What will the 600 th output card contain? What will each accumulator and the distributor contain when the machine stops after the 600th card? What will each contain after the power is shut off?

11. Is it possible that any of the sums $\sum_{i=1}^{600} Z_i$, $\sum_{i=1}^{600} X_i$, $\sum_{i=1}^{600} Y_i$ might contain more than 10 digits? Why must this possibility be considered?

12. In Section 1–5 we decided that, for the present, the sign of each instruction should be +. However, no such restriction was placed on the data. Suppose some of the 600 cards in Problems 3 and 5 contain negative values for either X_i or Y_i, or both. What, if any, changes need be made in the programs?

13. Reprogram Problem 5 so that $\sum_{i=1}^{k} T_i$ is also punched on the kth card. Discuss the possibility that $\sum_{i=1}^{k} T_i$ might be larger than 10 digits, for some k in the range $1 \leq k \leq 600$.

14. Devise a program which will enable you to number 600 cards consecutively within the numbers placed in the 8th word of the card. This could be combined with another program, if desired.

15. Devise a program which will reproduce the first seven card words (Col. 1–70) of each card in Data Deck 1 on output cards, and will place a card number k in the 8th word of the kth card. (*Hint*: Use LDD-69 and STD-24 to move the words and use the program of Problem 14 to number the cards.)

1-12. LOGICAL TEST—NZU.

In working Problem 3 of the last set, when did you place the X_i and Y_i into storage locations 1527 and 1528? It would be possible to do this after Z_i was computed, but both computing time and storage locations can be saved by storing a needed word when it *is already in the distributor*. Although there is no time saving on this particular program (why not?), it is a good habit to program economically, even when this is not essential. One possible program would be:

DRUM LOCATION OF INSTRUCTION	← INSTRUCTION: OPERATION	DATA ADDRESS	NEXT INSTRUCTION ADDRESS →	COMMENTS
0500	RCD 70	1851	0510	READS 8 WORDS INTO 1851–58.
0510	RAL 65	1851	0520	X_i IN LOWER.
0520	STD 24	1527	0530	STORE X FOR PUNCH.
0530	ALO 15	1852	0540	$Z_i = X_i + Y_i$ IN LOWER.
0540	STD 24	1528	0501*	STORES Y_i FOR PUNCH.
0501	STL 20	1530	0511*	STORES Z_i FOR PUNCH.
0511	PCH 71	1530	0500	PUNCH AND REPEAT.

*By using 0501 rather than 0550 as the address of our next instruction, the program is kept in the same band on the drum.

Under the conditions of the given problem we were assured that the sum would not exceed a 10-digit number. However, once a program is written, it may be used by others who fail to observe this restriction; perhaps we, ourselves, know that we shall wish later to use the same program to add a series of 10-digit numbers which might have an 11-digit sum. If an 11-digit sum is developed in the lower accumulator, the 11th digit will be found in the low-order position of the upper half of the accumulator. (The reader should recall that there is, really, only one accumulator, and that it is a 20-digit accumulator. It is merely convenient to speak of its upper half and its lower half as separate entities.) An output card from the above program looks like this:

	000 ← X_i →	000000 ← Y_i →	← ZEROS →	LAST 10 DIGITS OF $X_i + Y_i$	← ZEROS →
COL. NO.	1 10	11 20	21 30	31 40	41 80
CARD WORD	WORD 1	WORD 2	WORD 3	WORD 4	WORD 5 WORD 6 WORD 7 WORD 8
CORRESPONDING DRUM LOCATION	1527	1528		1530	

The zeros in word 3 and in words 5 to 8 are the result of having zeroed the drum before loading the program. We shall alter the program so that it will test the contents of the upper accumulator, and if it is not all zeros—i.e., if an overflow into the upper accumulator has occurred—will punch the contents of the upper accumulator into word 3 of the output card and also punch a special warning by punching a 9 into Column 80 if, and only if, an 11-digit sum was developed.

Operation code 44 NZU (Branch on Nonzero in Upper half of accumulator) is an entirely new type of operation, namely, a logical test. This code performs no arithmetical operation. Instead, it tests the upper accumulator to see whether or not it is zero. If the upper accumulator is *nonzero*, the machine goes to the drum location given in the *data address* for its next instruction—i.e., it *branches* on nonzero in the upper accumulator. If the upper accumulator is *zero*, the machine goes to the drum location of the *instruction address* for its next instruction as usual. Thus,

NZU 44	0621	0521
	IF NONZERO	IF ZERO

will send the machine to drum location 0621 for its next instruction if the upper is nonzero, and to 0521 if the upper is zero. We may use this in our program as follows:

DRUM LOCATION OF INSTRUCTION	←——— INSTRUCTION ———		→
	OPERATION	DATA ADDRESS	NEXT INSTRUCTION ADDRESS
0500	RCD 70	1851	0510
0510	RAL 65	1851	0520
0520	STD 24	1527	0530
0530	ALO 15	1852	0540
0540	STD 24	1528	0501
0501	STL 20	1530	0511
0511	NZU 44	0621 (IF NONZERO UPPER)	0521 (NEW INSTRUCTION) (IF ZERO UPPER)
0521	PCH 71	1530	0500
0621	STU 21	1529	0631 (NEW INSTRUCTION)

Let us now suppose that drum location 1998 contains a special code word, say 0000000009, which we wish to have punched into word 8 of each card containing an 11-digit sum. This may be accomplished as follows:

0631	LDD 69	1998	0641
0641	STD 24	1534	0521

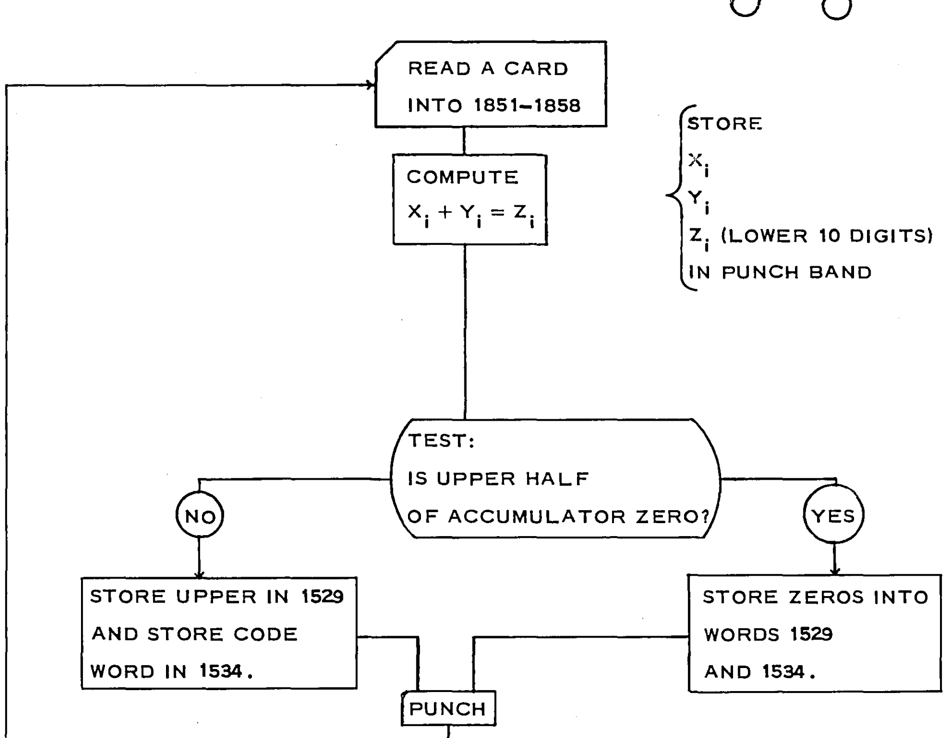
If the sum in the accumulator contains 10 or fewer significant digits, the program will proceed as before. If it contains 11 digits, we shall go from drum location 0511 to 0621. The instruction in 0621 places the digits from the upper accumulator into punch storage location 1529, which contained all zeros in the program. The instructions in 0631 and 0641 place the special code word which identifies an 11-digit-sum card in punch storage location 1534, before continuing with the old program in location 0521, as before. The instruction in 0521 (e.g., 71 1530 0500) causes all eight words in 1527–1534 to be punched. Notice that many of the instructions need not be written in any particular sequence. The important thing is that the drum locations and data addresses be correct.

The reader should fill in this program on a 650 planning chart, filling in the accumulator and distributor entries both when a 10-digit and when an 11-digit sum is developed. This is a vital part of this course, the purpose of which is to teach you to think in a pattern compatible with the machine's operation—a quality which is essential in the successful use of computers. *This suggested program contains a serious blunder.* You can find it, if you will fill in a planning chart. If an 11-digit sum is developed, *all subsequent outputs* will show 11-digit sums and will contain a 9 punch in Col. 80. Why? See if you can discover the source of this difficulty before continuing. Consider what the difficulty is—namely, once an 11-digit sum is developed, an 11-digit sum and a 9 in Col. 80 appear even when only a 10-digit sum is developed subsequently—and discover why the 650 behaves so. Remember that the 650 does exactly what you tell it to do—no more, and no less. Try your hand at "debugging" this program before you continue!

The source of the difficulty is that, when the overflow digit is placed into drum word 1529 and an identifying 0000000009 in drum word 1534, they remain there until they are replaced by something else; and since neither location is disturbed when a 10-digit sum is developed, the contents remain and are punched even though unwanted. If we want to avoid this difficulty, we should store zeros (possibly from drum location 1859) into each of these locations (before we punch) when the 44 test finds the upper all zeros. This is called "housekeeping."

EXAMPLE 1:

A flow chart of our proposed program looks like this (It is customary although not essential in a flow chart, to indicate a branch by a long oval and two circles:



The astute reader will be able to suggest other possible programs which will accomplish the desired ends.

LOCATION OF INSTRUCTION	OPERATION	INSTRUCTION DATA ADDRESS	NEXT INSTRUCTION ADDRESS	COMMENT
0500	RCD 70	1851	0510	READS CARD INTO 1851–58
0510	RAL 65	1851	0520	X_i INTO LOWER
0520	STD 24	1527	0530	STORE X_i IN PUNCH LOCATION
0530	ALO 15	1852	0540	$X_i + Y_i = Z_i$ IN LOWER
0540	STD 24	1528	0501	STORE Y_i IN PUNCH
0501	STL 20	1530	0511	STORE Z_i IN PUNCH
0511	NZU 44	0621 (IF NONZERO)	0521 (IF ZERO)	BRANCH
0621	STU 21	1529	0631	IF NONZERO UPPER BRANCH
0631	LDD 69	1998	0641	CODE WORD FROM 1998
0641	STD 24	1534	0502	CODE WORD INTO PUNCH
0521	LDD 69	1859	0531	IF ZERO UPPER BRANCH
0531	STD 24	1529	0541	STORE ZEROS IN 1529
0541	STD 24	1534	0502	STORE ZEROS IN 1934
0502	PCH 71	1527	0500	PUNCH AND REPEAT

Another way to guard against developing an 11-digit sum is available if you perform the operations in the *upper* half of the accumulator. When an 11-digit number is developed in the upper half of the accumulator, an *overflow* occurs. When it does, the overflow light on the console comes on, indicating that an overflow has taken place. If the overflow switch (also on the console) is set to STOP, the program will stop. There is also a "branch on overflow" operation, 47 BOV, which may be used to reroute the program if an overflow occurs, provided the overflow switch is set in the SENSE position. As usual on branch operations, if the specified condition occurs (here, an overflow), the next instruction executed will be in the location given by the Data Address; otherwise the instruction address is used as usual.

1-13. MULTIPLICATION.

The operation multiply (19 MPY) is somewhat different in nature from those already discussed. The accumulator is *reset* and one of the 10-digit factors is placed in the *upper* accumulator before the instruction MPY

19	DATA ADDRESS	NEXT INSTRUCTION ADDRESS

is given. The location of the other 10-digit factor is specified in the Data Address of the instruction

19	DATA ADDRESS OF 2ND FACTOR	NEXT INSTRUCTION ADDRESS

. The 20-digit product stretches across *both the upper and lower halves of the accumulator*. If it is known that the product will not contain more than 10 significant digits—say, the product of the two 4-digit factors, (000000xxxx)·(000000yyyy)—then the upper accumulator will contain all zeros, and only the lower accumulator is pertinent.

We have already discussed 2000 of the 2004 possible addressable locations of the IBM 650. (What are they?) The other four are the console switches (8000), the distributor (8001), the lower accumulator (8002), and the upper accumulator (8003), each of which is a valid and convenient data address or instruction address for most operations. The 800x series addresses are the most quickly accessible on the 650, and should be used where possible. The 650 will *not* accept an 800x series address as the data address of a store operation (20,21,22, 23,24); in fact, the machine will stop if an 800x series address is used with a store instruction. Storage of a word in an 800x address may be achieved by using OP codes 60,65,69.

EXAMPLE 1:

In a correlation analysis, 600 pairs of 5-digit integers (X_i, Y_i) are available on a set of 600 IBM cards, with the X's in word 3 and the Y's in word 4. Either or both values may be negative.

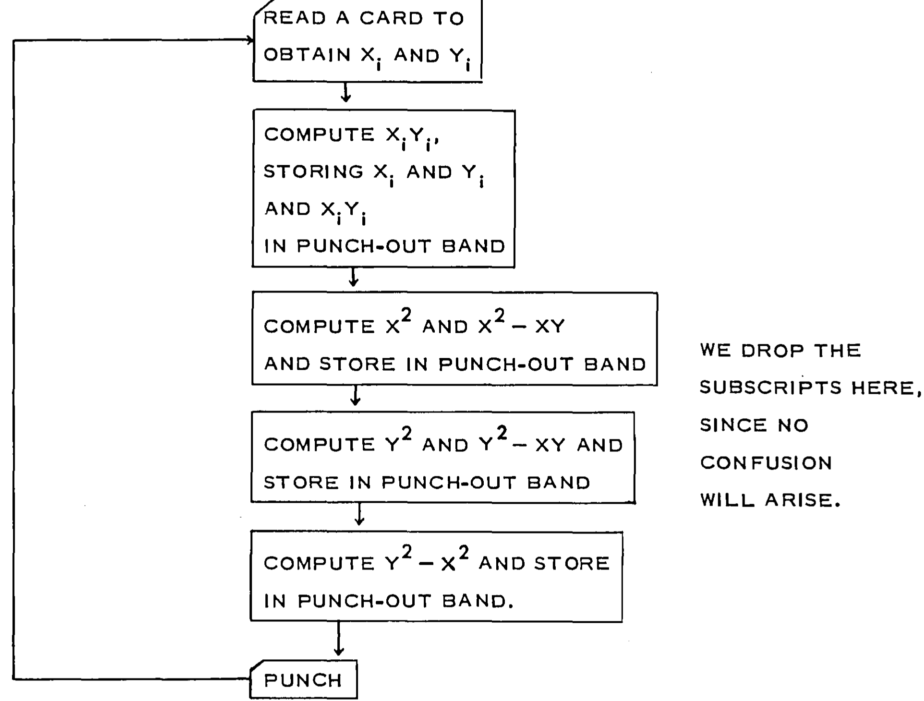
$$-69999 \leqq X_i = 00000xxxxx \leqq 69999$$

$$-69999 \leqq Y_i = 00000yyyyy \leqq 69999.$$

We wish to compute the necessary results, and punch cards containing the following:

X_i	Y_i	X_i^2	X_iY_i	$X_i^2 - X_iY_i$	Y_i^2	$Y^2 - X_iY_i$	$Y^2 - X^2$
WORD 1	WORD 2	WORD 3	WORD 4	WORD 5	WORD 6	WORD 7	WORD 8

A flow chart follows:



The top of your 650 planning chart should be filled in to indicate that word 3 of input contains a 5-digit X_i to be stored into drum location 0303, while word 4 contains a 5-digit Y_i to be stored into location 0304. The output words will be:

WORD	1	2	3	4	5	6	7	8	9	10
MEMORY ADDRESS	1077	1078	1079	1080	1081	1082	1083	1084		
OUTPUT	X	Y	X^2	XY	$X^2 - XY$	Y^2	$Y^2 - XY$	$Y^2 - X^2$		

A possible program follows: The command RAU 60 0303 1010 resets the entire 20-digit accumulator and adds the contents of drum location 0303 to the upper half of the accumulator before going to drum location 1010 for its next instruction.

LOCATION OF INSTRUCTION	OPERATION	DATA ADDRESS	NEXT INSTRUCTION ADDRESS	COMMENTS
	INSTRUCTION			
1100	RCD 70	0301	1105	
1105	RAU 60	0303	1010	
1010	STD 24	1077	1015	STORE X
1015	MPY 19	0304	1020	COMPUTE XY
1020	STD 24	1078	1025	STORE Y
1025	STL 20	1080	1030	STORE XY
1030	RAU 60	0303	1035	
1035	MPY 19	8001	1040	COMPUTE X^2

(Continued on next page)

LOCATION OF INSTRUCTION	OPERATION	DATA ADDRESS	NEXT INSTRUCTION ADDRESS	COMMENTS
1040	STL 20	1079	1045	STORE X^2
1045	SLO 16	1080	1001	COMPUTE $X^2 - XY$
1001	STL 20	1081	1006	STORE $X^2 - XY$
1006	RAU 60	0304	1011	
1011	MPY 19	8001	1016	COMPUTE Y^2
1016	STL 20	1082	1021	STORE Y^2
1021	SLO 16	1080	1026	COMPUTE $Y^2 - XY$
1026	STL 20	1083	1031	STORE $Y^2 - XY$
1031	SLO 16	1081	1036	COMPUTE $Y^2 - X^2$
1036	STL 20	1084	1041	
1041	PCH 71	1077	1100	

The reader is expected to fill in the accumulator and distributor columns at each step of the program, as he copies it on to a 650 planning chart. Attention is called to the fact that, if an 11-digit product or a sum were developed, the program would ignore the most significant digit and no one would be the wiser. Unless more is known about the nature of the data, it is quite possible that any of $X^2 - XY$, $Y^2 - XY$, or $Y^2 - X^2$ might be 11-digit numbers. (Why?) In the next problem set the reader is asked to reprogram the problem to take care of this possibility, either by including 44 NZU operations which will stop the 650 by sending it to an invalid address, if an 11-digit difference is developed, or by using the upper half of the accumulator and overflow stop. (*Note:* The difference between two 10-digit numbers *can* be an 11-digit number, if the terms are opposite in sign.)

1-14. NEW OPERATION CODES.

10 AUP (Add to **UP**per): The "10-digit and sign" word in the location specified by the Data Address is added to the upper half of the 20-digit accumulator. If an overflow occurs, the overflow circuit will be activated. If the overflow switch is set to STOP, the program will stop when an overflow occurs.

11 SUP (**S**ubtract from **UP**per): Similar to 10, but subtracts.

60 RAU (**R**eset accumulator and **A**dd to **U**Pper half): Resets the entire 20-digit accumulator to plus zero and adds the contents of the Data Address to the upper 10 digits of the accumulator.

61 RSU (**R**eset and **S**ubtract from **U**pper): Resets the entire 20-digit accumulator and subtracts the contents of the location given in the Data Address from upper half.

66 RSL (**R**eset and **S**ubtract from **L**ower): Similar to 61, but subtracts from the lower half.

21 STU (**ST**ore **U**pper): Causes the upper half of the accumulator, with sign of the accumulator, to be stored in the drum location specified by the Data Address. The contents of the accumulator are unaffected by the operation. The word also appears in the distributor, of course.

44 NZU (Branch on **N**on**Z**ero in **U**pper): If the upper half of the accumulator contains all zeros, the next instruction to be performed will be found in the Instruction Address. If the upper half is nonzero, the address of the next instruction performed will be found in the Data Address. This operation does not affect the arithmetical units nor the storage units of the 650, but provides a logical test by means of which different branches of a program may be selected.

45 NZE (Branch on **N**on**Z**ero **E**ntire accumulator): Similar to 44, but the entire 20-digit accumulator is examined.

46 BMI (**B**ranch on **MI**nus): If the accumulator has a negative sign, the next instruction performed will be that in the location specified by the Data Address. Otherwise the Instruction Address will be used, as usual.

47 BOV (**B**ranch on **OV**erflow): When the overflow switch on the console is set to SENSE, the program will branch to the location specified in the Data Address if an overflow occurs, but use the Instruction Address as usual if no overflow is present. If the console switch is set to STOP, the program will stop on overflow. (*Note:* If the console switch is in the sense position and an overflow occurs, but the BOV is not used, the overflow light will come on, but the 650 will continue to run ignoring the fact that an overflow occurred. Always leave the overflow switch in the STOP position unless you are using the BOV **OP**eration code.)

00 NOP (**N**o **OP**eration): The 650 merely goes to the Instruction Address for its next instruction. This code is useful for switching the path of the program if an instruction must be deleted or added. Some valid Data Address (usually 0000) must be included in the Data Address location or the machine will stop.

01 HLT (**H**a**LT** or Stop): If the console *programmed stop* switch is set to STOP, the 650 will stop at this operation. If the switch is set to RUN, the 01 code is treated as a NO OPertion (00) code. This is useful in debugging programs and will be discussed in greater detail later.

1-15. FLOW CHARTS.

The flow chart or block diagram is the heart of all true programming—be it for the 650 or for another computer. No program of any complexity should even be discussed, let alone programmed, without a careful flow chart. A flow chart should show the exact sequence of operations including each branch and its alternatives. It should also indicate how and when loops are tested. A flow chart does *not* show individual steps used in computing. The object of a flow chart is to present the overall method of attack, without details. There is little or no mention of specific 650 operation in a flow chart. A competent coder can take a flow chart and translate it into a program for an IBM 650, or for a DATATRON, or for one of the IBM 700 series, or almost any other computer. Flow charting is often used to explain processes which do not involve any computer.

The bit of whimsey on the following page which has been making the rounds of computer laboratories for several years may illustrate the point. It purports to be a program for getting to 8 o'clock class.

It would be hard to overemphasize the importance of flow charts. Indeed, the time may come when you will make only the flow chart, leaving the details of programming to someone (or some*thing*) else. "For transit" is an example of a "something."

Problem Set 1-15

1. Copy the program given at the end of Section 1-12 on a planning chart, filling in the accumulator and distributor columns.

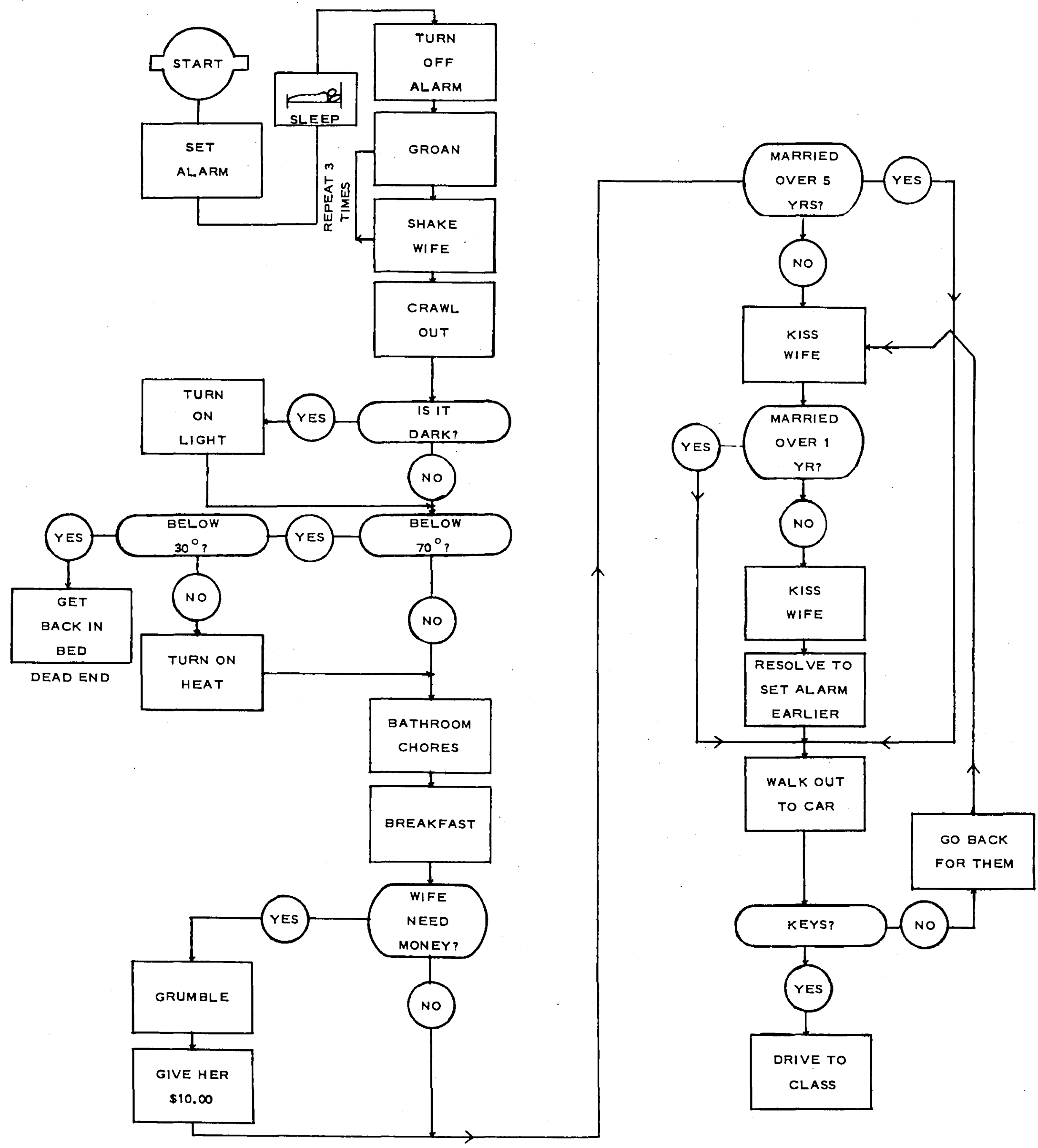
2. Reprogram Example 1, Section 1-12, using the upper accumulator so that, if an overflow occurs, the program will stop.

3. Explain what the 650 will do upon receipt of each of the following instructions (beware of booby traps). At the beginning of each of the operations, the following situation prevails:

8003	8002	8001
3333333333	2222222222+	1111111111−

(a) RAU	60	8001	XXXX	(d) LDD	69	8003	XXXX
(b) SLO	16	8002	XXXX	(e) PCH	71	8002	XXXX
(c) STU	21	8001	XXXX	(f) SLO	16	8003	XXXX

Show the contents of each accumulator and the distributor at the end of the operation.



4. Explain the effect of each of the following operations on the entire accumulator which, at the beginning of the operation, contains 20 nonzero digits and a plus sign.

(a) SUP 11 8003 XXXX	(d) RAL 65 8002 XXXX
(b) RAU 60 8003 XXXX	(e) SUP 11 8002 XXXX
(c) SLO 16 8003 XXXX	(f) RAU 60 8002 XXXX

5. The following programs are intended to form a product of two 3-digit factors located in drum locations 0733 and 0734. However, when the cards are punched and the first word of the output card examined, the desired product is not there. Can you "debug" the program?

(a)

LOCATION OF INSTRUCTION	OP		DA	IA
0500	RAU	60	0733	0505
0505	MPY	19	0734	0510
0510	STU	21	0727	0515
0515	PCH	71	0730	XXXX

(b)

LOCATION OF INSTRUCTION	OP		DA	IA
0328	RAU	60	0734	0338
0338	MPY	19	0733	0337
0337	NZU	44	0300	0345
0345	STU	21	0727	0301
0345	PCH	71	0715	0300

6. With the X_i, Y_i from the set of 600 cards having X = 000000xxxx in word one, and Y = 000000yyyy in word two, make up a flow chart and program which will accomplish the following:

Form the necessary computations to compute and punch output cards of the following form for each input card giving X and Y.

X	Y	$2X - Y$	$2X^2 - XY$	$2X^3 - X^2Y + X$	← ZEROS →
WORD 1	WORD 2	WORD 3	WORD 4	WORD 5	WORDS 6, 7, 8

7. Modify your program of Problem 6 before reading the next card, so that it will also punch a digit 9 in Col. 80 if $2X^3 - X^2Y + X$ is positive.

8. Make up a segment of a program which will accomplish the following:

Obtain A-B in the lower accumulator where A was stored in 0576 and B in 1576. If A-B < 0, go to an instruction located in 1426; if A-B = 0, go to an instruction in 1626; and if A-B > 0, go to an instruction in 1926. (*Hint*: Use both 45 NZE and 46 BMI operation codes.)

9. Write a flow-chart and a program which will use the 600 cards of Problem 6, and give the following on the output card:

(a) Punch X in word 1 and Y in word 2.

(b) If $X^2 < XY$, punch the difference $X^2 - XY$ in word 4.

(c) If $X^2 = XY$, punch an 8 in Column 80.

(d) If $X^2 > XY$, punch a 9 in Column 80 and punch the difference $X^2 - XY$ in word 5. Be sure to include the necessary housekeeping to erase unwanted previous results from the punch band on the drum if nothing is stored there later.

10. Devise a program which will enable you to count the number of drum revolutions the 650 makes in a 1-minute period, assuming that you can start and stop the program at will. (Which you can do by pushing buttons on the console.)

CHAPTER 2

THE 650 CONSOLE

2-1. READ AND PUNCH.

Before continuing with our discussion of the IBM 650, it is essential that you learn more of how it works. This will greatly improve your programming ability as well as assisting you immeasurably in discovering flaws in your program.

You already know that the code 70 RCD causes the contents of an input card to be read into the 650, and that it reads the entire card (80 digits) into either xx01 – xx10 or xx51 – xx60 locations. We have assumed that the 80 columns of the card were read into the first 8 words of the read strip on the drum, and that zeros were fed into the other locations. Although this may be done, it is not necessary, since control-panel wiring in the 533 read-punch unit can be used to place any digit desired in any given position in the 100 (10 words) available drum digits and any sign desired in any of the 10 available sign positions of these words. This data may be taken from the input cards, or may consist of constants supplied by the 533 unit. The control panel may even be so wired that the location of a High Punch (+ or 12 punch) in certain columns will change the entire arrangement in which data are fed into the drum. The wiring of these boards is a separate study, and need not concern us at present, except that even nonprogramming users of the 650 must realize that the input is quite versatile! Similar remarks apply to the output. This is especially important in business data processing such as inventory, accounting, payroll applications, etc. Students interested in these fields should study board wiring.

The 650 has an input rate of 200 cards per minute, but this does *not* mean that the machine is unable to compute during the 60/200 = 0.3 seconds = 300 milliseconds, during which a card is read. An ingenious arrangement called a *buffer storage* makes 270 of these 300 milliseconds available for computation! If properly programmed, the 650 could perform about 300 10-digit additions and subtractions in the 257 milliseconds available.

The punch buffer acts in a similar fashion but twice as much time is available (110 drum revolutions).

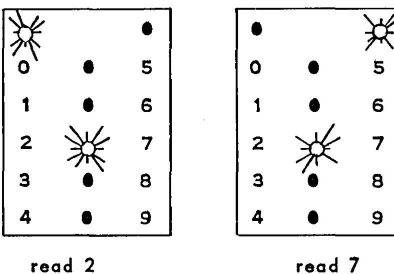
2-2. POWER CONTROL SWITCH.

If a 650 is available, go and look at it. If not, turn to the front cover of this book where the console control panel is shown. A word of warning to beginners. On the lower right-hand

edge is a recessed "Master Power" switch. This is for *extreme emergency use only*! If it is turned to OFF, all power to the 650, including that to the cooling blowers, is cut off and severe damage to the machine may result. Once this switch is turned to OFF, a special locking device is engaged which makes it *impossible* to turn the 650 on again. A customer engineer must be called in order to release the lock and, since damage to the 650 often results, the machine may be out of order for a week or more. *Stay away* from the master power switch unless a real emergency arises. The "power-off" and "power-on" buttons at the top of the console are used to turn the power off and on. If the "power-off" button is pushed, the "power-on" button should not be pushed until the drum has stopped rotating (about 5 minutes); otherwise a belt may be snapped, again causing delay until the customer service engineer can come to replace it.

2-3. DISPLAY LIGHTS.

Numerical data is displayed in a simple biquinary (2-5) code, requiring one light of a pair of lights, and one light of a set of five lights to be lit to represent a number.



The pair of lights at the top tells which column of digits to read, and the set of five lights between the two columns of digits tells which digit in the column to read. When the 650 is computing, it is usually impossible to read these lights; they change so fast that the impression remains that almost all of the lights are continuously lit.

The top bank of lights contains 10 digits and a sign (see cover). By suitable positioning of the display switch, the contents of either accumulator, the distributor, the program register, or any drum location may be displayed there.

The lower bank of numerical display lights consists of a 2-digit operation register and a 4-digit address register. The remaining lights tell what the machine is doing and, if stopped by an internal validity check, where an error was sensed. These lights are the best friend you have when trouble arises (which it does—and almost invariably it is your fault, not the machine's.)

2-4. FLOW OF INSTRUCTIONS.

Let us wander through the machine, as it executes a few steps of the program of Example 1, Section 1-13.

	OP		DA	IA
1100	RCD	70	0301	1105
1105	RAU	60	0303	1010
1010	STD	24	1077	1015

The 650 is told (by means of a card, or the storage entry switches) to seek an instruction in location 1100. Here is what happens:

OP LIGHTS	ADDRESS LIGHTS
BLANK	1100

The 650 now goes to location 1100, finding the instruction 70 0301 1105 which is brought into the program register (this is inside of the 650, but can be displayed on the upper display lights by proper setting of the display switch.) (*Note*: the sign is not shown when reading the program register.)

After the instruction has been stored in the program register, the first 6 digits are then moved into the operation and address registers. At this time the lower lights show the first part of the instruction.

OP	ADDRESS
70	0301

The 650 now sends the 10 words, which are already waiting in the buffer storage, over to words 0301–0310. It then moves the last 4 digits (address of next instruction) of the word stored in the program register into the address register, blanks out the operation register, and displays this in the lower display lights

OP	ADDRESS
BLANK	1105

The 650 goes to location 1105, finding instruction 60 0303 1010 which is then brought into the (internal) program register. While it is doing this, and for the next couple of hundred steps, it is also independently reading another card onto the read buffer storage to be ready for the next 70 code read instruction when it comes. This is the slow part that takes about 27/100 of a second, but the 650 can compute merrily, if it has computing to do, while this is being done. After the instruction was brought into the program register, the first 6 digits were automatically brought into the operation and address registers, so that the lower display lights now show:

OP	ADDRESS
60	0303

and the 650 executes this instruction (reset the entire 20-digit accumulator and add the 10-digit-plus-sign word stored in drum location 0303 to the upper half of the accumulator). The last 4 digits from the program register are displayed next:

OP	ADDRESS
BLANK	1010

which sends the 650 scurrying to drum location 1010 for its next instruction, 24 1077 1015. It places this in the internal program register and the lower lights

OP	ADDRESS
24	1077

Next, this instruction is executed (store the 10-digit-plus-sign word now in the distributor into drum location 1077). The location of the next instruction is then displayed, and the machine goes there to find it, etc.

OP	ADDRESS
BLANK	1015

At the same time the 533 read-punch unit is slowly (very slowly, considering the speed of the operations going on in the arithmetic unit) reading another card into the read buffer storage. The arithmetic unit may do several hundred arithmetic operations on 10-digit numbers in the 27/100 second needed for the 533 read-punch unit to get the 80 digits of information sorted into the 100 digits of drum space (plus signs) in the read-buffer storage.

The progress of a program through the 650 can be followed step by step on the 650 console by using the half-cycle switch.

It is interesting to note the three quite distinct forms which a number has in its course through the 650:

(a) a punched hole in an IBM card, which is "read" by the length of time a current is interrupted while the card separates a brush from a charged roller as it passes between them.

(b) magnetized spots on the drum (permanent until remagnetized in a new pattern).

(c) capacitor storage (transient).

2-5. INTERNAL CHECKING OF THE 650.

One of the strong points of the IBM 650 Magnetic Drum Computer is the excellent system of internal checking which has been engineered into its design. Every digit that leaves the distributor or the accumulator (which means any time a word moves in the machine) is automatically checked to see if it has exactly one binary and exactly one quinary bit.* If not, the machine will stop and the error-sense light will come on. The program register also has a separate validity check built into it. In addition to this, the machine will also stop if any of the following conditions occur:

(a) An invalid address is given on an instruction (i.e., other than 0000–1999, 8000, 8001, 8002, 8003, the latter four not being valid on store, read, or punch instructions).

(b) An invalid operation code is used (say, 29 1726 1344).

(c) An 11-digit number is developed in the upper accumulator (overflow), with the overflow switch set on STOP.

(d) Division by zero, or by an improperly positioned divisor, is attempted (causing what is called overflow).

The reader should consult Figure 2–1 to help assimilate these facts.

Problem Set 2-5

1. (A class problem). Assign one member of the class to take the part of each of the following portions of the 650 at the blackboard.

(a) OP and Address register.

(b) Distributor.

(c) Upper accumulator.

*Internally the 650 is not biquinary, but the distinction is not important here and now.

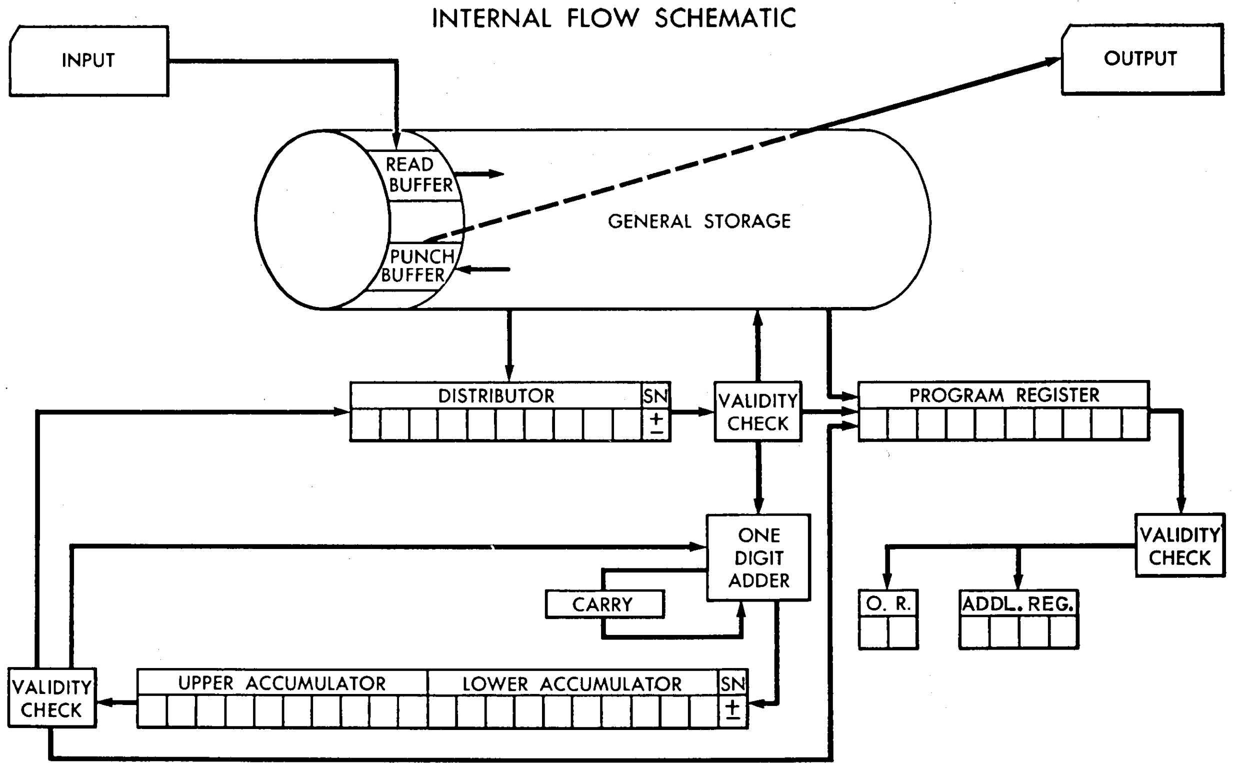


FIGURE 2-1. INTERNAL FLOW SCHEMATIC

(d) Lower accumulator.
(e) Drum (show only location being used at the time).
(f) Read buffer and punch buffer.

Carry out the steps of Example 1, Section 1-13, showing exactly what happens at each step.

2. (A class problem). (a) Use the technique of Problem 1 to determine what will happen on the following program. Assume data input cards having X = 0000XXXXXX in word 1 and Y = 0000000YYY in word 2 are used.

LOCATION OF INSTRUCTION	OP	DA	IA	
0190	RCD 70	1801	0199	
0199	RAU 60	1802	0207	
0207	MPY 19	1801	0208	
0208	STD 24	1831	0209	
0209	STU 21	1828	0210	
0210	STL 20	1827	0207	sic.
0211	PCH 71	1830	0190	

(b) Assume that the input words contained 8 significant digits each. What effect would this have on the machine's behavior? Assume that, if an 11-digit number is developed in the *upper* accumulator, the 650 will stop (overflow), but if developed in the lower, it will merely use the upper to store the extra digits. (c) Answer the same questions assuming that 0210 contains 20 1827 0211, and 0211 contains 71 1830 0207.

3. What happens if the following program is used, assuming that the cards of Problem 2 are used? Describe the output cards, if any. Code RAM 67 is Reset the entire accumulator and Add into the lower half the Magnitude (absolute value) of the 10-digit word given in the location specified by the Data Address.

LOCATION OF INSTRUCTION	OP	DA	IA	
0200	RCD 70	1851	0209	
0209	RAM 67	1852	0217	(LOWER)
0217	STL 20	1878	0235	
0235	RAM 67	1851	0267	(LOWER)
0267	RAU 60	8002	0284	
0284	STD 24	1877	0201	
0201	MPY 19	1878	0218	
0218	STU 21	1879	0220	
0220	STL 20	1880	0225	
0225	PCH 71	1852	0200	

4. What will the following program do, assuming that location 1099 contains the constant 0000000007, that 1098 contains 0000000001, and that location 0200 is the first instruction of a valid program which will multiply the number, X, in location 1027 by the number 0000000002, store the product back into 1027, and then go to 0399 for its next instruction. Describe output cards, if any.

LOCATION OF INSTRUCTION	OP	DA	IA	
0501	RCD 70	1901	0502	
0502	LDD 69	1901	0503	
0503	STD 24	1027	0504	
0504	STD 24	1028	0200	
0200	{PROGRAM DESCRIBED ABOVE, MANY STEPS		0399}	SUBROUTINE
0399	RAL 65	1099	0400	
0400	SLO 16	1098	0401	
0401	NZE 45	0200	0402	
0402	PCH 71	1027	0501	

5. Explain how the program above would need to be modified, both in the portion given and in the subroutine, if the output card were to contain both X and $X(2)^7$, assuming that X is of the form 000000XXXX.

6. Would the program of Problem 4 punch X and $X(24)^7$ if X were a 4-digit number and if the drum location containing the 0000000002 contained 0000000024 instead?

7. (a) Describe the effect of the three following programs. Assume drum location 1999 contains the number 0000000001, and that 1998 contains the number 1080018002

PROGRAM A

LOCATION	OP	DA	IA
0120	RAU 60	1999	0200
0200	AUP 10	1999	0200

PROGRAM B

LOCATION	OP	DA	IA
1319	RAU 60	1999	1320
1320	AUP 10	8001	1320

PROGRAM C

LOCATION	OP	DA	IA
1520	RAL 65	1998	1521
1521	AUP 10	1999	8002

(b) Which program will have a higher number in the *upper accumulator* at the end of 1 minute? Why? Explain what each does.

(*Note*: The remainder of Chapter 2 may be read quickly now, and referred to in more detail as the need arises.)

2-6. LOAD CARDS.

A wide variety of card forms, both for input and output, is available as a result of board wiring. No matter what board wiring is used, a card having a "high punch" (12 punch) in Column 1 will be known as a *load card*. It would be quite possible to use a high punch in any other column to designate a load card, but we shall use Column 1. (*Note:* A high punch is, in appearance, the same as a "12" or plus punch. Since it is used to control the way the card is read in, rather than as the sign of a word, it is given the name of "high punch.") This load (high) punch in Column 1 does two things:

1. It instructs the 533 to ignore all board wiring on the control panel and to read the contents of the card in as 80-80—i.e., just as we have been doing so far, with zeros entered into words 9 and 10.

2. In a read instruction, RCD **70 DA IA,** if the card read is a load card, the next instruction will be obtained from the location specified by the Data Address, rather than by the Instruction Address (i.e., a Branch on load card is built into the machine.)

The primary use of load cards is to load instructions and constants onto the drum. However, load cards may be used at any time, supplying a branching operation. All 10 columns of a word which will be used in the 650 must be punched, and each word must have a sign (+, –) punch over the units positions of every word used. If a word is not used in the program, this restriction does not apply.

Standard "prepunched" cards of the form shown below are often used for loading instructions onto the 650 drum.

Cards are prepunched with 69 1954 1953 in word 1, 24 ---- 8000 in word 3, and high punches (+ signs) placed in Columns 1, 10, 20, 30, 40.

Card word 1 (Cols. 1-10)	69 1954 1953 (Prepunched)
Card word 2 (Cols. 11-20)	Blank now. Will contain problem reference data if desired.
Card word 3 (Cols. 21-30)	24 ---- 8000 The blank DA will contain the drum address into which the instruction will be stored.
Card word 4 (Cols. 31-40)	Will contain instruction to be punched.
Card words 5 to 8 (Cols. 41-80	Blank now. Used to punch descriptive comments if desired.

Set storage entry switches (8000) on the console to 70 1951 9000. Let us examine what happens.

LOCATION OF INSTRUCTION	OP	DA	IA	
8000	RCD 70	1951	9000	
1951	LDD 69	1954	1953	
1953	STD 24	xxxx	8000	(WHERE xxxx REPRESENTS THE DRUM LOCATION IN WHICH THE INSTRUCTION IN 1954 IS TO BE STORED.)
1954	THIS IS THE 10-DIGIT INSTRUCTION WHICH IS TO BE STORED.			

What does this program do?

8000 (console switches) 70 1951 9000 says, "Read a card into drum locations 1951 to 1958 and, since it is a load card, go to 1951 for the next instruction."

1951 is the first word on the card (Cols. 1-10) which says, "Load distributor with the contents of 1954 (the instruction to be placed on the drum) and go to 1953 for the next instruction."

1953 says, "Store distributor into xxxx (the desired drum location) and go back to the console 8000 for your next instruction.

Read another card into drum location 1951–1958 and ...."

Eventually we may come to a nonload card (no high punch in Col. 1). In this case the 650 goes to drum location 9000 for its next instruction and the card is also read in through board wiring. Since 9000 is an invalid address, the program stops on a data-address storage-selection error and does not proceed until sent to an address containing a valid instruction. This is easily done using a transfer card, described later. The transfer card, a load card, contains 00 0000 NNNN in word 1 where NNNN is the location of the first instruction in the program. How does it work?

This is a one-instruction-per-card load routine. Each card carries its own complete loading program. Two hundred instructions, constants, etc., may be fed into the 650 in 1 minute. This method is almost universally used to debug programs, since corrections can be made by merely inserting another card at the end of the deck; there is no need to remove the card which is in error, since the correction card fed in at the end of the deck will store the correct instruction in the drum location, erasing whatever was there before. After all errors are corrected, it is possible to shorten the input time on a long program by using load routines which store more than one instruction per card. There are 4-per-card, 5-per-card, 6-per-card, 7-per-card and even 8-per-card load routines, each of which has certain advantages. Now, we shall use only the 1-per-card load routine.

2-7. BOARD WIRING FOR LOAD CARDS.

A person familiar with general board wiring for IBM machines will find the 533 board familiar in most details. Others may find it quite awesome. We merely observe here that in any installation it will always be possible to have a board wired to meet your own personal needs by someone who understands this phase of the work, and that "standard boards" will be available in permanent form. At present we describe only the wiring needed for load cards. This consists of one wire running from the hub marked 1 in the "first-reading" section to either of the two hubs marked "load," as shown on the diagram below. If you are interested in board wiring, consult the IBM 650 Programmers Manual for further details.

1 — FIRST READING — 10
LOAD
21 25
ENTRY A
41 45
ENTRY B
61 65

2-8. OPERATING AND CHECKING LIGHTS.

The best friends you will have in debugging a program on the 650 will be the operating lights and the checking lights on the console.

The six operating lights tell what the 650 is doing.

1. *The Data Address* and, 2. *Instructions Address Lights*, indicate which half-cycle of the program is ready to be executed (see Section 2-3).

3. *The "Program" Light* indicates a programmed stop (01), manual stop (button on console), or address stop (console).

4. *The Accumulator Light* is on whenever the accumulator is in use.

5. *The Input-Output Light* indicates that the machine is ready to use one of these facilities. If the 650 stops with the input-output lights on, look for the following reasons:

(a) You forgot to press the 533 read or punch feed button, hence no cards are available when the program calls for them.

(b) The read or punch hopper has run out of cards.

(c) Someone pressed the read or the punch stop key.

(d) A feed failure (often the result of a bent or torn card) has occurred. Be careful of this, or a card jam may result.

6. *The Overflow Light* will come on if overflow is detected. An overflow condition can be caused by one of the following:

(a) An excessive accumulation.

(b) Trying to develop a quotient of more than 10 digits.

(c) Trying to exceed the number of shifts called for in a shift and count operation.

The eight checking lights show up the following causes of program stoppage:

1. *The Program Register Light* indicates the detection of a validity error at the output of the program register.

2. *The Storage Selection Light* indicates errors of the following type:

(a) A store operation with a data address of the 8000 series.

(b) An invalid D or I address (other than 0000 through 1999 and 8000 through 8003).

(c) An attempted manual entry from storage entry switches to 8001, 8002, or 8003.

3. *The Distributor Light* will come on if a validity error is detected at the output of the distributor.

4. *The Accumulator Light* comes on if a validity error is detected at the output of the accumulator.

5. *The Clocking Light* lights if an error is detected in the clocking (timing) circuitry. This is a machine error. If it occurs, try to run your program again. If trouble persists, give the 650 a rest and tell your service man about it.

6. *The Error-Sense Light* is operative when the error-sense switch is in the SENSE position and will come on when one of the following errors has been detected:

(a) Validity error

(1) Program register

(2) Accumulator

(3) Distributor

(b) Clocking (timing) error

This light will remain on until reset by the operator. It indicates machine trouble. The same comments as given in 5 apply. Overheating due to inadequate venting is a common trouble maker.

7. and 8. *The Control Unit and Auxiliary Lights* (not on all units) indicate difficulties in these sources.

One of the most common troubles is to "get into a loop or repeated sequence of instructions and not get out of it." This may occur if an error is made in an instruction address, sending the program back to a previous instruction. For example, if the next to the last instruction of Example 1, Section 1–12, had been

DRUM LOCATION	OP	DA	IA	
0530	20	1530	0510	(THE IA SHOULD HAVE BEEN 0540)

the program would loop back to 0510 and repeat itself over and over again without ever reaching the 0540 step of "punch and go to read card."

If, through an error, the 650 is given an IA of a drum location not used in this program, it will "take off" on whatever it finds there. If someone else has a program in that location,

you may find yourself working his problem with your data—an unsatisfactory arrangement. To prevent this, it is usual to load the entire drum with all zeros before starting to load in a program. Several good "drum zero routines" are available. The author prefers to "stop code" the drum rather than zero it. This places 01 NNNN 0000 in drum location NNNN.

If the 650 is sent to a drum location containing 00 0000 0000, it behaves as follows: Since 00 says "no operation, go to Instruction Address," it does. If drum location 0000 also contains 00 0000 0000, the machine will very rapidly do

"no op on 0000 and go to 0000; no op on 0000 and go to 0000; no op on 0000 and go to 0000; etc."

The lights will give the impression that the machine has stopped, since it will be repeating "no op and go to 0000" at the rate of 12,500 times per minute. (*Query:* Since it takes less than .003 second to perform a "no op," why are there not more than 12,500 repetitions per minute?) The reader should decide what happens if 01 NNNN 0000 is in drum location NNNN. HLT 01 is the operation code which halts the program if the program switch is on STOP. (*Note:* Drum zero routines usually clear the drum to zero with a *minus sign.* There is a very good reason for this. In most mathematical operations resulting in a zero, the computer produces a zero with a plus sign. Hence, a plus zero in a given storage location indicates that it is the result of computation, not of the drum-clearing routine.)

2-9. ADDRESS STOP.

In debugging programs it is often useful to check and see how far a program runs before getting into trouble. The *Address Stop* permits this. Set the control switch on the console to Address Stop. Set the desired address at which you wish to stop into the four address selection switches. Press the program start button. Every time the program uses the stop address as either a Data Address or an Instruction Address, the computer will stop. When Program Start is depressed, the computer will continue with the program.

2-10. PROGRAM START.

The address selection switches are also useful in other ways. If you wish to begin computing on a program with a certain instruction (possibly, but not necessarily, the first instruction in the program), the following steps are useful.

1. Set the control switch to *manual.*
2. Set drum location [i] of the first instruction to be executed into the address switches.
3. Depress the Program Reset (or Computer Reset) key.*
4. Depress the Transfer key (the address register will now show the address of the drum location to which you desire to go).
5. Set the control switch to Run or to Address Stop depending upon your wishes.
6. Depress the Program Start key.

Another method of starting the program involves the set of 11 storage entry switches:

1. Set 00 0000 [i] in the storage entry switches where [i] is the drum location of the first instruction to be executed.
2. Depress the Program Reset or the Computer Reset* key.
3. Set the control switches in either Run or Address Stop positions.
4. Depress the Program Start Key.

*The Program Reset will reset the program register only. The Computer Reset will reset both the program register and the accumulator.

2-11. COOKBOOK DIRECTIONS.

For the convenience of the beginner running his first 650 program, we present the following "cookbook" directions for running the 650. This approach may be justified during the first two weeks of 650 use, but, after that, understanding of the 650 and its operations should replace such methods. If you are to learn to use the 650 intelligently, a general understanding of its construction is essential.

650 Console Operations

To Load and Start Program (Using transfer card).

0. Punch in on the time clock, if one is used.
1. Be sure that the correct control panel is in 533.
2. Obtain drum zero cards from your instructor or the library file. Place the load cards behind (i.e., so they are read later by the 533) the zero drum card(s).
3. Place the transfer card behind the load cards. (Transfer card: Cols. 1–10, 00 0000 xxxx, where xxxx is the location of the first instruction overpunches in Columns 1 and 10.)
4. Place the data cards behind the transfer card. (Data cards are cards which will be read in by program.)
5. Place the above deck in the read hopper, 12 edge first, face down.
6. Place blank cards in the punch hopper. (Be sure that the punch feed has been cleaned out—i.e., hold the punch start button down 3 or 4 cycles while the cards are removed from the hopper, if in doubt.)
7. Set the *Storage Entry* switches to 70 1951 9000.
8. Set the *Control* switch on RUN.
9. Set the *Display* switch to ACCUMULATOR, DISTRIBUTOR, OR PROGRAM REGISTER.
10. Set the *Half cycle* switch on RUN.
11. The *Programmed Stop, Overflow,* and *Error* switches can be set according to desire.
12. Push the *Computer Reset.* (This destroys contents of accumulators and distributor.)
13. Push the *Program Start.*
14. Push the *Read Start* button on 533.
15. Push the *Punch Start* button on 533.

To read the contents of a given drum location (say 0546):

1. Push the *Program Stop* button.
2. Set the *Address Selection* switches to 0546.
3. Set the *Control* switch on MANUAL.
4. Set the *Display* switch to READ-OUT STORAGE.
5. Push the *Program Reset* button. (This destroys the contents of distributor.)
6. Push the *Transfer* button.
7. Push the *Program Start* button.
8. Read the contents of 0546 in the DISPLAY lights.

Problem Set 2-11

1. The following simple program may be punched on a single card; the overpunches (plus signs) are placed in Columns 1, 10, 30, 40, 50, 60, 70, 80. A minus overpunch is used in Column 20. Console switches are set to 70 0004 xxxx.

LOCATION OF INSTRUCTION	OP	DA	IA	±	COMMENT
8000	RCD 70	0004	9900	+	CONSOLE SWITCHES
0004	RSU 61	0008	0007	+	CARD WORD 4
0007	LDD 69	0006	0005	+	CARD WORD 7
0005	STD 24	0000	8003	+	CARD WORD 5
8003	STL 20	1999 [xxxx]	0003	−	NOT ON DRUM. IN UPPER ACCUMULATOR. ADDRESS [xxxx] WILL CHANGE IN PROGRAM. IT STARTS AS 1999.
8002	00	0000	0000	−	NOT ON DRUM, BUT IN LOWER ACCUMULATOR
0003	AUP 10	0001	8003	+	CARD WORD 3
0001	00	0001	0000	+	CONSTANT, CARD WORD 1
0002	00	0000	0000	−	CONSTANT, CARD WORD 2
0006	HLT 01	0000	8000	+	CARD WORD 6
0008	STL 20	1999	0003	+	CARD WORD 8

Since the card is a load card, the 650 goes to drum location 0004 for its first instruction after having read the card into words 0001 to 0008. This routine will clear all drum locations to 00 0000 0000− except locations 0000 and 0001; these will contain 01 0000 8000 and 00 0001 0000, respectively. Explain how it works. Make out a planning chart and fill in the accumulator and distributor columns. This is not the best drum-clearing routine available, by any means, but it is simple and satisfactory. If you have a 650 available, time it to see how long it takes. Can you think of a possible advantage of all zeros and a minus sign in almost every location on the drum before beginning to load a program?

2. A load card contains a high punch (+) in Columns 1, 10, 20, 30, 40, 50, 60, 70, 80, and the following information:

Card word 1 60 0002 8003
Card word 2 20 LLLL 0003 (fill in LLLL with lower clearing limit)
Card word 3 10 0004 0005
Card word 4 00 0001 0000
Card word 5 11 0006 0007
Card word 6 20 UUUU 0003 (fill in UUUU with upper clearing limit)
Card word 7 44 0008 8000
Card word 8 10 8001 8003

Set the console switches to 70 0001 9000. The program will then clear the drum to zero between the limits (drum locations) given in the Data Address of words 2 and 6.

(a) How does it work?
(b) Will the drum locations specified in the limits also be cleared to zero?
(c) Will the cleared locations contain zero and a plus sign or zero and a minus sign?
(d) What happens if LLLL = 0001 and UUUU = 1492?

3. Make up a drum-clearing routine similar to that in Problem 2 but having the following characteristics: (a) It uses console switch settings of 70 1951 9000 and (b) clears between limits, but (c) the upper limit must be less than 1900.

4. What will happen if someone uses the program you have written in Problem 3, but places 1999 as the upper limit?

5. The reader is already familiar with the example of Section 1-12. Your instructor will furnish you with a set of load cards supposedly for this program, but which actually contain an error. Different errors may be given to different students. Debug the program.

6. The following program is supposed to read a card into the 650, compute the product of the second and the seventh card words and punch them and their product into an output card. Debug this routine before the cards are punched, punch the cards, and test the revised program on the 650.

LOCATION OF INSTRUCTION	OP	DA	IA
0901	RCD 70	1535	0902
0902	AUP 10	1502	0903
0903	STD 24	1377	0904
0904	MPY 19	1507	0916
0916	STD 24	1378	0927
0927	STL 20	1381	0928
0928	STU 21	1380	0934
0934	PCH 71	1377	0901

In Problems 7–15 certain common troubles encountered in running the 650 are described. See if you can diagnose the possible sources of trouble, and explain its cause and cure.

7. (a) The program deck is placed in the 533 read hopper and the switches on the console are properly set. David pushes the Computer Reset and the Program Start buttons but the cards are not read into the 650. Since the input-output light is on, he reads the OP register which shows 70 1951. What should he do next?

(b) David now pushes the read button on the 533 and two or three cards are fed into the 533, but then action stops again. This time the following lights are lit: Instruction Address, Program, Storage Selection. Can you diagnose David's trouble? (If not, see Step 1 in the cookbook instructions)

8. David fixes the trouble of Problem 7 and begins again. All goes serenely for a while and about 30 or 40 cards read in and load onto the desired positions on the drum when suddenly action stops again. The OP register is blank and the address register shows 9000. The following lights are lit: Instruction, Program, Storage Selection. Since the trouble is indicated in the Program register, David displays it, finding 70 1951 9000. Can you suggest at least three possible sources of David's trouble?

9. David's instructor removes the cards which remain in the 533 read hopper and holds down the start button to clean out the cards inside the 533 read. He then counts back four cards from the last one which comes through and shows it to David, who notices that it does not have a high (load) punch in Column 1. David punches a high punch in Column 1 and returns it to the pack and begins again. Exactly the same thing happened as in Problem 8 (but from a different cause). David repeats his instructor's action and finds the fourth card from the end is the same card he had trouble with before. He's puzzled at first, but then realizes what the trouble is. He returns the card to the pack, glances quickly at a few cards on each side of it, and starts again. This time the entire program loads into the 650 and the transfer card starts the program which runs perfectly. What was David's trouble?

10. Alice punches and verifies a program from an old listing. The program and the cards are both correct. She then places three blank cards on the end of the program (which requires no data cards) and loads it onto the 650 drum in the usual fashion. Everything goes fine down to the last card, but then the action stops. The address register shows 9000 and the instruction, program, and storage selection lights are on. Alice reads the cookbook instructions and finds her trouble. Can you?

11. Alice punches an 8-word card containing 00 0000 $134\overset{+}{6}$ in word 1, with words 2 to 8 blank to send the 650 to drum location 1346 where her program starts. She reloads the program but her trouble of Problem 10 is repeated. What did she fail to do to her load card, and why did it cause trouble?

12. Lois finds an error in her program. An instruction reads 60 0813 1645 but should read 60 0813 0645. She "duplicates" the first 36 columns of the card using the key punch "duplicate" button and punches 0645 in columns 37–40. When she reloads her program, action stops with 69 1954 in the operation lights and the Data, Program, and Distributor lights lit. What did Lois overlook on the card she repunched?

13. Lois corrects her error of Problem 12 by punching one additional punch, and then loads her program. Since she has a valid transfer card between the program cards and the data cards, the 650 transfers into her program, reads a data card, and computes a while. Suddenly the data address and the input-output lights come on and the operation register shows 71 0712. What did Lois forget to do?

14. Betty punches the necessary button to let Lois' program continue. It does so, reading, computing, and punching merrily along for about two or three minutes when suddenly the same lights appear as in Problem 13. What happened this time?

15. Betty was so pleased with the result of her button punching that she punched a few more 533 buttons. The "end of file" light was lit when Lois returned and the action had stopped. Lois quite properly depressed the stop button on the 533, turning off the end of file light, before depressing the read and punch buttons. Her program ran fine and, since she had placed three blank cards at the end of the data cards, all the data cards fed into the 650 and all the results were punched. However, when she listed (407 or 402) her output cards, the answers corresponding to the last one or two input cards were missing. Where are they?

CHAPTER 3

MORE ADVANCED PROGRAMMING

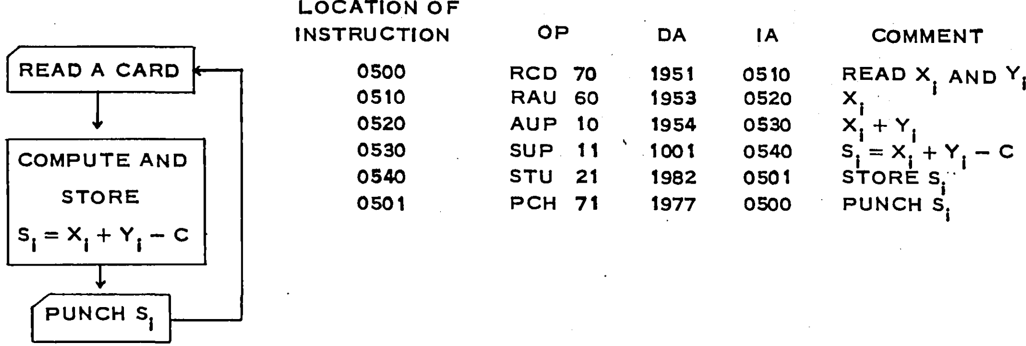
3-1. IMPROVING SPEED.

In Problem 3 Set 1-11, $Z_i = X_i + Y_i$, it took six minutes to compute Z_i and punch X_i, Y_i and Z_i for 600 values of i. The time needed to obtain the 600 Z_i may be cut in two by punching two sets of results ($X_{2i-1} Y_{2i-1} Z_{2i-1} : X_{2i} Y_{2i} Z_{2i}$ for i = 1, 2, . . . , 300) into each output card. No further increase in speed is possible unless the input cards are also altered. It may be both feasible and economical to alter the input cards. If a less expensive machine, like a summary punch, were used to put 4 pairs of X_i, Y_i onto each input card, the program could be speeded up even more. If, for example, a program were written which would read in and compute Z_i for 8 pairs $X_i Y_i$ from two of the new cards, and punch all 8 answers plus some identifying mark (quite possible, since the 8 answers will need only 64 of the 80 available columns), it would be possible to compute all 600 values in 45 seconds—a savings of $7 (at $80 per hour for 650 time).

Actually, the amounts involved in the above problem are small, and probably not worth the effort of reprogramming, but the ideas are essential. The given problem, $Z_i = X_i + Y_i$, is not even a reasonable 650 problem, but would be relegated to a smaller, less expensive machine in most production schedules. However, it is an excellent instructional problem, since it offers much leeway for discussion without the more involved considerations of advanced programming.

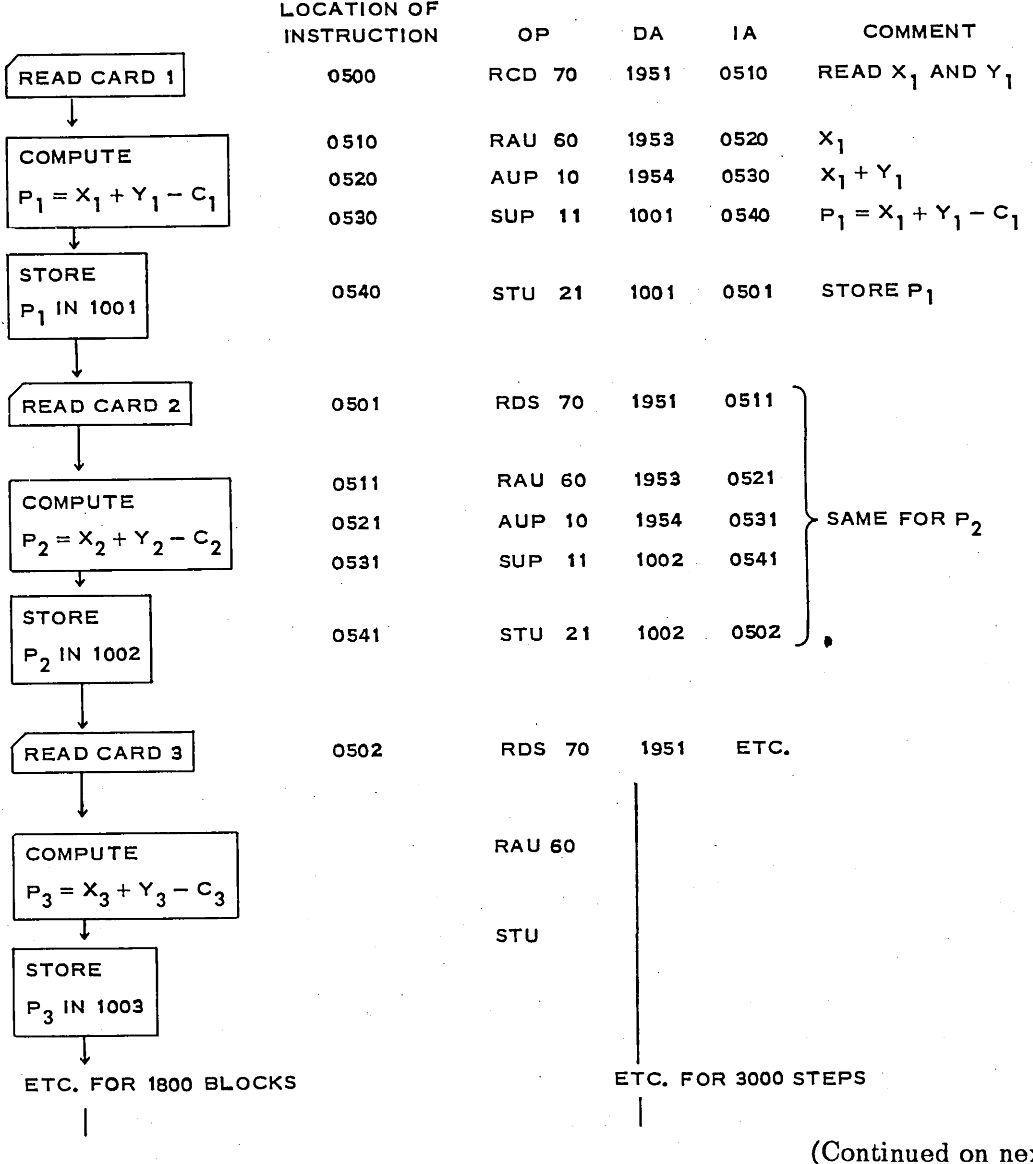
3-2. STEPPING INSTRUCTIONS.

In previous problems the data have been read from cards as needed, and punched into cards as soon as they were computed. In actual programming, part of the data may be on cards and part on the drum from a previous calculation. A very simple example, which serves to introduce the next topic, would be to compute $S_i = X_i + Y_i - C$ where the single constant C is already on the drum (in location 1001) while X_i, Y_i are read into the locations 1953 and 1954 from words 3 and 4 of cards. The S_i are to be punched into output cards as computed. A possible program will be the following:



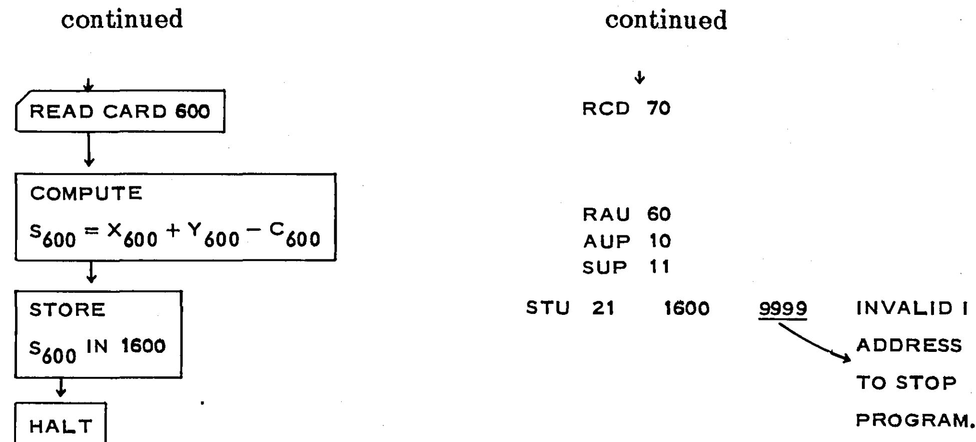
LOCATION OF INSTRUCTION	OP		DA	IA	COMMENT
0500	RCD	70	1951	0510	READ X_i AND Y_i
0510	RAU	60	1953	0520	X_i
0520	AUP	10	1954	0530	$X_i + Y_i$
0530	SUP	11	1001	0540	$S_i = X_i + Y_i - C$
0540	STU	21	1982	0501	STORE S_i
0501	PCH	71	1977	0500	PUNCH S_i

This is not a particularly efficient program, but it serves as an introduction to our main problem. In the above program the same C was subtracted from each $X_i + Y_i$. Consider now the more realistic problem in which 600 values of C_i are stored in drum locations 1001 to 1600, and it is desired to compute $P_i = X_i + Y_i - C_i$ obtaining C_i from the drum and X_i and Y_i from words 3 and 4 of cards, as before, and to store the P_i in locations 1001 to 1600 (i.e., store P_k in drum location which had contained C_k). Your first thought may be:



LOCATION OF INSTRUCTION	OP		DA	IA	COMMENT
0500	RCD	70	1951	0510	READ X_1 AND Y_1
0510	RAU	60	1953	0520	X_1
0520	AUP	10	1954	0530	$X_1 + Y_1$
0530	SUP	11	1001	0540	$P_1 = X_1 + Y_1 - C_1$
0540	STU	21	1001	0501	STORE P_1
0501	RDS	70	1951	0511	SAME FOR P_2
0511	RAU	60	1953	0521	(SAME FOR P_2)
0521	AUP	10	1954	0531	(SAME FOR P_2)
0531	SUP	11	1002	0541	(SAME FOR P_2)
0541	STU	21	1002	0502	(SAME FOR P_2).
0502	RDS	70	1951	ETC.	
	RAU	60			
	STU				
	ETC. FOR 3000 STEPS				

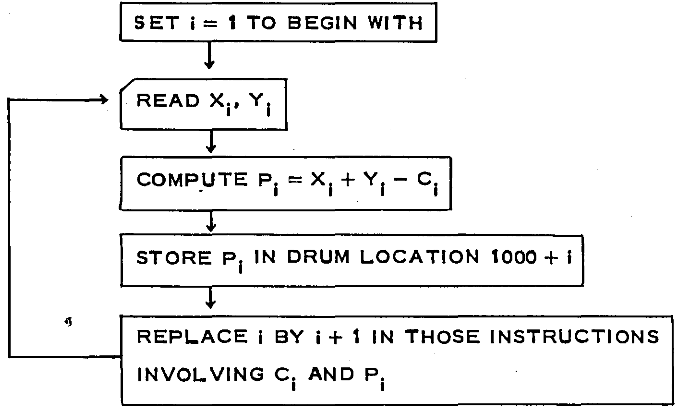
(Continued on next page)



However, this will be a long, long program—containing about 3000 instructions—which is not feasible with only 2000 drum locations available, especially since 600 are needed for storing the C_i and P_i values. Even if it were possible to store this long program, it would be very tedious to write, since much of the work is repetitious. Repetitious work is the very thing the 650 was built to do! Let us use our ingenuity to make the 650 do the repetitious part of its own programming. Essentially, the steps of this program are very similar except that the value of i is changed in each successive set of five instructions.

EXAMPLE 1:

It seems reasonable, then, to replace the 1800 blocks (3000 instructions) of the above diagram with a program of the following type where the program "loops" around after modifying the instructions to repeat the same operations with new data.



The first three blocks of the loop are similar to the first three blocks in the other flow diagram, and lead to a similar program, namely, the five instructions:

LOCATION OF INSTRUCTION	OP	DA	IA	COMMENT
0500	RCD 70	1951	0510	READ X_i AND Y_i
0510	RAU 60	1953	0520	X_i
0520	AUP 10	1954	0530	$X_i + Y_i$
0530	SUP 11	[1001]	0540	$P_i = X_i + Y_i - C_i$
0540	STU 21	[1001]	0545	STORE P_i

Now we use our ingenuity. The addresses enclosed in [ ] should be increased by one each time the program reuses the set of instructions. Since an instruction is, after all, a 10-digit number, we may operate on it arithmetically, just as with data. One plan is to load the 10-

digit number (instruction) into the accumulator, and then add 00 0001 0000 to it and store the result back into the proper drum location. We do this by continuing the program (assuming that 00 0001 0000 is in drum location 0906).

LOCATION OF INSTRUCTION	OP	DA	IA	COMMENT
0545	RAU 60	0530	0501	INSTR. FROM 0530 INTO ACCUM.
0501	AUP 10	0906	0511	INCREASE DA BY ONE
0511	STU 21	0530	0521	STORE NEW INSTR. INTO 0530
0521	RAU 60	0540	0531	SIMILARLY
0531	AUP 10	0906	0541	STEP INSTR.
0541	STU 21	0540	0500	IN 0540

With the given program, the computation proceeds as fast as the cards can be read. The new program requires a dozen program steps instead of 3000, and the 650 now has been set up to do the tedious portions of the programming. The machine is not smart; you, who gave it its instructions, are smart. The machine merely follows accurately the program you have set to develop its further instructions. The thoughtful reader may have begun to gain some small glimmering of the power that lies ahead. The use of symbolic assembly programs (Chapter 5), interpretive systems (Chapter 7), and compilers (Chapter 8) absorbs much programming drudgery. First, however, the *fundamental principles must be learned*, as they are common to all systems.

Problem Set 3-2

1. Write out the program of Section 3-2 which involves the loop. Fill in the distributor and accumulator entries. Decide whether or not the program will run *before* you put it on the 650. Debug as necessary. Be sure you "initialize" it by setting i = 1 to start with.

2. What happens in the problem just discussed if an overflow occurs? Can an overflow occur if the data are as specified? What data specifications will insure you against overflow?

3. (a) Reprogram Problem 3 of Set 1-11 with the input bunched 3 sets per input card plus an identifying number, and the output bunched 6 answers per card plus both identifying numbers.

(b) As an extra-credit problem, devise a method whereby 4 sets of data can be put on each input card and 8 answers on each output card, plus some identifying number on each so the problems and answers may be collated later.

4. Explain the action of the 650 upon receiving the instruction 00 0000 0000 (a) from location 0924, (b) from location 0000.

5. What will the following program do, if anything?

LOCATION OF INSTRUCTION	OP	DA	IA
1951	60	1952	1953
1952	15	8001	8003
1953	15	1954	8003
1954	00	0000	0001

Punch these on *one* (load) *card* in words 1, 2, 3, 4 and set console to 70 1951 9999 and start program.

3-3. LOOPING AND STEPPING.

The word "loop" is used to describe a portion of a program which is repeated within the program and hence the flow chart contains a closed loop. It is quite usual to modify the instructions each time a loop is repeated. Even intermediate programming introduces loops within loops within loops. Indeed, it is the ability to modify looped instructions which provides the stored-program machine with much greater versatility than is possessed by an externally programmed machine. Care should be taken to program an inner loop as efficiently as possible, even though it may seem to involve only a few steps in a much longer program. In actual computation, more than half of the computing time is often spent in the small loop! As a matter of fact, a 2000 instruction program which completely filled the drum could be executed in less than 2 seconds if no loop were involved. It is not surprising that a great many different techniques have been developed for looping. The one described in Section 3-2 adds the constant 00 0001 0000 to certain instructions to modify the data address before repeating the loop.

Another important technique which may save time is to "step" the first instruction (called key instruction), and then to develop the other modified instructions by adding constants to the key instruction.

In the program discussed in Section 3-2, only two instructions are to be changed:

```
0530   11[1001]   0540 \
                        >----> DIFFERENCE = Δ = 10 0000 0005
0540   21[1001]   0545 /
```

If $\Delta = 10\ \ 0000\ \ 0005$ is added to the instruction in 0530, the instruction in 0540 will result, no matter what common number is used to replace [1001]. It is, therefore, feasible to step the instruction in drum location 0530, as before, and then replace the last three steps (stepping the instruction in 0540) by the two steps given below, since the accumulator already contains the result of stepping the instruction from 0530. Assume 0926 contains $\Delta = 1000000005$.

```
0521   AUP 10   0926   0531
0531   STU 21   0540   0500
```

In this particular problem no saving in time is achieved (Why not?—consider read speed.), but in other programs the savings may be very worthwhile. The technique is also important in instances when stepping is not involved.

Readers who are seriously interested in becoming expert programmers should read the IBM 650 operation manual on operation 22 SDA and consider the possibility of using this operation in the above program.

Problem Set 3-3

1. Write a program which will compute $S_i = X_i + Y_i - C_i$ where the X_i and Y_i are on cards and the C_i on the drum, as in the example of Section 3-2. When 6 values of S_i have been computed, punch the 6 values plus a card number onto output cards. Let the card number be the i of the last S_i on that card. Will all of the answers be punched out with your program? Would they be, if there were 500 rather than 600 values of S_i?

2. Write a program which will find the sum of 1000 10-digit numbers which are stored in drum location 0501-1500 and which will punch this total on a single card. Be sure that your program punches the entire total, which may easily contain a dozen digits.

3. (a) Design a program which will form the sum of one 1, two 2's, three 3's, four 4's, five 5's, . . . , one hundred 100's, . . . , k k's for any preassigned number $k = 0000000kkk < 10^3$

where k is placed in location 1954. In other words, form $S_k = 1 + (2 + 2) + (3 + 3 + 3) + (4 + 4 + 4 + 4) + \ldots + (k + k + k + k + \ldots + k)$.

(b) Does the following program yield the same final result? If so, which takes longer to compute? How does each work? Assume that OP 14 divides the 20-digit number in the accumulator by the 10-digit number whose address is specified by the DA, leaving the quotient in the lower half of the accumulator.

```
0400   RAU 60   1954   0410   k
0410   AUP 10   1915   0420   k + 1 [NOTE CONSTANT LISTED AT BOTTOM]
0420   STU 21   0415   0430
0430   AUP 10   1954   0440   2k + 1
0440   MPY 19   8001   0401   (2k + 1) k
0401   RAU 60   8002   0411
0411   MPY 19   0415   0421   (2k + 1) k (k + 1)
0421   DIV 14   1926   0431   (2k + 1) k (k + 1)/6
0431   STL 20   1977   0402
0402   PCH 71   1977   9999

CONSTANTS

   1915      00 0000 0001
   1926      00 0000 0006
   0415      TEMPORARY STORAGE (k + 1)
```

(*Hint:* Consult a mathematical handbook for a convenient formula.)

4. What happens in the following program? Will either eventually overflow? Why? Assume location 1955 contains 00 0000 0010.

```
(a) 0400   RAU 60   1955   0410
    0410   MPY 19   8003   0420
    0420   RAU 60   8002   0410
(b) 0500   RAL 65   1955   0510
    0510   ALO 15   8001   0510
```

5. Describe the effect of the following program: Explain how many times the contents of 0502 is added and what, if anything, is punched. How many output cards are there for each input card? Drum location 0502 contains a 7-digit positive number Y = 000yyyyyyy$^+$.

```
1400   RCD 70   0501   1410
1410   LDD 69   0501   1420
1420   STD 24   0827   1430
1430   RAU 60   0671   1440   SEE CONSTANTS AT
1440   SUP 11   0672   1401   END OF PROGRAM.
1401   ALO 15   0502   1471
1471   NZU 44   1440   1421
1421   STL 20   0828   1431
1431   PCH 71   0827   1400

             CONSTANTS

        0671    00 0000 0005
        0672    00 0000 0001
```

3-4. THE COUNT BOX.

Let us now assume that all the data are on the drum, and that the results are to be placed on the drum—i.e., that this is but one small fragment of a larger program.

X_i IN 1101 TO 1190
Y_i IN 1201 TO 1290

where X_i are 5-digit integers 00000xxxxx and the Y_i are 4-digit integers 000000yyyy. Compute

$$T_i = X_i Y_i - Y_i^2$$

and store T_k in location [1700 + k]. It is quicker for the 650 to compute (X − Y)Y than to compute $XY - Y^2$, and we shall use the quicker form. A possible computation block for T_i would be:

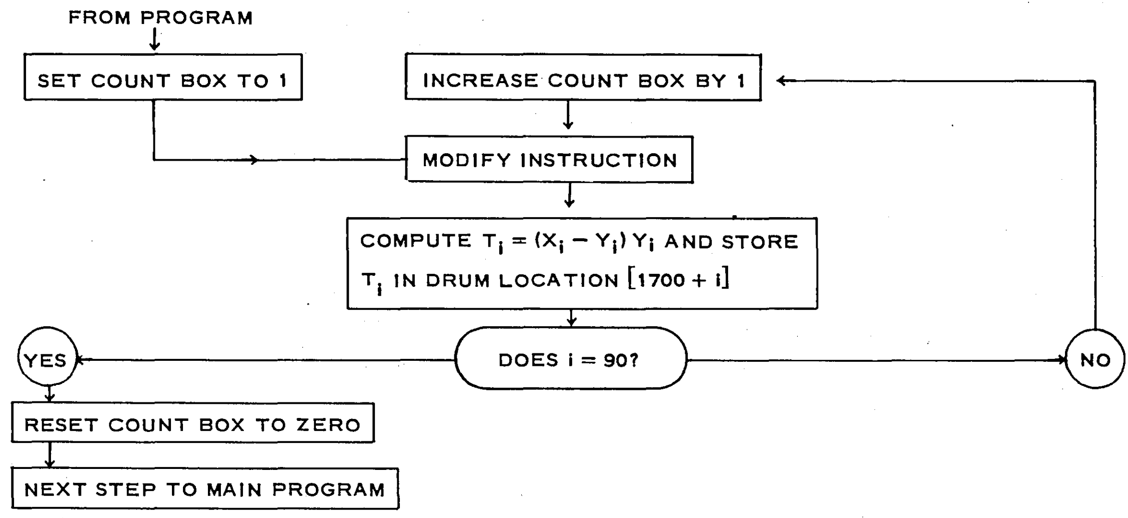
```
0700   RAU 60   [1101]   0710         Xi
0710   SUP 11   [1201]   0720         Xi − Yi
0720   MPY 19    8001    0730        (Xi − Yi) Yi
0730   STL 20   [1701]   0740   STORE LOWER 10-DIGITS
                                OF PRODUCT
```

Since the 20-digit product contains at least 10 leading zeros, it is unnecessary to examine the upper half of the accumulator. It is, of course, necessary to modify (step) the instructions before T_2 is computed.

In previous examples, the computation stopped when it was finished, either because an invalid next instruction address was given, or because the 533 unit received a 70 RCD instruction when no further cards were in the read hopper. Since this is merely a program fragment, neither course is desirable. Instead we shall make a count box ("B box") or *index register*, which will register k when T_k is computed and will test after each computation to see if k = 90. If k does not equal 90, the instruction will be modified and the loop continued. If k does equal 90, the next instruction in the major program will be taken. For reasons to appear later, we shall make a count box of the form 00 00kk 0000 in drum location 1915 by adding 1's of the form 00 0001 0000 from drum location 1925:

```
0510   RAU 60   1915   0520   COUNT BOX INTO ACCUMULATOR
0520   AUP 10   1925   0530   ADD 1
0530   STU 21   1915   0600   STORE NEW COUNT INTO 1915
```

A possible flow diagram follows.



It is quite feasible to make the desired modification of instructions by adding 00 0001 0000 to those instructions containing a [ ] in the data address position, but we prefer to demonstrate another method which has proved desirable in repeated subloops. The three instructions to be modified are:

0700	60	1100 + i	0710	$\Delta = -48$ 9899 9990
0710	11	1200 + i	0720	$\Delta = +09$ 0500 0020
0730	20	1700 + i	0740	

Store the number 60 1100 0710 on the drum, say in location 1905. If the count box 00 00ii 0000 is added to 60 1100 0710, the first desired instruction 60 [1100 + i] 0710 results. In order to do this easily, we made the count box (in location 1915) by adding 1's in the fifth digit position—i.e., 1 = 00 0001 0000. The desired i will always be in position such that 60 1100 0710 plus 00 00ii 0000 is the instruction needed in drum location 0700. The instruction desired in 0710 may be obtained by adding $\Delta = -48$ 9899 9990 (from drum location 1935) to 60 [1100 + i] 0710 which is in the accumulator. Finally, the instruction needed in 0730 is obtained by adding $\Delta = +09$ 0500 0020 (from drum location 1906) to 11 [1201 + i] 0720, now in the accumulator, using the technique described in Section 3-3.

0600	RAU 60	1905	0610	MASTER INSTRUCTION TO ACCUMULATOR
0610	AUP 10	1915	0620	ADD COUNT BOX
0620	STU 21	0700	0630	STORE NEW 0700 INSTRUCTION
0630	AUP 10	1935	0640	ADD
0640	STU 21	0710	0601	STORE NEW 0710 INSTRUCTION
0601	AUP 10	1906	0611	ADD
0611	STU 21	0730	0700	STORE NEW 0730 INSTRUCTION

This method has the advantage of being self-setting if the fragment is reused later in the program for different data. We conclude by making the test, "Does i = 90?" In this we assume that drum location 1924 contains the constant 00 0090 0000, and that drum location 1946 contains all zeros, 00 0000 0000.

0810	RAU 60	1915	0820	COUNT BOX INTO ACCUMULATOR
0820	SUP 11	1924	0830	SUBTRACT 90
0830	NZU 44	0510	0840	BRANCH ON NON-ZERO UPPER
0840	LDD 69	1946	0800	RESET COUNT BOX TO ZERO
0800	STD 24	1915	NEXT INSTRUCTION IN MAIN PROGRAM	RESET COUNT BOX TO ZERO

The reader is asked to assemble this program on a 650 planning sheet, improving it where possible. For example, a step is saved by using 10 1905 0620 in place of the instruction now given in 0600, since the count box is already in the accumulator. Doing such assembly work is an important part of learning to program. Be certain you understand each step.

Problem Set 3-4

1. Assemble the program just discussed and debug it. It may be necessary to modify some of the instruction addresses to connect the program pieces.
2. If drum location 0740 contained the instruction 00 0000 0810, what effect would it have on the program suggested in the text of this section?
3. Write a program of your own using a loop instruction.

3-5. ANOTHER TERMINATING TECHNIQUE.

As a subloop of a larger program we have 45 sets of values of each of 4 variables (W_i, X_i, Y_i, Z_i) stored on the drum in the locations shown:

W_i stored in 1101 $\longrightarrow$ 1145 (i.e., W_k in 1100 + k)
X_i stored in 1201 $\longrightarrow$ 1245 (i.e., X_k in 1200 + k)
Y_i stored in 1301 $\longrightarrow$ 1345 (i.e., Y_k in 1300 + k)
Z_i stored in 1401 $\longrightarrow$ 1445 (i.e., Z_k in 1400 + k)

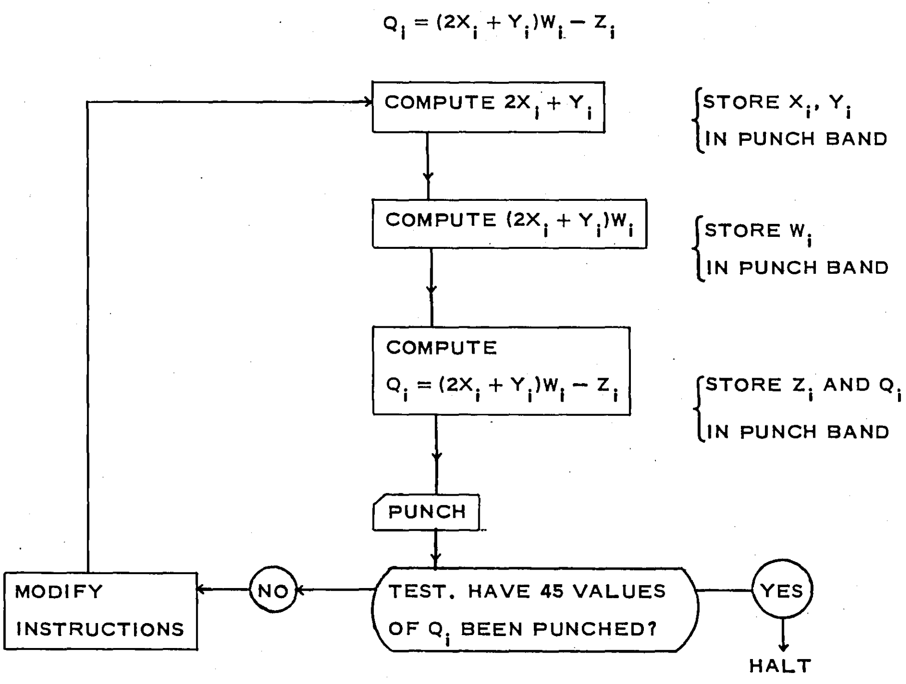
Each W_i is a 3-digit integer (whole number) stored in the form 00 0000 0WWW. Each of the numbers X_i, Y_i, Z_i is an integer having not more than 6 significant digits. Each is stored in the word form 00 00dd dddd.
Our problem is to form

$$Q_i = (2X_i + Y_i)W_i - Z_i$$

and to punch 45 output cards of the following form:

	W_i	X_i	Y_i	Z_i		Q_i	
COLUMN NOS.	0 10	11 20	21 30	31 40	41 50	51 60	61 80

In programming this problem, it will either be necessary to write a separate program to compute each Q_i (since the W_i, X_i, Y_i, Z_i are in different locations for various values of i), or else to loop, modifying the instructions before repeating them. The latter method will be used.



We could make a "count box" which would register k when Q_k is computed, and test after each card is punched to see if k = 45. However, since the instructions must be modified, it is feasible to check whether or not the modified instruction is in its final form. In this case one of the final instructions will be

SLO 16 1445 0240 (Subtract Z_{45})

Place the instruction 16 [1400 + k] 0240 into the accumulator and subtract 16 1445 0240. If the result is zero, 45 values of Q_i have been punched; if not, repeat the computation loop.

Before considering the program, let us examine the top lines on a 650 planning sheet.

Let us elect to punch from drum locations 1827–1834. This decision is quite arbitrary at this stage, except that we must use permissible punching locations and we must *not* use 1101 ⟶ 1145, 1201 ⟶ 1245, 1301 ⟶ 1345, 1401 ⟶ 1445. (Why not?)

Take a fresh 650 planning chart. Note that across the top is a line reading "Storage Exit Words." We shall use only 8 of these words with our control-panel wiring, so cross out words 9 and 10. Fill in the form as follows

STORAGE-EXIT WORD	1	2	3	4	5	6	7	8
MEMORY ADDRESS	1827	1828	1829	1830	1831	1832	1833	1834
OUTPUT CARD	W_i	X_i	Y_i	Z_i		Q_i		

Recall that the distributor (8001), lower accumulator (8002), and upper accumulator (8003) may be used as valid addresses for data and instructions, just as are drum locations. (A *Store* into operation such as 24 STD may *not* be used to store into an 800n address. This is accomplished by another instruction. For example, 60 8001 IA (RAU) will, in effect, store the contents of the distributor into the upper accumulator.)

Enclose in square brackets those portions of instructions which will be modified later. (It is convenient to use *red,* if color is available.)

LOCATION OF INSTRUCTION	OP		DA	IA	8003 UPPER	8002 LOWER	8001 DIST.
0200	RAU	60	[1201]	0205	0000 X	0000000000	0000 X
0205	STD	24	1828	0210	0000 X	0000000000	0000 X
0210	AUP	10	8001	0215	000 2X	0000000000	0000 X
0215	AUP	10	[1301]	0220	000 2X + Y	0000000000	0000 Y
0220	STD	24	1829	0225	000 2X + Y	0000000000	0000 Y
0225	MPY	19	[1101]	0230	0000000000	(2X + Y)W	00000000 W
0230	STD	24	1827	0235	0000000000	(2X + Y)W	00000000 W
0235	SLO	16	[1401]	0240	0000000000	(2X + Y)W − Z	0000 Z
0240	STD	24	1830	0245	0000000000	(2X + Y)W − Z	0000 Z
0245	STL	20	1832	0201	0000000000	(2X + Y)W − Z	0000 Q
0201	PCH	71	1827	0216			

The four computation blocks are complete. We have computed $Q_i = (2X_i + Y_i)W_i - Z_i$, and have stored W_i, X_i, Y_i, and Q_i in the designated punch locations and punched the output card. The reader should study this portion of the program until he understands it, before continuing.

Let us assume that drum location 1926 contains the constant 16 1445 0240. We test to see if the modified instruction in drum location 0235—namely, 16 [1400 + i] 0240—is equal to 16 1445 0240.

0216	RAU	60	0235	0221
0221	SUP	11	1926	0231
0231	NZU	44	0241	NNNN NEXT INSTRUCTION IN MAJOR PROGRAM

This sends the program to 0241 for its next instruction if i < 45, and to the next instruction in the major program if 45 values of Q_i have been punched.

Our next task is to step the four data addresses containing [ ]. This is left as an exercise for the student. Suitable methods were explained in Sections 3–2 and 3–3.

Problem Set 3-5

1. Complete the program of this section using the stepping method of Section 3-2. Debug the program. Check to see that no storage location is used for two different instructions—a common error.

2. Complete the program of this section using the modified instruction method of Section 3-3. Debug the program.

3. Assume that storage location 1777 contains a particular value of x, while 1778 contains e^x, location 1779 contains e^{-x} for the specified value of x. Prepare a program which will punch the following data from punch locations 1777-1781.

LOCATION	1777	1778	1779	1780	1781
CONTENTS	X	e^x	e^{-x}	SINH X	COSH X

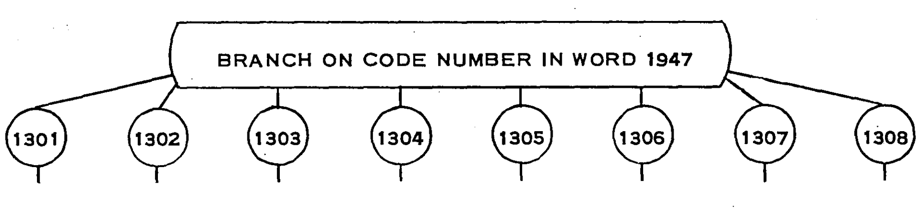
4. Bring a *simple* problem of your own to class and block diagram it.

5. Spend 5 minutes explaining your problem and block diagram to another member of the class, and then listen to his explanation of your problem to see if he understood it. Do not interrupt him!

3-6. BRANCHING ON A CODE NUMBER (Optional.)

It is often desirable to be able to direct a program into one of several channels depending upon a code number in the input card. This could be done by machine sorting the cards before putting them on the 650, or by board wiring, but it is often more convenient to use the 650 itself to make the switch, by having the key number appear as 00 0000 00kk on the drum. (If you are short of card space, board wiring can place the key number into word 9 or 10 of the drum read band using only two card columns.) If the key number is added to a master instruction before it is performed, the next instruction will depend upon the key number, and the program may be branched into any of several paths. Other variations will occur to you as you program. This book is not a list of tricks, but rather a study of *methods*. Develop tricks of your own. A good programmer first decides what he would like to have the machine do, and then invents tricks to do it. New tricks are constantly being invented and the better you understand the 650, the more ingenious you can be.

A program fragment will make the method clear. We assume that drum location 1937 contains 00 0000 1300, and that the code number 00 0000 000k with k = 1, 2, 3, 4, 5, 6, 7, or 8 is in drum location 1947.



LOCATION OF INSTRUCTION	OP	DA	IA	COMMENT
0432	RAL 65	1937	0442	MASTER INSTR.
0442	ALO 15	1947	8002	ADD CODE NO.

The next instruction received will be in the accumulator 8002—namely, 00 0000 130k—and the program will branch to the desired location.

Many variations are possible.

Problem Set 3-6

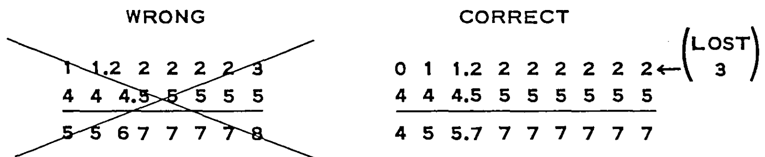
1. Devise a step in a program which will branch in drum location 0464 to any of 5 positions 120k, with k = 1, 2, 3, 4, 5 depending upon which pay code 1, 2, 3, 4, 5 is stored in drum location 1954.

2. (a) In devising a program to play tic-tac-toe, it is desirable to have several "branches into one of nine positions." Devise such a branch.

3-7. SCALING.

Thus far, all our problems have dealt with integers. The 650 deals only with integers—but really, so does all computation. We now study modifications of the program needed when elements are decimal fractions rather than integers. A method of describing the location of the decimal point without great verbiage is convenient. One common notation is to speak of 473.2908600 as a "3—7 number," meaning that there are three significant digits before the decimal point and seven after it; xxx.xxxxxxx is another common notation for the same concept. The latter is particularly useful if some of the digits are known to be zeros—for example, xxx.xxxxx00. This does *not* imply that the digits represented by x's are all identical, but only that the decimal point follows the third digit and that the ninth and tenth digits are always zeros. The number 004.3200000 is of the form xxx.xxxxx00. So is 989.9743400.

If a 2—8 number 11.22222223 and 3—7 number 444.5555555 are to be added on the 650, it is necessary to align the decimal points, just as one would do if he were using pencil and paper.



Since the 650 has no decimal point, it is quite willing to add either way. It is you, the programmer, who must be alert. Even problems in which the data are all in integers will often involve decimal output, since a division is necessary—in computing an average, for example.

We must find a way of aligning decimal points for add and subtract operations. In other words, a *shift* operation is needed. The 650 is provided with four different types of shift operations. In the problem under discussion, all that is needed is to shift the 2—8 number 11.22222223 one digit to the right, giving 011.2222222(3). *Shifting is done in the accumulator only.* Since only 10 digits can be handled in each half of the accumulator, a "shift to the right one" instruction will place the low-order digit of the upper accumulator into the high-order position of the lower accumulator, while the low-order digit of the word in the lower accumulator is lost completely. Zeros are automatically supplied into the "blanks" which occur on a shift operation, but the reader should remember that he is, as always, dealing with one 20-digit accumulator, not two separate 10-digit accumulators.

VVVVVVVVVV	LLLLLLLLLL	±	before Shift Right one
0VVVVVVVVV	VLLLLLLLLL(L)	±	after SRT 0001

In a shift operation code, the units position of the Data Address indicates *the number of places* to be shifted (a new use of the DA). Any valid data address 0000–1999, 8000, 8001, 8002, 8003 may be used, since *only the units digit* is analyzed by the machine.

30 000N xxxx SRT (Shift Right) will Shift the 20 digit accumulator to the Right N digits, for $0 \leq N \leq 9$. If N = 0, no shift results.

31 000N xxxx SRD (Shift and Round) will Shift the 20-digit accumulator Right N digits for $1 \leq N \leq 10$, and will "Round" the new units digit by adding 5 to the last digit lost. If N = 0, a 10-digit shift and round results.

35 000N xxxx SLT (Shift Left) will Shift Left N digits for $0 \leq N \leq 9$. If N = 0, no shift results. Numbers shifted off the left end are lost, but the overflow circuit is not activated.

36 000N xxxx SCT (Shift and Count). This useful operation is discussed in more advanced work. Consult the IBM programmers manual if interested.

The shift operations are essential in "scaling" numbers for arithmetic operations, but they also are useful in other capacities. For example, if X is a 5-digit number located in the lower 5 positions of drum location 1851 (i.e., 00000xxxxx) and Y is a 5-digit number 00000yyyyy similarly located in location 1852, then, if both X and Y are positive, the program

```
----   RAL  65   1851   ----
----   SLT  35   0005   ----
----   ALO  15   1852   ----
----   STL  20   1527   ----
```

will place both X and Y in punch location 1527 as xxxxxyyyyy. This technique permits more output on one card even if the "standard board" is used.

The usual axiom in scaling is "expect the worst and plan for it." Thus X + Y = Z becomes

```
 00XX.XXXXXX
 0000.0000YY
 -----------
 0ZZZ.ZZZZZZ
```

Even though the sum *may* have two (or more) initial zeros, only one can be guaranteed. (Why?) Scaling is another spot where the person who knows most about the problem can do the best job. If, for example, he knew that $X \leq 0099.999900$, it would then follow that Z could be guaranteed to have two leading zeros instead of one. (Why?)

The more that is known about the nature of the data, the more efficient the resulting program is likely to be.

Problem Set 3-7

1. Assume that the following is on the drum:

X_i = 0xx.xx00000 in [0900 + i] for i = 1, 2, . . . , 45.
Y_i = .yyy0000000 in [1000 + i] for i = 1, 2, . . . , 45.

(a) Design a program which will punch X_i, Y_i, $X_i + Y_i$, and $X_i - Y_i$ from drum locations 1977, 1978, 1980, 1982, respectively.

(b) Design a program which will also punch $X_i + (i \cdot Y_i)$ from drum location 1979. Consider the problem of possible overflow.

2. Drum locations 0201–0300 contain one hundred 9-digit values of X_i. The corresponding Y_i and Z_i are stored in drum locations 0301–0400 and 0401–0500 respectively, each being a 10-digit number. Make a block diagram which will store $(X \cdot Y \cdot Z)$, rounded off to 10 digits, in locations 0701–0800. (*Hint*: Multiply X_i by Y_i. Since a 19-digit product is developed, it is necessary to reduce this to a 10-digit product before multiplying by Z_i. [Why?]) This may be done as follows where drum location 1999 contains the constant 5000000000.

```
----   SLT 35   0001   ----
----   ALO 15   1999   ----
----   RAU 60   8003   ----
----   MPY 19          ----
```

It will be desirable to modify (step) instructions in this problem. Could SLT 35 0002 be used?

3. The accumulator contains the 20-digit number

11111223334445566666+

before each of the following commands is received. Explain what it will contain after the command is executed.

```
(a) 30 0002 1956     (h) 35 0002 1956
(b) 30 8002 1956     (i) 35 0010 1956
(c) 30 0010 1956     (j) 35 0742 1956
(d) 31 0002 1956     (k) 31 0007 1956
(e) 31 8002 1956     (l) 31 0008 1956
(f) 31 0010 1956     (m) 31 0009 1956
(g) 31 0000 1956     (n) 30 1492 1956
```

3-8. SCALING IN MULTIPLICATION.

The scaling problem in multiplication is not one of aligning decimal points, but rather of *keeping track* of decimal points. For example:

```
                                    DEC. PT.                                      DEC. PT.
MPY      3 3 3.3 3 3 3 3 3 3 3     {3—7}          MPY    3 3.3 3 3 3 3 3 3 3 3    {2—8}
         1 1 1 1 1 1.1 1 1 1       {6—4}                 1.1 1 1 1 1 1 1 1 1 1    {1—9}
0 3 7 0 3 7 0 3 7.0 2 9 6 2 9 6 2 9 6 3           0 3 7.0 3 7 0 3 7 0 2 9 6 2 9 6 2 9 6 3
                                   {9—11}                                         {3—17}
```

In the first example a 3—7 number is multiplied by a 6—4 number, giving a 9—11 number as product. In the second example a 2—8 number is multiplied by a 1—9 number, giving a 3—17 number as product.

In general, if an h—k number is multiplied by a l—m number, the result will be an (h + l)—(k + m) number, if the correct number of leading (nonsignificant) zeros is supplied.

EXAMPLE 1:

In computing an hourly payroll, it is necessary to multiply the number of hours worked by the hourly pay rate, and then round to the nearest cent. Assume

DRUM LOCATION	CONTENTS	TYPE	FORM	SAMPLE
1201	HOURS WORKED	{9—1}	0000000hhh	000000024.7
1214	PAY RATE/HR	{7—3}	000000rrrr	0000001.795

Obtain the amount earned to the nearest cent, as an 8—2 number in drum location 1977. Upon multiplying the numbers as they stand, a 16—4 number results. (Why?) This will be rounded to the nearest cent by using the SRD 31 0002 - - - - operation, giving an 18—2 number, the first 15 digits of which will be zeros. (Check this last statement.) Hence we shall store the lower, which will be in the form 00000eeeee.

LOCATION OF INSTRUCTION	OP	DA	IA
0500	RAU 60	1201	0510
0510	MPY 19	1214	0520
0520	SRD 31	0002	0530
0530	STL 20	1977	XXXX

Let us examine the multiplication 444 · 99 to see how the location of the zeros needed to make the word affects the location of the product.

```
                          0 0 0 0 0 0 0 4 4 4
        MPY               0 0 0 0 0 0 0 0 9 9
0 0 0 0 0 0 0 0 0 0   0 0 0 0 0 4 3 9 5 6
       upper                  lower

                          4 4 4 0 0 0 0 0 0 0
        MPY               0 0 0 0 0 0 0 0 9 9
0 0 0 0 0 0 0 0 4 3   9 5 6 0 0 0 0 0 0 0
       upper                  lower

                          4 4 4 0 0 0 0 0 0 0
        MPY               0 0 0 0 0 9 9 0 0 0
0 0 0 0 0 4 3 9 5 6   0 0 0 0 0 0 0 0 0 0
       upper                  lower
```

Some people use the notation 5/7-3/3 to represent the number 00000xx.000, where the digits before and after the / indicate numbers of leading and terminal zeros. The author finds it simpler to do the analysis without using the notation.

Two basic facts are the 650 computes the product of two 10-digit numbers, obtaining a 20-digit product, and the decimal point can be located using the h—k notation as described above. A little practice on a desk calculator will help you to understand this.

3-9. DIVISION.

In the 650 computers, division is accomplished by repeated subtraction, just as it is on a desk calculator. Scaling precautions are necessary, as on a desk calculator, to prevent developing a quotient having more digits than the 10-digits which the accumulator can handle. For the benefit of readers unfamiliar with the desk calculator, we state the rule. *The absolute value of the divisor must be greater than the absolute value of that portion of the dividend which is in the upper half of the accumulator.*

In division, a 20-digit dividend (the entire accumulator) is divided by a 10-digit divisor (on drum) producing a 10-digit quotient in the lower half of the accumulator. If operation DIV 14 (Divide) is used, the remainder will be found in the upper half of the accumulator. If operation DVR 64 (Divide and Reset Upper) is used, no remainder will be retained. In

```
      Q                       10-digit quotient
    D/N    10-digit divisor / 20-digit dividend
```

many problems the dividend will be a 10-digit or smaller number, and if this is put into the lower half of the accumulator with all zeros in the upper (RAL 65), no overflow trouble will occur unless a division by zero is attempted. (Attempted division by zero always stops the program, no matter how the overflow switch is set.) The use of RAL 65 may at first appear to

provide a very simple "out" on division scaling difficulties. Like most "easy methods" it requires a "price." In this case the price may be the accuracy of the result. One thing that often helps when a number N of fewer than 10 digits is being divided by D is to put the spare zeros in the low-order positions on N but on the high order positions on D. For example, if N = 37.259, place it in the lower accumulator as the 2—8 number 37.25900000 rather than as a 7—3 number 0000037.259. The student who fails to grasp the reason for this will probably be well advised to find a desk calculator (there should be at least one in every 650 lab) and carry out a few divisions. Perhaps your instructor will demonstrate in class. Briefly, here is what happens in machine division *using 3-digit in place of 10-digit words to conserve space,* $Q = \frac{N}{D} = \frac{3.4}{1.2}$.

```
           0 0 2.                 0 2.8                   2.8 3                    overflow
0 1.2|0 0 0|0 3.4     0 1.2|0 0 0|3.4 0     0 1.2|0 0 3|.4 0 0     0 1 2/0 3 4|0 0 0
      upper | lower         upper | lower         upper | lower             upper | lower
                                        overflow
                                  0 1.2/3.4 0   0 0 0
                                        upper   lower

           0 0 0                  0 0 2.                 0 2.8                   2.8 3
1.2 0|0 0 0|0 3.4     1.2 0|0 0 0|3.4 0     1.2 0|0 0 3.|4 0 0     1.2 0|0 3.4|0 0 0
      upper | lower         upper | lower         upper | lower           upper | lower
                                       overflow
                                  1.2 0|3.4 0|0 0 0
                                        upper | lower
```

The question of which answer is the most desirable leads to the most vexing question in computer work,—namely, error analysis.

If the original numbers are each accurate to the nearest .1 (2 significant digits), then the "true value" may lie anywhere in the range:

$$N = 3.4 \pm .05, \quad D = 1.2 \pm .05$$

$$\frac{3.35}{1.25} \leqq \text{"true value } \frac{N}{D}\text{"} \leqq \frac{3.45}{1.15}$$

$$2.68 \leqq \text{"true value } \frac{N}{D}\text{"} \leqq 3.00$$

The mere fact that the 650 gives the answer as 2.833333333 does not alter the fact that the true value T may lie anywhere in the range $2.68 \leqq T \leqq 3.00$, and that it is quite likely that the 10-digit quotient is *incorrect* in its second digit, as well as in all following digits. This is not a problem of the 650, but is a problem in the mathematical theory of error analysis which pervades all arithmetic involving approximate numbers, no matter how the arithmetic is done. We shall have more to say about this later, but prefer to discuss the matter of scaling first.

In general, if a 16—4 number is divided by a 9—1 number, the result is a 7—3 number.

The decimal point in a product was located by using addition with the h—k notation. In a similar fashion, the decimal point in a quotient may be located by using subtraction and the h—k notation.

```
16—4
 9—1
----
 7—3
```

If the problem of scaling seems a bit difficult at this point, take heart, since there are three ways in which much of the difficulty of scaling can be sidestepped:

1. By careful positioning of the data on the input cards, or by adequate use of board wiring.
2. By use of floating-point arithmetic (see Chapter 4).
3. By use of the "shift and count" operation 36 SCT.

The latter is probably the most satisfactory, although the present trend is to favor floating-point arithmetic because of its simplicity in programming. In business work and data handling, on the other hand, scaling is usually simple enough so that direct scaling is used.

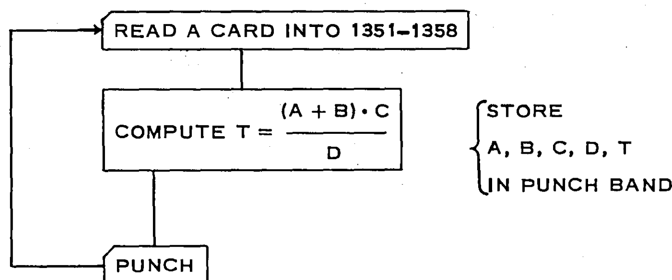
3-10. $\frac{(A + B) \cdot C}{D} = T.$

EXAMPLE 1:

As an example of a problem involving multiplication and division, we program

$$\frac{(A + B) \cdot C}{D} = T$$

assuming that the input cards are the 600 cards of the data deck and A = word 1 (000xxxxxxx); B = word 3 (00000xxxxx); C = word 2 (0000000xxx); D = word 4 (00000xxxxx) where all are 10—0 numbers (integers). Later we shall consider the same problem with A, B, C, D decimal fractions.



Since A is a 7-digit number and B is a 5-digit number, (A + B) may contain 8 digits. When one 8-digit number is multiplied by a 3-digit number, an 11-digit $(A + B) \cdot C$ may develop. This is of no concern, since we are not storing the sum and it is unlikely that the 5-digit number D is smaller in absolute value than the eleventh digit—i.e., that part of the dividend in the upper.* We shall not worry about division overflow here. If an overflow does occur, the overflow circuit will be activated and the program will stop.

The reader should fill in a 650 planning chart, giving particular attention to the location of nonzero digits in the accumulator columns for the following program. Fill in the storage entry and storage exit words as well.

*The reader may feel that a 5-digit number is *never* smaller in absolute value than a 1-digit number, but that will be because he has overlooked the fact that the 5-digit number may have four leading zeros, 00001, or even all zeros, 00000.

LOCATION OF INSTRUCTION	OP	DA	IA	COMMENT
0101	RCD 70	1351	0102	READ
0102	RAU 60	1351	0106	A
0106	STD 24	1377	0132	STORE A FOR PUNCH
0132	AUP 10	1353	0108	A + B
0108	STD 24	1378	0133	STORE B FOR PUNCH
0133	MPY 19	1352	0109	A PLUS B TIMES C
0109	STD 24	1379	0134	STORE C FOR PUNCH
0134	DVR 64	1354	0110	DIVIDE BY D
0110	STD 24	1380	0135	STORE D FOR PUNCH
0135	STL 20	1382	0139	STORE RESULT T FOR PUNCH
0139	PCH 71	1377	0101	PUNCH

Problem Set 3-10

1. Fill in a 650 planning chart for the above program, showing the accumulator and distributor columns.

2. Flow chart and program: $T = \frac{(A + B)\,C}{D}$ where

$$A = 000xxxx.xxx$$
$$B = 0000xxxxx.x$$
$$C = 0000000x.xx$$
$$D = 00000xxx.xx > 100$$

Be sure to pay careful attention to the scaling and indicate where the decimal point will be in the answer.

3. Reprogram Problem 2 with D > 1 in place of D > 100. What difference does this make on the likelihood of overflow? Can you plan for this possibility?

4. The following represents the contents of the accumulator, distributor, and an appropriate drum location before the instruction occurred. What will it contain after the instruction is carried out?

	INSTRUCTION OP CODE	DA	ACCUMULATOR UPPER(8003)	LOWER(8002)	DISTRIBUTOR (8001)	DRUM STORAGE LOC.	CONTENTS
(a)	MPY 19	1942	+9922222221	0000000000	+2222222221	1942	−0009220000
(b)	DIV 14	0135	+0625000000	0000000000	+0000000094	0135	+1250000000
(c)	DIV 14	0135	+6250000000	0000000000	+0000000087	0135	+1250000000
(d)	DIV 14	0135	+0000625000	0000000087	+0000000087	0135	−1250000000
(e)	DVR 64	0135	−0000625000	0000000087	+0000000087	0135	+1250000000
(f)	DVR 64	0742	+0000625000	0000000087	0000000087	0782	0000000125
(g)	SRT 30	0004	+0000700000	0000086542	+0000700000	0697	+0000086542
(h)	SLT 35	0003	+0000700000	0000086542	+0000700000	0697	+0000086542
(i)	SLT 35	0002	+1234500000	0000006000	+0000006000	0697	+0000006000

5. What sets of instructions will take the accumulator in Problem 4(b) into each of the forms given in 4(c), 4(d), 4(e), 4(f)?

6. Rework Problem 4 parts (g), (h), (i), where the given command is replaced by SRD 31.

7. The following data are on a card:

WORD 1	IDENTIFYING MAN #	000000mmmm	10--0
WORD 2	HOURS WORKED	000000hhhh	8--2
WORD 3	HOURLY RATE	000000rrrr	7--3

Block diagram and write a program which will compute the earnings and punch it, along with the input data, on an output card. Earnings should be rounded to the nearest cent.

8. In an inventory problem, the identifying part number P is stored in drum location 1851; the total cost T, as an 8—2 number < 10^5, is stored in drum location 1855; and the total quantity Q, as a 10—0 number < 10^6, is stored in drum location 1856. All extra zeros are leading zeros. Compute the unit price U = T/Q rounded to the nearest mill, and punch out a card containing

WORD 1	WORD 2	WORD 3	WORD 4	WORD 5	WORD 6	WORD 7	WORD 8
P	U	BLANK	BLANK	T	Q	BLANK	BLANK

Be very careful of your scaling. When you have finished the problem, follow your directions *exactly* using a desk caclulator.

9. Design a program which will compute a numerical grade based on 100 as perfect from the following data: 17 short-quiz grades Q_i where $0 \leq Q_i \leq 10$, four 1-hour examinations E_i where $0 \leq E_i \leq 100$, and one final examination F with $0 \leq F \leq 100$, if the final examination is to be weighted twice as much as an hour examination, and the average of the 17 short quizzes is to be weighted as much as an hour examination.

10. Design a program which will compute a student's grade-point average to date using the following available data: the number of grade points earned and the number of credit hours taken during the last semester; the total number of grade points earned and the total number of credit hours taken during all semesters *prior* to the last semester.

11. Bring to class a problem of your own interest and at least block diagram a possible program for it.

12. Discuss the results of Problem 11 with a classmate.

3-11. OPTIMUM PROGRAMMING.

You should pay serious attention to the relative merits of various drum locations in positioning instructions and constants. If instructions are placed in *consecutive* drum locations—say locations 0201, 0202, 0203, etc.—then by the time instruction in location 0201 is executed, the drum has already passed drum location 202 and must wait a full revolution (or perhaps even two revolutions, depending upon the data address used) before it can pick up the next instruction. Ideally, if the instruction is located in drum location 201, the data address of that instruction should be just far enough around the drum so that its drum location will come under the read head just as needed, and the address of the next instruction should be just enough further around so that it will be under the read heads at exactly the time it is called for by the program register. Since the execution time for different instructions varies, it is necessary to keep track of them with care in order to secure an optimum program. This will be discussed later, but a good rule to follow at this stage is the 5–10 rule, which states that whenever possible the data address should be 5 greater (modulo 50) than the address in which the instruction is stored, and the next instruction address should be 5 greater than the data address just used. In the case of a shift operation it is usual to make the next instruction address 10 greater than the location in which the instruction is stored. This does not give an optimum program, but it will decrease the running time of most programs. Often programs set up using this rule will take only 1/5 or even 1/10 of the time required by sequential programming, unless, of course, the program requires a great deal of punching or reading.

A glance at Figure 1–3 or at a drum layout chart discloses that drum locations 0201, 0251, 0301, and in fact (0001 + 50k) for any integer k = 0, 1, 2, . . . , 39, are located the same distance around the drum and hence are accessible at the same instant. In fact, for any drum

location N, the 40 drum locations contained in the set (N ± 50k) are all accessible at the same time.

Problem Set 3-11

1. Each of the following programs adds 0000000001's to the upper half of the accumulator from the drum. Put each on the drum and let it run for 1 minute. Examine the totals in the upper accumulator at the end of each run.

```
(a) 0200   RAU 60   0202   0201
    0201   AUP 10   0202   0201
           CONSTANTS
    0202   0000000001
```

```
(b) 0500   RAU 60   0503   0507
    0507   AUP 10   0510   0515

    0515   AUP 10   0518   0523
    0523   AUP 10   0526   0531
    0531   AUP 10   0534   0539

    0539   AUP 10   0542   0547
    0547   AUP 10   0501   0507

    CONSTANTS 0000000001 IN EACH
    OF LOCATIONS 0503, 0510, 0518;
    0526, 0534, 0542, AND 0501.
```

3-12. INDEX REGISTERS (OPTIONAL).

An index register (sometimes called B-box or Z-box) is a small accumulator (4 digits and a sign). The 650 has available (on special order) three such index registers, known as index registers A, B, and C. They may be used as ordinary accumulators, but are more frequently used as count boxes, or in modifying addresses. Special circuitry permits the use of the index register for direct modification of addresses. If, for example, the instructions

SUP 11 (1001) 0540

were replaced by

SUP 11 (3001) 0540

then the actual command executed by the 650 will be

SUP 11 (1001 + CONTENTS OF INDEX REGISTER A) 0540;

that is, if either an instruction address or a data address is increased by 2000, then the 650 will perform as if, instead of adding 2000 to the address, the contents of index register A had been added to it.

INDEX REGISTER A	COMMAND RECEIVED	COMMAND EXECUTED
0012+	SUP 11 3001 0540	11 1013 0540

Only one command is required to change (step) the contents up on the index register. For example:

	OP	DA	IA
(ADD TO INDEX REGISTER A)	AXA 50	0002	----

will add the number 0002 to the contents of index register A. The command

(SUBTRACT FROM INDEX REGISTER A)	SXA 51	0001	----

will cause the number (0001) given in the data address to be subtracted from index register A. Note that the numbers added or subtracted are *not* the numbers in drum locations 0002 and 0001, but the actual number given in the data address (providing this number lies between 0000 and 1999 inclusive).

The command

	OP	DA	IA
(RESET AND ADD TO INDEX REGISTER A)	RAA 80	1437	----

resets index register A and adds the constant 1437 to it.

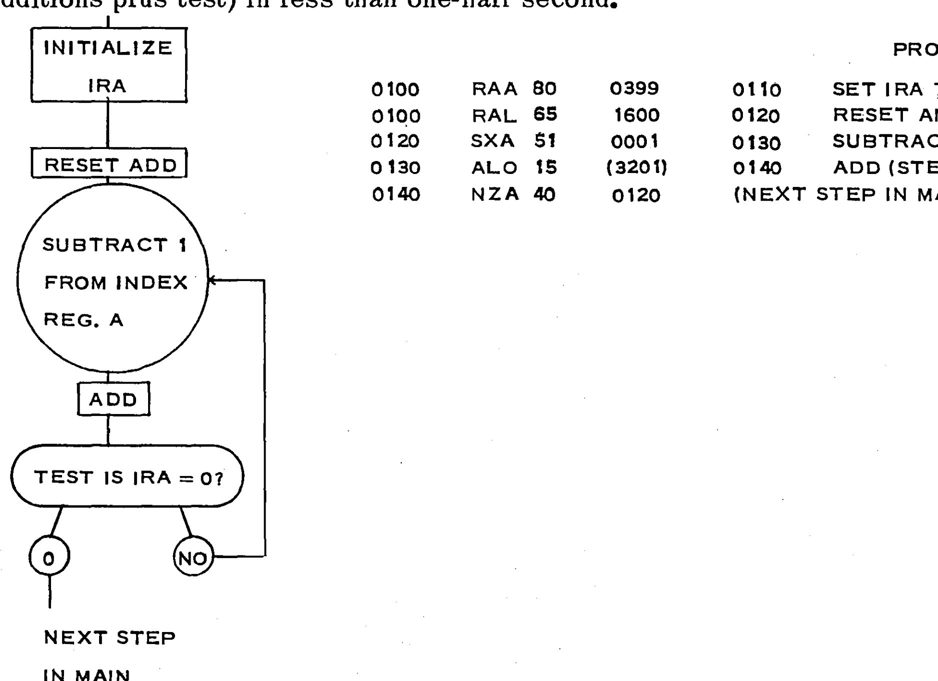
Similarly (RESET AND SUBTRACT FROM INDEX REGISTER A) RSA 81 resets index register A to zero and subtracts the four-digit number (< 1999) from it.

There exist Branch operations for index registers just as for any accumulator.

	OP	DA	IA
(BRANCH ON NON ZERO IN INDEX REGISTER A)	NZA 40	(GO HERE IF IRA IS NONZERO)	(GO HERE IF IRA IS ZERO)
(BRANCH ON MINUS IN INDEX REGISTER A)	BMA 41	(GO HERE IF IRA IS −)	(GO HERE IF IRA IS NOT MINUS)

There are similar operation codes for index registers B and C. Consult back cover for numeric codes.

Let 400 numbers be stored in drum locations 1201 to 1600. A simple program forms the sum of these 400 numbers and then passes to the next step in the main program as index register A indicates that 400 passes have been made. The addition is done by starting with drum location 1600 followed by 1599 = 1201 + 0398, etc., until the number in drum location 1201 = 1201 + 0000 is finally added in when index register A shows 0000, where the next step in the main program is completed. The 650 will complete the program (four hundred 10-digit additions plus test) in less than one-half second.



PROGRAM

0100	RAA 80	0399	0110	SET IRA TO 400
0100	RAL 65	1600	0120	RESET AND ADD INTO UPPER
0120	SXA 51	0001	0130	SUBTRACT 0001 FROM IRA
0130	ALO 15	(3201)	0140	ADD (STEPPED)
0140	NZA 40	0120	(NEXT STEP IN MAIN PROGRAM)	

The index register is of such great utility that it should certainly be used if available. Consult the IBM publication on index registers for additional information.

Problem Set 3-12

1. As a speed demonstration, the following program is prepared. Explain what it does and how many operations are performed on the 650 between the punching of each card. (The cards will not contain any desirable output data, they merely serve as a timing device.) Make a "guesstimate" of how long will elapse between punching the second and third output cards. Place the program on the 650 and run it to see how long it really does take between cards.

```
0201 RAA 80 1999 0208
0208 RAU 60 0202 0203
0202 SXA 51 0001 8002
0203 ALO 15 0205 8003
0205 NZA 40 8003 0207
0207 PCH 71 0000 0201
```

2. (a) See if you can arrange to load and run the entire program given above on one input card.

(b) Can you rewrite the program so as to use the standard 70 1951 9000 console setting?

3. Rewrite some of the programs of this chapter using index registers.

4. Use index registers to obtain the sum of the first 1000 terms of the (divergent) series $\sum_{i=1}^{\infty} \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \ldots .$

5. Use index registers to write a program to form the "sum of products" $Z = \sum_{i=}^{600} X_i Y_i$ where the one output card, containing the sum Z (20 digits), is punched into words 1 and 2. Assume the X_i are in drum locations 0601–1200 and the Y_i in 0001–0600.

6. Devise and code a routine which will read two integers A and B (with A < B) from words one and two of a card, compute the greatest common divisor of A and B, G = (A, B), and which will punch A, B, and G on an output card. Hint: Consult any text on Modern Abstract Algebra or on Number Theory under "Euclid's Algorithm" for a method of computing the greatest common divisor of two integers—for example, see p. 26 of *Selections from Modern Abstract Algebra* (Holt, 1958).

7. (a) Write a program which will take a 0-10 number X = .xxxxxxxxxx which is in the upper half of the accumulator and form X^{129} (accurate to not more than 10 decimal places, probably less), placing X^{129} in the upper accumulator when it is computed, and going to drum location 1327 for the next command. In drum location 1327 place the command 01 0000 1327 for the present.

(b) Note how many fewer steps the computer will take to compute $X^{129} = ((((((X^2)^2)^2)^2)^2)^2)^2 \cdot X$ rather than as $X^{129} = X \cdot X \cdot X \cdot \ldots \cdot X$.

CHAPTER 4

EASY PROGRAMMING VIA SUBROUTINES

4-1. ACCURACY OF COMPUTED RESULTS.

One of the most serious problems in any computation is that of determining the accuracy of the computed result. This problem is not confined to the 650—the same problem exists on a desk calculator or in hand computation. Since almost all scientific and/or engineering computations involve approximate numbers, the difficult subject of error analysis looms large on the scientific horizon. When few people did calculations, error analysis was usually ignored in college courses, except for a few crude (and usually incorrect) remarks concerning significant digits in multiplication and division, and decimal places in addition and subtraction. Now that large-scale computers are available to all, error analysis becomes a *must.*

The subject is strange (hence difficult) for two reasons:

1. No even reasonably simple rules on the accuracy of computed results have yet been devised.

2. Error analysis shakes our faith in the fundamentals of arithmetic, since certain basic algebraic rules such as $(a+b)c = ac + bc$ are *not valid* when approximate computation is used. If the two members $(a+b)c$ and $ac + bc$ are each computed, different results may be obtained!

You have heard the old "rule" that in addition and subtraction the result is accurate to one fewer decimal place than the least accurate data, and that in multiplication and division the result has one less significant digit than the least accurate piece of data. It has been known for a long time that *these rules are not valid,* but we point out two simple examples here to make this fact more vivid.

EXAMPLE 1:

Find $\sum_{i=1}^{1200} a_i$ where a_i = x.xx, accurate to the second decimal place—i.e., (true a_i) = x.xx ± .005. If most of 1200 pieces of data happened to be .005 too small, then the correct answer would be about $\sum_{i=1}^{1200} a_i + \sum_{i=1}^{1200} .005 = \sum_{i=1}^{1200} a_i + 6.00$. In other words, not only would the result not be accurate to the nearest 1/10, as the rule states, but even the units digit, and

possibly even the 10's digits, would be incorrect. The error here is quite apt to be greater than many of the individual pieces of data. You may think it unlikely that the majority of the errors will be of the same type, but can you be sure? Before you trust a result, you need some estimate of how accurate it is likely to be.

EXAMPLE 2:

An even more striking example is found in computing T = (A + B) · C. If A, B, C are each accurate to 10 digits, how much accuracy will T have?

Let A = + 12345.67899 with a possible error of ± .000005
B = − 12345.67898 with a possible error of ± .000005
C = + 77777777.33 accurate

Then (A + B) · C = (00000.00001) · (77777777.33),
= 777.7777733 computed answer.

ERROR ANALYSIS POSSIBLE "TRUE VALUES" OF	A	B	(A+B)	T = (A+B) · C
SMALLEST "TRUE T"	+ 12345.678985	− 12345.678985	ZERO	ZERO
LARGEST "TRUE T"	12345.678995	− 12345.678975	00000.000020	1555.5555466

In other words, although computation gives the answer 777.7777733, the correct answer may lie anywhere between 0 and more than 1555. Do you begin to see why some sort of error analysis is vital?

It is improbable that an answer of 777.8 will give satisfactory results if the correct answer is 0.52 or 1471.8. A more striking, but less typical, example may be obtained by using A = 9999999999, B = −9999999998, C = 9999999999, each with possible error of ± 0.5. Some indication of the possible variation in the computed result is essential, but not easy to obtain. All this has been known for years, but it is only recently, since large-scale computation has become widespread, that it is making itself felt among nonprofessionals.

It would be imprudent to attempt to give a course in numerical analysis as part of these notes, but it does seem essential that the vital need for error analysis at least be mentioned.

One colleague suggested including an outline of courses which would be appropriate. This sounds like an excellent idea until one tries to do so. The only conclusion is that those courses which are appropriate depend largely upon the use you plan to make of the 650. A person who planned to use the 650 only for inventory control, for example, really needs *no* college mathematics. A student of psychology, on the other hand, needs quite a bit, as does any other scientist or an engineer. The study of modern abstract algebra has come to the fore so much in all science and engineering during the last ten years that it probably needs no further mention except to say that modern abstract algebra is the heart not only of approximate computation, but also of the electrical circuitry involved in the computer itself (Boolean algebra)*. Computation which eventually resolves itself into a finite iteration of an infinite series requires a knowledge of Taylor's series *with remainder* and similar more advanced estimates of truncation error.

"Numerical analysis," like "statistics," means different things to different people. It is to be hoped that numerical analysis will not become the varied hodgepodge of courses which statistics presents today, where a single university may offer "statistics" courses in half a

*More detailed applications of Boolean Algebra to circuit theory will be found in the author's *Selections from Modern Abstract Algebra* (Holt, 1958).

dozen different departments. The variation which already exists between universities is startling. One midwestern university offers numerical analysis with no prerequisite other than calculus; another midwestern university has as its minimum prerequisite "two courses in real variable, two courses in modern algebra, and two courses in point-set topology." Some universities also require advanced probability and statistics as prerequisites for numerical analysis. It is difficult to say what courses constitute a reasonable program for a research scientist or engineer who wishes to use a computer. Certainly courses in advanced calculus, modern abstract algebra, and numerical methods are a portion of the bare minimum.

The reader now has a sufficient command of 650 programming to attempt programs of his own devising. Not all the possible 650 operations have been studied, but this need not deter you. The remaining operations, including 84 TLU (*Table Look Up*) and 90–99 BD(k) (*Branch on 9 in Distributor*) as well as work on 36 SCT (*Shift and Count*), are included the IBM Programmer's Manual. Many beginning programs can be programmed without their use, just as you can program without understanding board wiring. If you are to do serious programming, you will eventually want these tools, as well as more knowledge of floating-point numbers, and index registers.

4-2. SUBROUTINES.

Perhaps the secret of becoming a capable (if not expert) programmer in a short time is to write only the simple parts of programs. This can be done either by letting the 650 more or less write its own program (see Chapters 5, 7, 8) or by using *subroutines* which someone else has already written. It is the latter alternative which will be discussed next. Before considering a partial list of some of the hundreds of subroutines now available, let us examine the directions for using two specific subroutines. The subroutines themselves consist of a pack of IBM cards which are available in your computer library.

1. *Square Root-0070* (esentially SR2 of *Technical Newsletter No. 9*). Computes $\sqrt{A}$ for A in the range $0 \leqq A = .xxxxxxxxxx \leqq .9999999999$. Uses location 0070 and 22 locations in range 0158–0197.

Linkage: Put A into upper accumulator.
Load distributor with next instruction to be performed after $\sqrt{A}$ has been computed, and transfer control to 0070—i.e., LD 69 (address next instruction) 0070. $\sqrt{A}$ will be placed in upper accumulator after it is computed, and the program will continue with the instruction whose location was specified in the data address of 69 (next) 0070.

$$\sqrt{A} = .xxxxxxxxxx$$

Both A and $\sqrt{A}$ are 0–10 numbers.

2. e^X *E-to-the-X-0073* (essentially Exponential Subroutine of *Technical Newsletter No. 9*).
Computes e^X for X in the range $-1 < X = .xxxxxxxxxx < 1$. Uses locations *0073*, and *270* to *295*.

Linkage: Put X in upper accumulator.
Load distributor with next instruction to be performed after e^X has been computed.
Transfer control to 0073—i.e., LD 69 (next) 0073.
The pack of IBM cards in the subroutine takes care of computing e^X.
e^X will be placed in upper accumulator after it is computed.

$$e^X = x.xxxxxxxx$$

X is a 0-10 number and e^X is a 1–9 number.

EXAMPLE 1:
To compute $Y = e^{\sqrt{A}}$ for a set of 200 values of A, where the A's are positive 6-digit decimals of the form .xxxxxx0000, which are found in word 5 of a set of data cards, we may proceed, using these two subroutines, as follows. We shall read into 0401-0410 and punch from 0427-0434.

LOCATION OF INSTRUCTION	OP	DA	IA	COMMENTS
0411	RCD 70	0401	0412	READ IN A
0412	RAU 60	0405	0413	PUT A INTO UPPER
0413	STD 24	0427	0414	STORE A IN PUNCH BAND
0414	LDD 69	0415	0070	TO SQUARE ROOT SUBROUTINE
0415	LDD 69	0416	0073	TO E TO THE X SUBROUTINE
0416	STU 21	0430	0417	STORE RESULT, $E^{\sqrt{A}}$
0417	PCH 71	0427	0411	PUNCH

By loading this 7-step program plus the two subroutine decks into the 650, a program which computes $y = e^{\sqrt{A}}$ results.

EXAMPLE 2:
For given values of K when $0 < K < 10$ obtained from word 1 of input data cards, solve $X \sin X = K$ for some one positive solution X_s, and print out K and X_s on output cards. In essence, we ask, "Where does the curve $Y = X \sin X$ cross the line $Y = K$?"

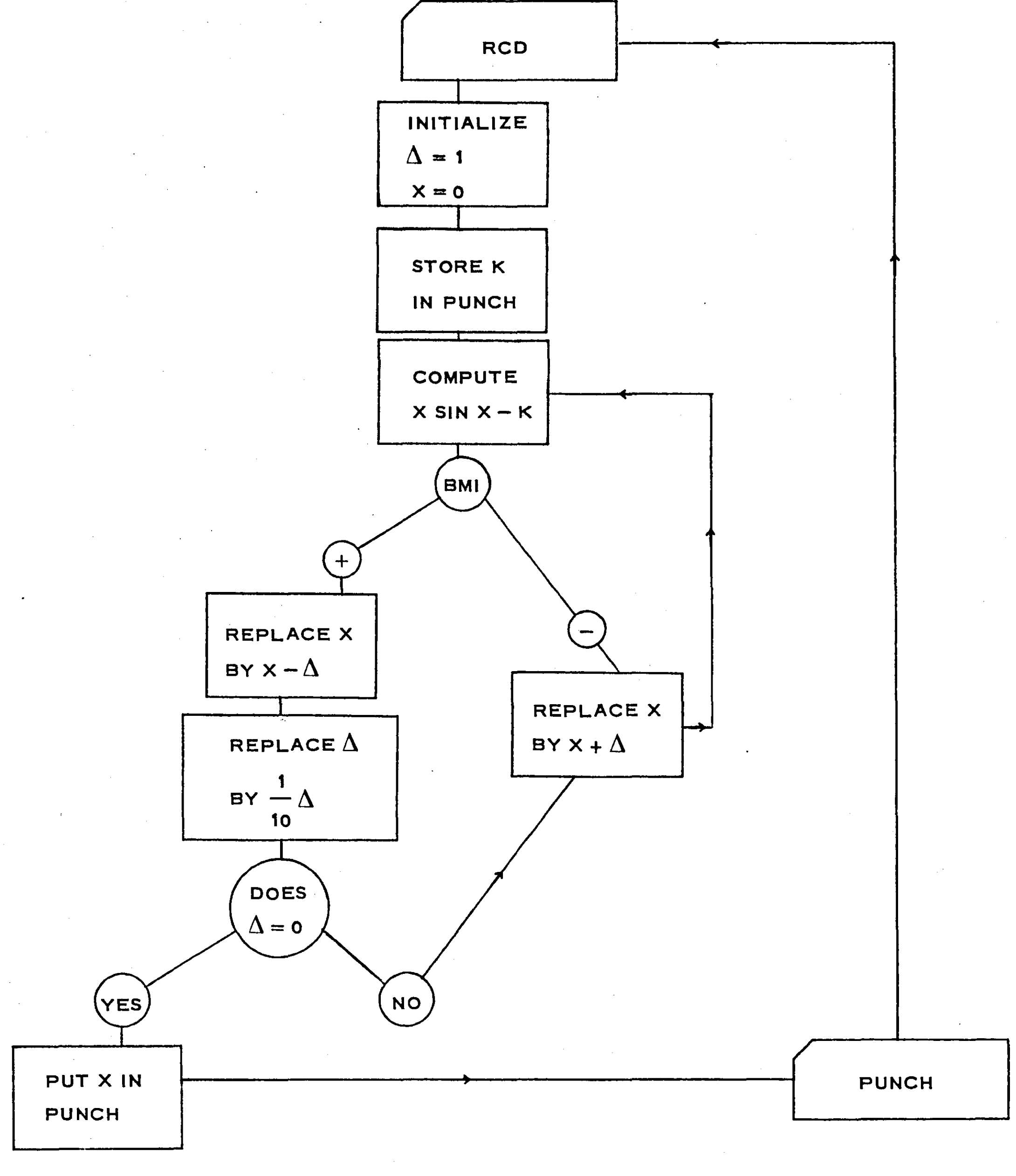
We shall find only some one positive (first) intersection, but others can be found by slight program modification. The technique used is one which would be crude for hand computation, but is well adapted to machine computation. Set $f(X) = X \sin X - K$. Note that $f(0) = -K$ is negative. We shall compute f(0), f(1), f(2), ... until we find a value of X, say X = 8, such that f(8) is positive. When this happens, we have passed the desired X_s value, so we back up one step, to X = 7, and compute f(7.1), f(7.2), ... until f(X) again becomes positive. We then back up one step (this time a .1 step, not a 1 step) and proceed taking steps only .01 long, etc., until X_s is obtained accurate to 9 or 10 significant digits, whereupon K, X_s are punched on an output card and the process repeated with a new value of K. We assume that a subroutine for computing sin X for $0 \leqq X \leqq 5$ is available which takes x.xxxxxxxxx from the upper as a 1-9 number and computes sin X when the command 69 (next) 0071 is given, where (next) is the address of the drum location containing the next instruction to be performed after sin X is computed and placed in the upper. Sin X will be a 1—9 number in the upper when the subroutine is completed.
A program could be written as:

LOCATION OF INSTRUCTION	OP	DA	IA	
0500	RCD 70	0501	0551	
0551	LDD 69	0554	0557	
0557	STD 24	0560	0513	△ IS STORED IN 0560
0513	LDD 69	0516	0519	
0519	STD 24	0522	0525	X IS STORED IN 0522
0525	LDD 69	0501	0604	
0604	STD 24	0527	0580	
0580	RAU 60	0522	0577	
0577	LDD 69	0630	0071	
0630	MPY 19	0522	0542	
0542	SUP 11	0501	0555	
0555	BMI 46	0558	0559	
0558	RAU 60	0522	0627	
0627	AUP 10	0560	0515	

(Continued on next page)

LOCATION OF INSTRUCTION		OP	DA	IA
0515		STU 21	0522	0577
0559		RAU 60	0522	0677
0677		SUP 11	0560	0565
0565		STU 21	0522	0575
0575		RAU 60	0560	0615
0615		MPY 19	0518	0538
0538		STU 21	0560	0563
0563		NZU 44	0558	0568
0568		LDD 69	0522	0625
0625		STD 24	0528	0581
0581		PCH 71	0527	0500
0554	CONSTANTS	ONE 10	0000	0000
0516	CONSTANTS	ZERO 00	0000	0000
0518	CONSTANTS	ONE-TENTH 10	0000	0000

A flow chart would be:



Note that it has been necessary to pay some attention to the scaling problem: X is a 1—9 number, hence to form X-Δ we need Δ as a 1—9 number. We wish Δ/10 to be a 1—19 number so it truncates to a 1—9 number. Hence we take 1/10 as a 0—10 number so that the product (1/10) (Δ) is a 1—19 number. It happens that 1/10 as a 0—10 number has the same form as the number 1 has as a 1—9 number, but both numbers have been kept to make it easier to understand. In actual practice only one need be used. Since X and sin X are both 1—9 numbers, X sin X is a 2—18 number, and if we are to subtract K without shifting, we must put K in as a 2—8 number.

Since there are many expert 650 programmers working out subroutines, more of these routines are being added all the time. A partial list of available subroutines and full programs is included below to give an indication of how broad the collection is. With the help of these subroutines and a knowledge of basic programs, *you* can write programs to take care of complicated problems. The only real difficulty remaining in many problems is that of proper scaling. This is often avoided by the use of floating-point arithmetic.

4-3. PARTIAL LIST OF AVAILABLE PROGRAMS AND SUBROUTINES.

Matrix-vector Multiplication
Matrix Addition
Matrix Inversion
Matrix Inversion by Gaussian Elimination
Large-scale Matrix Multiplication
Double Precision Matrix Inversion
Complex and Real Eigenvalues
Small-scale Matrix Multiplication
Multiple Regression Analysis
Simple Correlation Coefficients
Correlation Coefficient Routine
Analysis of Variance Program
Auto-correlation Program
Polynomial of Best Fit by Least Squares Method
Multiple Correlation for 50 Variables
Unbiased Standard Error of the Regression Coefficients
Weighted Least Square Polynomial Approximation
Real and Complex Roots of Algebraic Equations
Roots of a Function of a Real Variable
Optical Ray Tracing
Neutron Diffusion
Critical Reactor Assembly
One-space-dimensional Multigroup
Lost, A Cross-section Averaging Program
Survey Traverse
Cut and Fill Program
Survey Traverse with Balancing
Calculation of Piping System Expansion Stresses
Square Root
Cube Root
Exponential
Exp A
Sinh A, Cosh A, Exp A
Sinh X and Cosh X
Sine or Cosine
Arc Sin A
$\text{Log}_{10}A$, $\text{Ln}_e A$
Natural Logarithm
Polar to Cartesian Coordinates
Square Root $|X|$
Arctan X and Ln $|X|$
Sum and Sum of Squares
Circular and Hyperbolic Functions
Regular Bessel Functions
Irregular Bessel Functions
Simultaneous First Order Differential Equations
Multiple Numerical Integration
Fourier Synthesis
LaPlace Transformation
Systems of First Order Ordinary Differential Equations
Solution of Simultaneous Linear Equations
Complex Arithmetic Matrix Inversion
Well Bore Deviation Record
Linear Programming
Transportation Problem
Numerical Integration (Runge-Kutta or Milne)
Chi Square
t-test
Numerous special traces and debugging programs.

The subroutine library of your own installation will list the subroutines available, and will undoubtedly furnish you with "entry data summaries" similar to those given for $\sqrt{A}$ = square root, and e^X = E to the X, found in Section 4-2.

A few words of warning concerning subroutines are pertinent.

1. It is usual to have the sign of each instruction in a subroutine *negative,* since this permits the use of a punch-trace routine which will trace positive instructions at a speed of 100 instructions per minute (since a card is punched for each instruction, giving the instruction, the contents of both the accumulators and of the distributor, and other data), but which will trace negative instructions (known to be correct) at what is almost normal machine speed without punching cards.

2. In general it is possible to relocate subroutines if this should be necessary.

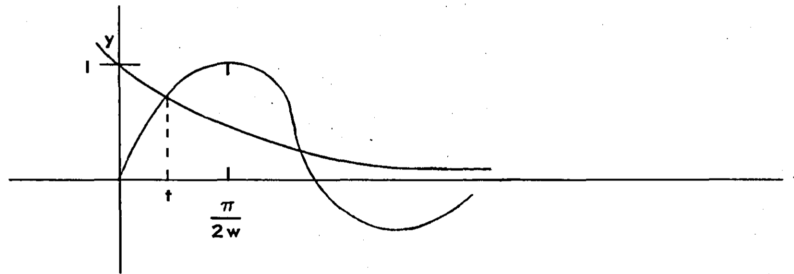
3. It is always necessary to know what locations the subroutine uses, even though you may never know how it works, since the same drum location must not be used for two instructions.

4. It is important, in using a subroutine or a program that someone else has set up, that you ascertain the range of permissible input values and the form of the input and output values. For example, if the input X on the e^X subroutine were a form of a 4-6 number 0000.xxxxxx instead of a 0-10 number .xxxxxxxxxx, it would have been necessary to shift left 4 before going into the subroutine to secure the desired answer, even though it was known that the first four digits were all zeros (i.e., $-1 < X < 1$).

4-4. A PROBLEM.

The following is an actual problem which occurs in the planetary gear systems currently used in automatic transmissions on automobiles. The search for smooth automatic shifting using planetary gears rather than the inefficient double turbine system is still (1958) in progress. The problem of Example 1 below is solved repeatedly for various values of w and a. It is quicker to compute the answer anew each time it occurs, using infinite series, than to look it up in a table, even if such a table were available.

Example 1:
The curves $y = \sin wt$ and $y = e^{-at}$ are plotted on the same set of y, t-axes. Determine the smallest positive value of t at which they intersect.



Before continuing, the reader should decide how he would solve this problem without using the computer, and should actually obtain one solution accurate to 5 decimal places for a specific pair of value of a and w, say $-a = -.625$ and $w = .475$. It would, of course, be ideal if the pair of equations could be solved in general, obtaining some neat formula involving a and w for the intersection t, much as the quadratic formula $x = \dfrac{-B \pm \sqrt{B^2 - 4AC}}{2A}$ is a general

solution of $Ax^2 + Bx + C = 0$. Persons who are skilled in transcendental analysis may wish to attempt this, but it is neither easy nor necessary.

The problem reduces to finding the smallest positive root of the equation

$$f(t) = e^{-at} - \sin wt = 0$$

for fixed values of a, w. At $t = 0$, $f(0) = e^{-at} - \sin wt = 1 - 0 > 0$, while at $t = \frac{\pi}{2w}$,

$$f(t) = e^{-at} - \sin wt < 0.$$

Somewhere between 0 and $\frac{\pi}{2w}$ is a value of t which makes $f(t) = 0$, since f(t) is continuous in the region $0 \leqq t \leqq \frac{\pi}{2w}$. (Would this be true if f(t) were not continuous?) How can you be sure that f(t) is continuous on the given region? What does "continuous over a region" mean? Name a function which is *not* continuous for the region $0 \leqq t \leqq 1$.

One possible technique is to compute $f(t_i + \Delta t)$ with $\Delta t = 1$ and $t_i = t_{i-1} + \Delta t$ starting with $t_0 = 0$ i.e., f(0), f(1), f(2), ... —until a value $t_i + \Delta t$ is found such that $f(t_i + \Delta t) < 0$. Assume that this happens at $t_i + \Delta t = 3$. We then back up one step to $t_i = 2$ and multiply the old Δt by 1/10 and continue obtaining, say, f(2.1), f(2.2), ... , f(2.8) before a change in sign in f is found at $t_i + \Delta t = 2.8$. We again back up one Δt step to $t_i = 2.7$, multiply Δt by 1/10, and continue. This yields f(2.72), ..., etc. Since the machine computation of f(t) is rapid, this method will be quite satisfactory.

If $0 < w < .1$, it may be desirable to take the original Δt as 10 or even 100 rather than 1. (Why?)

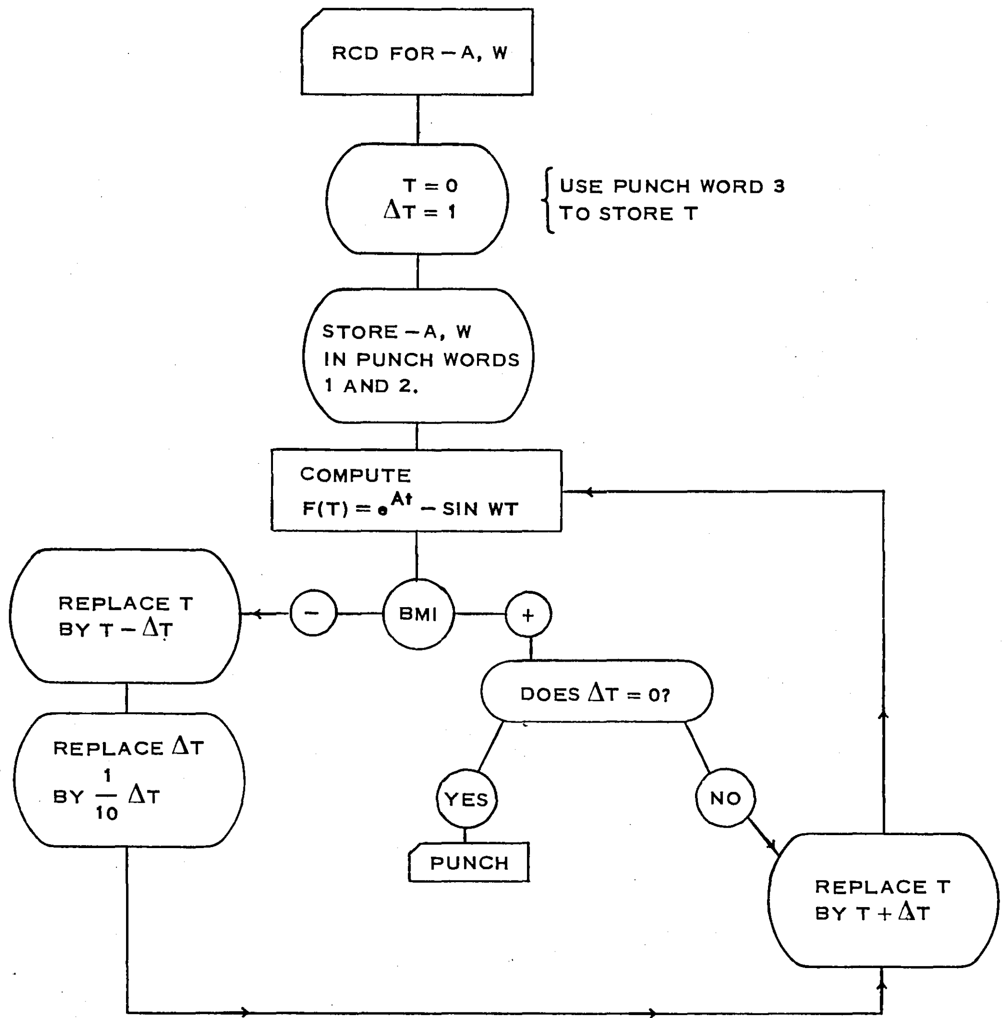
A possible flow chart for this problem appears on the next page.

4-5. A WORD TO THOSE WHO WILL HAVE OTHERS WRITE THEIR PROGRAMS.

If $w > .1$, then the desired solution $t < \frac{\pi}{2(.1)} < 16$ for all values of $-a < 0$. If $0 < w < .1$, it may be desirable to take the original Δt as 10 or even 100 rather than 1. Thus, the more the person who writes the program knows about the nature of the data, the easier it is for him to write a good program. Persons who depend upon others to write their programs often could save considerable money and time by giving the programmer more details concerning the data. In this case it makes a difference in programming whether you state $w > 0$, $w > .001$, or $w > .1$. In other problems the difference may be even more important. Certain inventory problems are impossible as 650 problems (without tape or Ramac) unless the data (issues and receipts) are sorted into two sets and the new receipts read into the 650 *before* the issues are read. Advance planning will always make this possible, and it is usually possible anyway, but the programmer must know this. The usual axiom of the professional programmer is "expect the worst and plan for it." If he plans for conditions which are much worse then actually will occur, it is costly not only in programming time but also in machine time.

You must also be careful in specifying what results you want. On a recent research problem in psychology, the running time on the problem was cut from 1 hour 23 minutes to just 7 minutes by cutting out the punching of certain unneeded intermediate results—a saving of over $100 each time the problem was run.

Moral: If you plan to have someone else do your programming for you, give him as complete details as possible—even the seemingly trivial details may be vital. Also, learn enough



about programming so that you know what output data could be obtained at no extra cost, and what output data would entail a considerable saving if omitted.

Give your programmer some leeway. Tell him (a) results you must have; (b) results you would like to have, if available at not more than $25 extra per run; and (c) results that would be handy if they could be obtained at no extra running time cost. (This is often possible.)

Problem Set 4–5

1. Solve $e^{-.625t} - \sin .475t = 0$ by hand methods, obtaining the answer correct to 5 decimal places. Use tables to speed your work, and keep track of your time, including time spent seeking numbers in the table. The 650 can perform the entire computation in less than 20 seconds.

2. Write a program which will read x and a from a card (the cards are not yet punched, so you may specify the form in which each is to be punched), compute

$$y = \frac{x + a - \sqrt{2x^2 + .001}}{7\,a^3}$$

and punch x, a, y in an output card and continue. Assume that $0 < x < .6$ and $1 < a < 3$ and that the square root subroutine 0070 described in Section 4–2 is available.

3. Write a program which will read a value of x (you decide what the permissible range and form of x shall be), from a card, and punch x, e^x, sinh x, cosh x on an output card. Assume the e^x subroutine of Section 4-2 is available.

4. Modify the program of Problem 3 so that $T = (1 - \cosh^2 + \sinh^2 x)$ will also be punched in word 8 of the output card. Of what value is the test function T? (*Hint*: If exact values of cosh x and sinh x were obtained, in place of decimal approximations, what constant value would T have?)

5. Your calculus tables will remind you that

$$\frac{d \text{ Arc cos } x}{dx} = \frac{-1}{\sqrt{1 - x^2}}$$

Devise a program which will read x from an input card and punch x and $\frac{d \text{ Arc cos } x}{dx}$ in an output card. Assume that $0 \leqq x \leqq .75$. Discuss your program with a classmate before running each on the 650.

6. It is possible to show that $\int_0^{\pi} \frac{dx}{a + b \cos x} = \frac{\pi}{\sqrt{a^2 - b^2}}$ if $a > b$. Devise a program which will compute the given integral for suitable values of a and b. Then write up your program indicating the limits and form of a and b which may be used with your program.

7. Design a program which will compute $\sqrt[1024]{x}$ by taking $\sqrt{\sqrt{\sqrt{\ldots \sqrt{x}}}}$. Be sure that you loop the correct number of times.

8. Obtain a list of the subroutines currently available in your 650 library and devise a program of your own interest, using some of these subroutines.

9. Examine one specific subroutine, a short one, and see how it meets the problem of getting back into the main program after the subroutine is completed.

10. Use the trapezoidal rule, with $t = 0.01$, to approximate the area under the curve $y = \frac{1}{\sqrt{2\pi}} e^{-t^2/2}$ between $t = 0$ and $t = X$ for values of X, read from word 7 of a set of input cards. The input X is of the form x.xx0000000 and $0 \leqq X \leqq 4$. This is, of course, the normal probability integral and is of great importance in scientific and engineering work as well as in the social sciences.

4-6. FLOATING-POINT ARITHMETIC.

Your most troublesome problem at present is probably scaling. The difficulties of scaling may have certain other (presumably lesser) difficulties substituted for them by using *floating-point arithmetic*. Recall the "scientific notation" for writing numbers as $0.nnnn \times 10^k$. (See p. 142 of *Fundamentals of College Mathematics* by Brixey and Andree, Holt, 1954.)

DECIMAL NOTATION	SCIENTIFIC NOTATION	
−375.67899 =	−.37567899 × 10^3	→ −37567899 (+3)
.0000123456 =	.123456 × 10^{-4}	→ 123456 (−4)

There is, actually, no need to write the 10, since the last expression on each line is clearly derived directly from the scientific notation and may be so interpreted. Since both the number and the exponents have signs, we must either store two signs (inconvenient) or devise a new

method of expression. The new method of expression used is simply to add 50 to the exponent, giving

DECIMAL	SCIENTIFIC		FLOATING POINT
−375.67899	= −37567899 × 10^3	⟶	(−)3756789953
.0000123456	= .123456 × 10^{-4}	⟶	(+)1234560046

Some systems place the (exponent plus 50) in the two right-hand digits (low order) while others use the two left-hand (high order) positions. Currently most of the IBM 650 floating-point routines favor the low-order position, and this notation will be used here even though the other has certain advantages.

Floating-point notation allows one to write any number N whose absolute value lies in the range $10^{-50} \leqq |N| < 10^{50}$ with an accuracy of eight significant digits. This range has been found quite adequate for most problems, providing the number zero can also be represented. Zero is represented in floating point by 00000000 00—i.e., the "exponent plus 50" is always taken as 00, along with the significant digit zeros.

SAMPLES:	DECIMAL	SCIENTIFIC		FLOATING POINT
	+ 3333333.3	= +.33333333 × 10^7	⟶	+3333333357
	− 444.44444	= −.44444444 × 10^3	⟶	−4444444453
	+.0000777777	= +.7777777 × 10^{-4}	⟶	+7777777746

This system appears to be the true answer to the scaling problem, particularly since both subroutines and hardware have been devised which perform the floating-point operations. However, there are two serious drawbacks—namely, accuracy and execution times. We discuss each briefly.

It may, at first, seem that an accuracy of eight significant digits is adequate for all normal problems. True accuracy of eight significant digits probably would be. However, consider the following type of problem:

$$(22222.222 - 22222.221) \cdot (5555555.5).$$

If the subtraction is performed by hand, one obtains

$$(.001)\,(5555555.5) = 5555.5555.$$

Since (.001) is accurate to only one significant digit, it is immediately apparent that the eight 5's certainly do not represent eight significant digits. In fact, since

$$(.001 \pm .0005) \cdot (5555555.5) = (5555.5555 \pm 2777.77775),$$

the correct answer may lie anywhere between 2778 and 8333 rather than near 5555. The same difficulty arises in fixed-point arithmetic (either hand or machine), but the programmer is more apt to be aware of the difficulty, since he has considered the scaling of numbers rather carefully. It has long been recognized that the old rules concerning the reliability of computed results being "a little less accurate than the least accurate piece of data" were unfounded, but use of the desk calculator with all intermediate results visible permitted the substitution of the (also invalid) rule "a little less accurate than the least accurate number that appears in the computation." High-speed stored-program computers have made it necessary to re-examine the entire subject of accuracy. Much new theory has been developed within the last five years, and no one who plans to work with a large-scale computer can afford to be without a modern, up-to-date course in numerical analysis!

An appalling example of the drastic effect of "slight round-off error" is obtained by expanding the polynomial

$$f(x) = (x-1)(x-2)(x-3)(x-4)\cdots(x-19)(x-20)$$

whose roots are the real integers 1, 2, 3, · · · , 20. Since the coefficients are large, floating-point notation is used. When the resulting polynomial is then solved, roots are indeed approximately 1, 2, 3, and 20, but many of the remaining roots prove to be *complex numbers*, $a + bi$, with b not at all near zero! In other words, the round-off error has drastically changed the basic nature of the polynomial.

The second possible disadvantage of the floating-point operations is that if floating-point arithmetic is performed by subroutines, it is much slower than fixed-point arithmetic. If your 650 is equipped with an automatic floating-decimal arithmetic device (an auxilliary feature in the 653 unit), multiplication and division execution times will be about the same as for fixed-point operations, while floating-point addition and subtraction take about twice as long as the corresponding fixed-point operations. *However,* if your 650 is not equipped with automatic floating-point operations, then it is necessary to use floating-point subroutines, in which case the arithmetic operations take between 10 and 20 times as long to perform as fixed-point operations. Interestingly enough, the general trend in computers seems to be toward the use of floating-point arithmetic in scientific computation, in spite of the time and accuracy disadvantages, since it is so *much* easier to use. Some of the modern subroutines automatically check each operation to see if a loss of more than two significant digits has occurred. If it has, the program will stop on a program stop (OP code 01) until the operator again pushes the program start key. This is a highly desirable feature and it is hoped that it will eventually be available on the automatic floating-point operations.

All floating-point operations are done in the *upper* half of the accumulator. *The lower and the distributor may be disturbed in the process,* but the final result will be in the upper. The operation codes are

FAD 32	FLOATING ADD
FSB 33	FLOATING SUBTRACT
FDV 34	FLOATING DIVIDE
FAM 37	FLOATING ADD MAGNITUDE
FSM 38	FLOATING SUBTRACT MAGNITUDE
FMP 39	FLOATING MULTIPLY
UFA 02	UNNORMALIZED FLOATING ADD. (THIS MAKES IT POSSIBLE TO ATTACH THE SAME EXPONENT TO A GROUP OF NUMBERS FOR FIXED-POINT OUTPUT, BY SUPRESSING THE NORMALIZATION WHICH OCCURS AFTER ADDING.)

Subroutines are available which use the floating point numbers. An example is the "Elementary Transcendental Function Package—Floating Point" which accepts floating point values of X from the upper accumulator and computes any of the following values, placing the result in the upper: $\tan^{-1}X$, cos X, sin X, e^X, $\sqrt{X}$, ln X, log X. The commands to be given are

```
RAU   X
LDD   NEXT ENTRY
```

when X is the address of the floating point number X, NEXT is the address of the next instruction to be performed after the subroutine is completed and ENTRY is the number specified below, which determines what function of X is computed:

FUNCTION	ENTRY
$\tan^{-1}X$	0050
cos X	0076
sin X	0176
e^X	0250
$\sqrt{X}$	0331
ln X	0439
log X	0457

This packet uses drum locations 0000 to 0459.

CHAPTER 5

SOAP

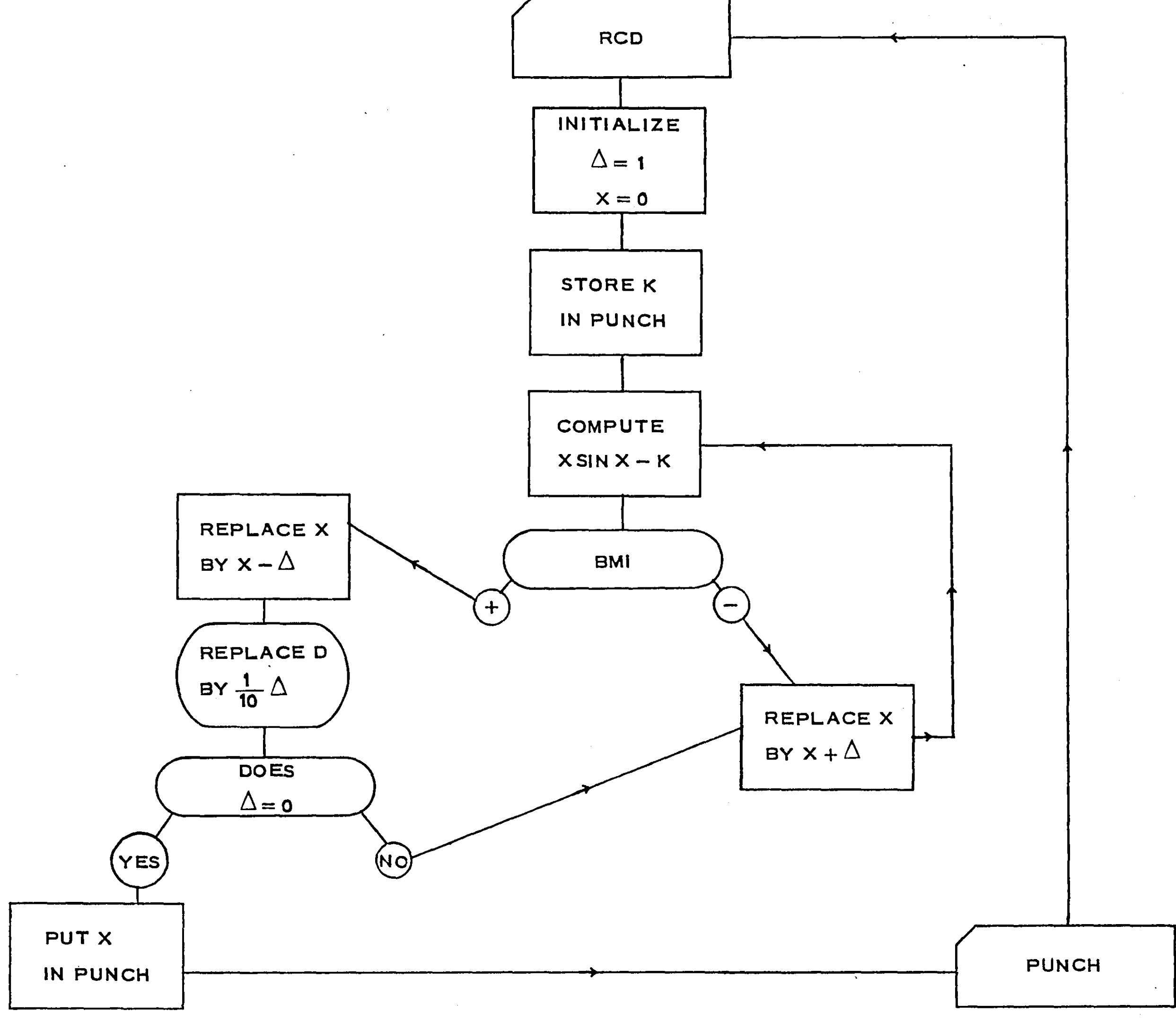
5-1. SOAP (SYMBOLIC OPTIMUM ASSEMBLY PROGRAM).

As was pointed out in Chapter 3, a program written serially, in consecutive drum locations, is about as poorly constructed as possible, since the drum must make at least one complete revolution between each program step. The 5-10* rule speeds up a program considerably, but still is not the best (optimum) program. To write an appreciably better program requires knowledge of exactly how long each operation takes. Tables giving this data are available in the IBM manual, but experience shows that the work is so tedious that most programmers are reluctant to spend the time necessary, and consequently fall back on some pet variant of the 5-10 rule. However, tedious arithmetic and memory work are exactly what the 650 is designed to do! Several programs exist which take a given unoptimized program and use the 650 to produce an optimized program. One of the best known of these programs is IBM's Symbolic Optimum Assembly Program, S.O.A.P. Not only does SOAP optimize a program, but it will do so from a program in which most of the addresses have been omitted, provided the steps are in order. Furthermore, SOAP utilizes the alphabetic device of the 650 so that the alphabetic mnemonic operation codes (RAL) may be used in place of the corresponding numeric (65) codes. Alphabetic names may also be used to designate storage locations if desired. Even with limited understanding of the 650 programming, you will find SOAP programming much easier than machine-language programming.

Special SOAP programming sheets and special SOAP cards are available. Three versions (I, II, and III) of SOAP are available, but the theory is the same in each case. SOAP II is used here. After loading the SOAP deck (your lab has one), it is possible to "process" (i.e., SOAP) as many programs as desired. The input (SOAP) cards consist of the unoptimized program written in acceptable SOAP symbols. The output cards will contain the optimized program expressed in machine language. Each output card also contains the program instruction in SOAP language from which it was obtained, for ease in debugging. The output cards contain the standard 650 loading code and the machine-language instructions in words 1 to 4, and are treated like the program (load) cards we have been using. Reload them, using the "standard board," and debug as usual. A special SOAP board is used in the 533 while the SOAP program is being optimized and converted into machine language. Let us now re-

*The 5-10 rule advises using a data address 5 greater than the location of the instruction and a next instruction address 5 greater than the data address whenever possible.

examine the program of Example 22, Section 4-2, in which we computed X_n such that $X \sin X = K$ for $0 < K < 10$. The flow chart follows:



Let us write the program in SOAP, first reserving drum location 0000 to 0499 and 1000 to 1999 for other purposes—i.e., the program can be placed in locations 0500 to 0999, but not elsewhere. This is done by placing Block Reserve

```
BLR    0000    0499
BLR    1000    1999
```

on the first two SOAP cards. We shall next reserve Regional Read (0501-0510) and Punch (0527-0534) areas thus:

```
REG    R0501    0510
REG    P0527    0534
```

Even though we only use one read location (0501) we reserve the entire block, since otherwise the SOAP program might place an instruction in one of these locations (say 0506) only to have it erased when a data card was later read in the program. We may now address drum

location 0501 either as 0501 or as R0001 if desired. Experience shows this is very convenient when programming. A sample SOAP program follows:

LOC	OP	DA	TAG	IA	TAG	COMMENTS
	BLR	1000		1999		
	BLR	0000		0499		
	REG	R 0501		0510		
	REG	P 0527		0534		
START	RCD	R 0001				READ CARD
	LDD	ONE				
	STD	DELTA				
	LDD	ZERO				
	STD	X				SET
	LDD	R 0001				
	STD	P 0001				
	RAU	X		LOOP		
LOOP	69			0071		TO SINE X
	MPY	X				
	SUP	R 0001				
	BMI	NEGI		POSI		
NEGI	RAU	X				
	AUP	DELTA				
	STU	X		LOOP		
POSI	RAU	X				
	SUP	DELTA				
	STU	X				
	RAU	DELTA				
	MPY	TENTH				
	STU	DELTA				
	NZU	NEGI		END		
END	LDD	X				
	STD	P 0002				
	PCH	P 0001		START		
ONE	10	0000		0000		
ZERO	00	0000		0000		
TENTH	10	0000		0000		

The 650 will translate and optimize this program giving the program as:

	OP	DA	IA
0500	RCD 70	0501	0551
0551	LDD 69	0554	0557
0557	STD 24	0560	0513
0513	LDD 69	0516	0519
0519	STD 24	0522	0525
0525	LDD 69	0501	0604
0604	STD 24	0527	0580
0580	RAU 60	0522	0577

	OP		DA	IA
0577	LDD	69	0630	0071
0630	MPY	19	0522	0542
0542	SUP	11	0501	0555
0555	BMI	46	0558	0559
0558	RAU	60	0522	0627
0627	AUP	10	0560	0515
0515	STU	21	0522	0577
0559	RAU	60	0522	0677
0677	SUP	11	0560	0565
0565	STU	21	0522	0575
0575	RAU	60	0560	0615
0615	MPY	19	0518	0538
0538	STU	21	0560	0563
0563	NZU	44	0558	0568
0568	LDD	69	0522	0625
0625	STD	24	0528	0581
0581	PCH	71	0527	0500
CONSTANTS 0554	1	10	0000	0000
CONSTANTS 0516	0	00	0000	0000
CONSTANTS 0518	1	10	0000	0000

This program has better optimization than most programmers would achieve in two hours time, but it is *not* the best that can be achieved by expert hand optimization. For most purposes SOAP is an excellent method of writing good programs with a minimum of effort, without using For Transit (see Chapter 8). There is more to be said about using SOAP, but the general rules are simple.

If you wish to use a machine-language numeric operation or address (drum location), leave the left-hand column (separated by a dotted line on the program sheet) *blank*. If you wish to use a *symbolic* (usually alphabetic) operation or address, be sure that the left column is *not* blank. Symbolic addresses may contain from one to five symbols. Typical examples are:

O	NE	S	TART
5		L	OOOP
3	RD	L	OOPZ
G	ROSS	M	ARY
D	ELTA	N	ET
D	X	X	
D	T	Y	
J	OSIE	E	ND
F	INIS	X	IX

Whenever a data address or an instruction address desired is the location of the next instruction, it may be left blank, provided the desired LOCATION column is also blank. Thus the address END used here is unnecessary, but harmless. The address NEG 1 and LOOP, however, are essential, since they are used later in the program as addresses to which the 650 is sent in a loop.

Any desired comments may be placed in the comments column. If more than 10 characters are needed for remarks, they should be placed on the next line of the program sheet with a 1 indicated in the type column (Col. 41). Several type 1 (i.e., 30 character comment) cards may be used in succession to identify different portions of a program, if desired. It is often desirable to begin with a type 1 (comment) card identifying the problem and the program. A "blank" type 1 card can be used to make the 402 (or 407) listing skip a space, thus providing a more readable output. (For example, put a blank type 1 card between your last block reserve and the beginning of your program.)

What you need now is practice—GO TO IT.

Obtain a copy of the SOAP II manual and refer to it as needed.

CHAPTER 6

A RE-EXAMINATION OF PRINCIPLES

6-1. SOPHISTICATION IN PROGRAMMING.

Experience provides the opportunity to become more sophisticated in the matter of programming. Genuine sophistication cannot be learned from books, but it may help to mention certain indications of sophistication which are usually observed in artful programming.

It must be remembered that elegant programming, like most creative work both in the sciences and outside of them, is more of an art than a science in the usual meaning of the words. As such, there must develop a "feeling for" programming rather than a set of rules. One way to develop this feeling rapidly is to work out your own solution to a specific problem and compare your solution with that of others. A set of situations designed to help improve your programming technique is presented at the end of this chapter.

First, a few comments gleaned from experience may serve as guideposts.

6-2. FLOW CHARTING.

Never program without a flow chart!

Never debug without a flow chart.

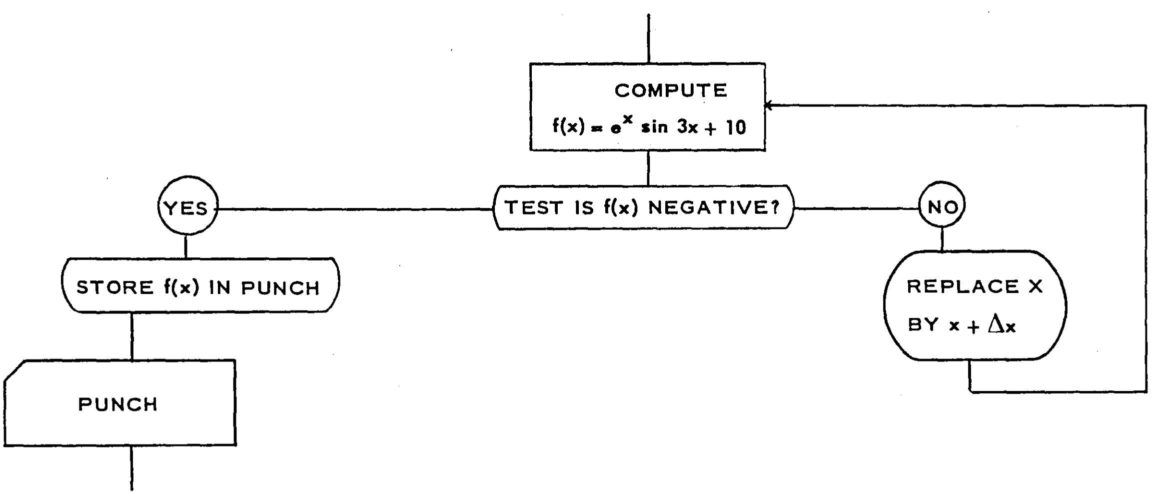
Never discuss a program with which you are unfamiliar unless you have a flow chart before you.

Never file a program you have written without also filing its flow chart. If the program is worth saving, so is the flow chart.

In discussing problems with nonprogrammers, use a flow chart rather than the program itself.

A flow chart should show the exact sequence of operations, including each branch and its alternatives. It should also indicate how and when loops are tested. It does *not* show individual steps used in computing; the overall method of attack, without details, is the object of flow charting. A typical portion of a flow chart is shown on the opposite page. The actual program for the block compute $f(x) = e^x \sin 3x + 10$ would involve two subroutines plus a dozen program steps.

There are some convenient conventions which you may (or may not) wish to adopt in writing flow charts. Boxes of specific shapes are often used for particular types of



operations—for example, a box with a truncated corner (as on an IBM card) PUNCH is often used for read and punch operations. A rectangle COMPUTE $f(x) = e^x \sin 3x + 10$ is used for arithmetic operations. Branch tests may be indicated by an elongated oval with circles indicating the possible decisions IS f(x) NEGATIVE YES NO. A fat oval is often used to indicate a modification of the instruction or the data REPLACE x by $x + \Delta$. A diamond is used to indicate that the program is sent to the console CONSOLE or to a deliberately invalid address to halt it. If a completed flow chart shows any portion which is not a closed loop or does not end in a ◇ type block, be very suspicious and check it before continuing.

Block diagramming (flow charts) of complex problems is often done in three stages.

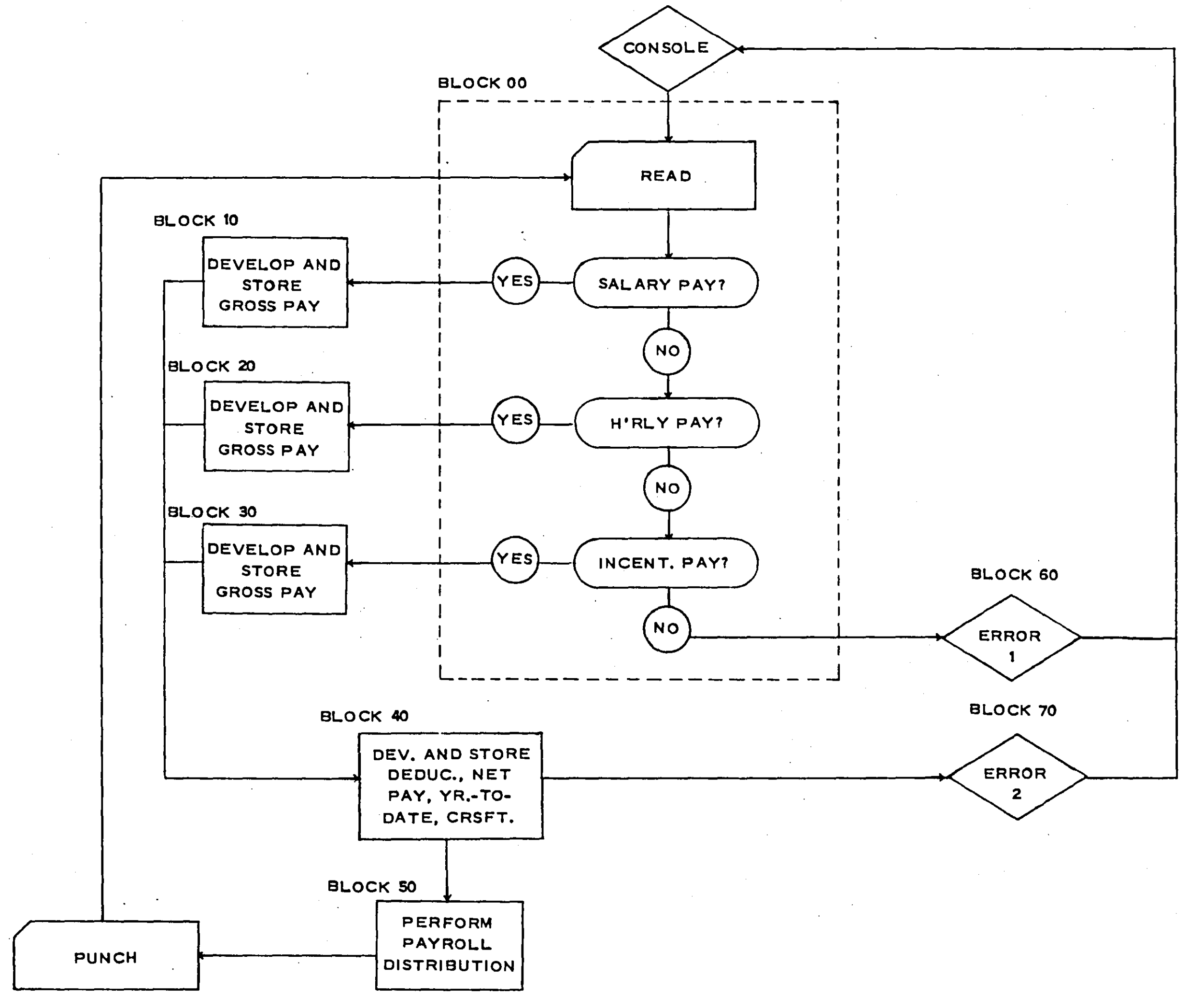
Stage 1: The overall flow chart representing the general flow of the program. We have not, as yet, worked any problems of sufficient complexity to merit this "flow chart of the flow chart" technique.

Stage 2: The semidetailed flow chart is used to explode each of the stage 1 blocks into more detail.

Stage 3: The detailed flow chart explodes each block of stage 2 into a series of logical operations. It should be noted that even at stage 3 there will be little or no mention of the specific 650 operations involved; indeed it is at this stage that the decision is often made whether to place the problem on a 650, relegate it to a lesser computer such as the 407 or 602A, or perhaps program it for one of the larger 700 series machines. A programmer can take the detailed (stage 3) block diagram and translate it into a program for whichever of these machines is suitable. So far, the flow charts we have used have all been of the stage 3 variety, since the problems have been easy to visualize.

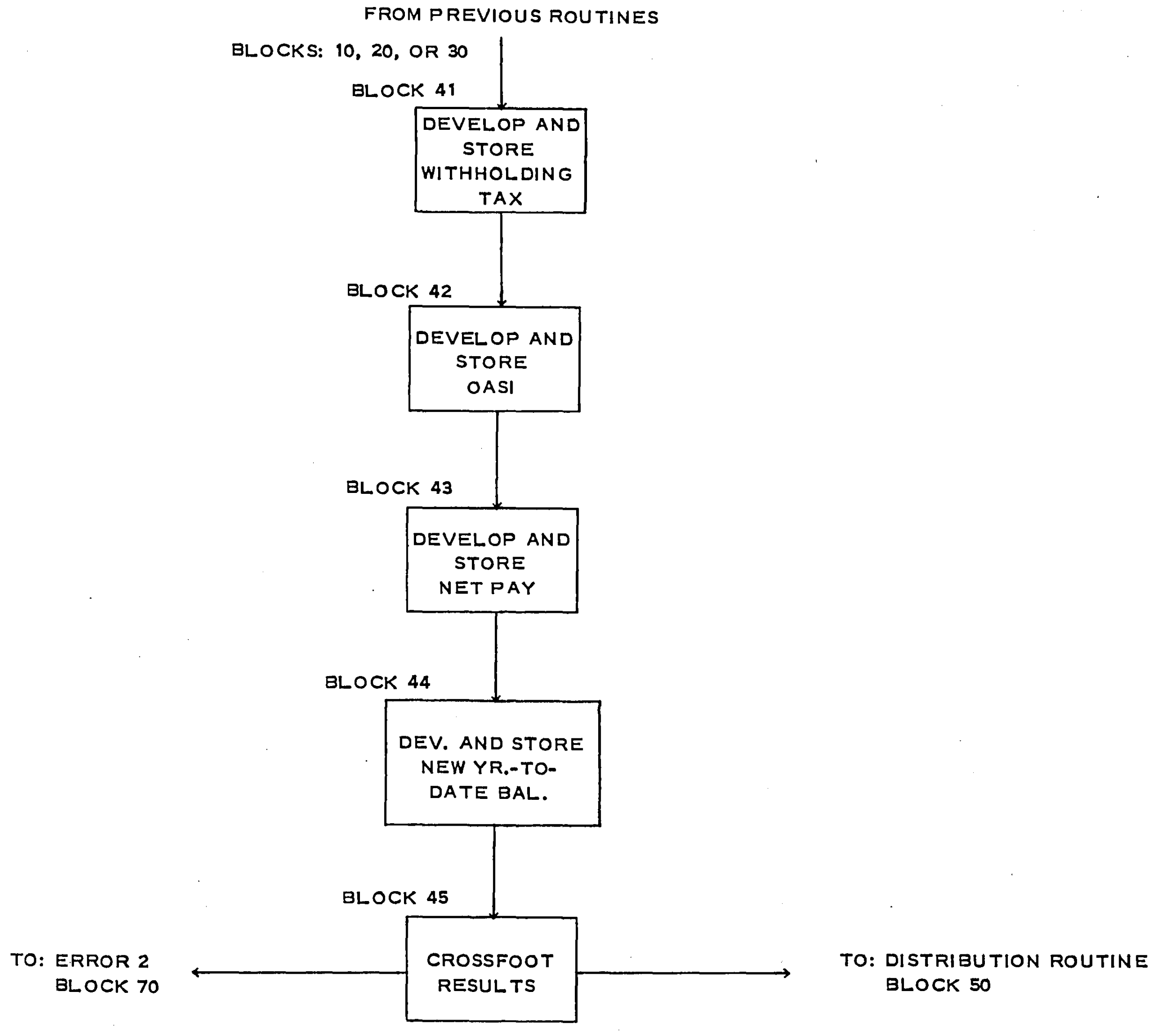
The booklet "Planning for IBM 650 Card Systems" (form F28-4016) may be helpful at this stage. The following diagrams of a "payroll problem" were taken from this source.

STAGE 1: OVERALL FLOW CHART



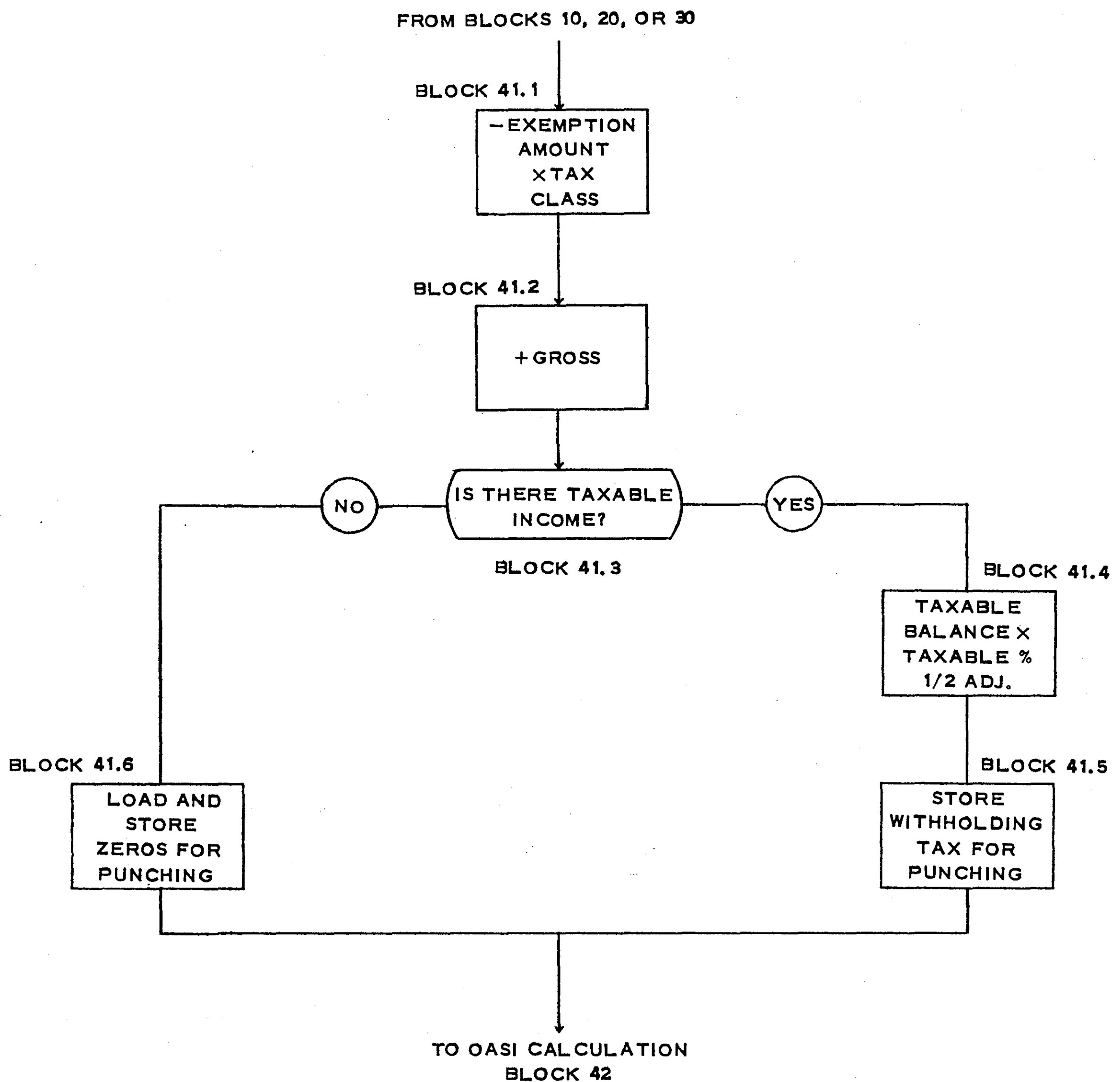
Stage 1 is omitted in uncomplicated problems.

STAGE 2: SEMI-DETAILED



There will, of course, be similar stage 2 charts for the other blocks of the stage 1 flow chart. In many of our problems both stage 1 and stage 2 have been omitted.

STAGE 3: DETAILED



There will, of course, be similar stage 3 charts for each block of each stage 2 chart.

Note: Practice suggests that one of the most useful devices in a detailed flow chart is the use of a "remarks box." The author uses a triangle and a dotted line (indicating that it is not an actual part of the flow chart) for this purpose. These "remarks" greatly simplify the reading of an unfamiliar flow chart (one you have not looked at for a month or more, or one written by another person). Any time you use an "ingenious trick," a counter which was initialized when the program was loaded, or a program which needs a bit of explanation, the reader will be well advised to insert it on the flow chart in a "remarks box." Few things help more than a dozen well-chosen remarks on a flow chart you are examining.

A good flow chart has many uses. It helps identify the exact problem, which, surprisingly enough, is often the most difficult part of writing a program. It permits later modification of the problem with a minimum of effort. Since the flow chart breaks the problem up into logical segments, it often is used to assign program sections to several programmers, thus reducing the total time needed to complete a complicated program. A flow chart is almost essential in discussing a program, and in the final write-up. One can easily justify the statement, "Programming and flow charting are synonymous—the remainder is mere coding."

6-3. LOOPS.

If your program has no loops, it probably should be relegated to a less expensive machine than the 650.

While looping is the technique which provides the 650 with so much power, it is also a serious source of programming errors. Always recheck the loop-testing steps of your program before using up valuable 650 time on a debugging run. A good technique is to pretend that each loop was to be executed only once, and analyze what your program would do (i.e., pretend the looping constants were different from those actually used in the programs). The most common errors in programming a loop result in going through the loop either $k - 1$ or $k + 1$ times instead of k times. Other common errors result in never getting out of the loop at all.

It is usually worthwhile to initialize a loop (place the correct number in the loop counter) just before it is used, rather than to depend upon its being there. It is often desirable to reuse (steal) a loop, or even an entire program section as part of a long program, and there is no telling what may be in a count box if it has been used previously. It is sometimes possible to write a few extra instructions in initializing a loop and remove them from the loop itself (i.e., from the part the program which repeats). Since three quarters of the machine time on a 500-step program may be spent in repetitions of a 10-step loop, a real savings may be effected by dropping two steps from the loop.

Another machine-time-saving technique on a program which involves loops within loops (as most advanced programs do) is to "unwind" the inner loop—i.e., to write out the innermost loop and carefully hand optimize it rather than modify the instructions. This results, of course, in a program which uses more drum locations, but may save considerable overall machine time, since often half the time on an advanced program is spent in the "inner loops."

In a loop which is repeated frequently, it may even make a difference whether instructions are "stepped" (i.e., modified) before or after passing through the loop. Which is more desirable depends upon the program involved. Sometimes it may be worthwhile to take six or seven extra steps in the nonlooping part of the program in order to keep the looping constant (count box) the same as either the number of iterations performed, or as the number still to be performed. This is particularly true if the program is to be used by many people or over a long time, thus increasing the likelihood that the "tricks" involved will be forgotten. On the other

hand, it is more often desirable to program as efficiently as possible and indicate the tricks in a "remarks box" on the flow chart.

6-4. FREE DATA

A sophisticated program makes much more output available for the same price (i.e., machine time). Any time you punch out a card with fewer than 80 columns of information, you should look around to see either if it is worthwhile combining several answers on a single output card (see section 3-1), or if other output data are available which might be useful. If such data are available store them in the punch band—they are free. In a program with too much reading or punching for the amount of computation, the console lights seem static for a moment at regular intervals while the 533 is operative. The 650 is waiting for the 533. This is often unnecessary (see Chapters 2 and 3) and wasteful, although sometimes it is unavoidable in a given portion of a program—for example, when the program is being loaded.

6-5. ECONOMY IN LOADING THE PROGRAM.

After a program is debugged and is running well, if it is to be used often it is worthwhile to place it on 7-per-card (or at least on 4- or 5-per-card) load cards in place of single-word load cards. Thus a given program will take only one seventh as long to load, in addition to taking less card storage room. There are routines which your laboratory has (or can get) which will automatically punch out a given program from the drum in any of these forms. Ask your instructor about it when you are ready. *Never* try to debug from a multiple load card.

6-6. PROGRAM ERRORS—DEBUGGING.

Inexperienced people are apt to consider it a sign of incompetence when a program fails to work the first time it is tried (or even the tenth time, if the program is complex). This is unjustified. *It is a rare program which works correctly the first time!* The sign of incompetence, then, is not that an undebugged program contains errors. If the programmer uses up a lot of machine time debugging the program *on the 650,* that is a sign of lack of experience. An experienced operator will almost always be off the machine in 10 or 15 minutes on a debugging run, since, if he doesn't find the error quickly, a trace routine will be used and the results taken to the quiet of his office for unhurried deliberation without tying up valuable 650 time. Some debugging is done on the 650, of course, but most is better done in the privacy of your own study.

Although the following suggestions are self-evident, experience suggests that even good programmers are prone to these slips. It may be well for a beginner to check them over, just as a pilot runs his checklist before taking off or landing a plane.

1. Do not place instructions (nor allow SOAP to do so) in drum locations 1951–1960 nor in any 10-word read band which your program uses. (Why not?)

2. Always list your program (on a 402 or 407), and *check the program carefully* before attemping to put it on the 650.

3. Have your key-punch operator verify the program cards before you accept them. (If a card has been verified, it will have a small semicircular nick on the right-hand edge.) A competent key-punch operator will often question your program. By all means encourage this, even if you happen to be correct this time. She may save you $20 worth of machine time in debugging your next program.

4. Find out the drum locations used by the trace routines in your installation and do not put program steps or data in these locations. If you use SOAP, block reserve (BLR) these locations.

5. If you use subroutines, be sure they do not use the same drum locations as each other or as your program or data. It is amazing how messy debugging becomes when the program starts to take a square root and ends up using instructions from a sine subroutine.

6. Always "zero" or "stop code" the drum before placing a program on it for debugging. Current policy seems to favor putting 0000000000- in each drum location. The author prefers putting 01 0000 nnnn in drum location nnnn. Then if the program progresses to an unused address, it will stop, and also show the unused drum location to which it was sent. If you do zero the drum, then be sure drum location 0000 does not contain an instruction. (Why?)

7. Check the *scaling* required by every subroutine used, and the permissible range of data.

8. Be sure that you have looped the desired number of times. Assuming that you had only wanted to loop once, what would your program have done?

9. If you are doing engineering or scientific computing rather than data processing, then learn as much as you can about numerical analysis! It is not easy, but it is vital! A worthwhile course in numerical analysis will have at least a complete course in calculus (including Taylor's series *with remainder* and improper integrals) as a prerequisite. If the Monte Carlo method is to be considered, and it should be, then probability theory is also a prerequisite.

10. Learn some modern abstract algebra—you'll need it. Modern algebra provides the vocabulary and many of the concepts needed in applied mathematics today.

11. Always have a flow chart and a program listing on the console shelf when you sit down to debug a program. Use trace routines to get off the 650 as quickly as possible (see Section 6-7).

12. Explain your program (using a flow chart) to someone who knows less about it than you do. Spouses, girl friends, and beaus make ideal victims.

13. Always debug a complex program using data for which the answer is known (run it out on a desk calculator using one- or two-digit numbers if necessary).

14. If you suspect you are "in a loop," use the address stop feature of the 650 to locate the loop.

6-7. SPECIAL TRACES AND DEBUGGING ROUTINE.

There are many debugging and tracing routines, and more are being developed. Your laboratory will have several. They are all easy to use. General descriptions are included here, while specific details will be found in the program write-up.

In general a "trace routine" punches a card for each step of the program performed. This card contains, usually, an identifying number, the location of the instruction, the instruction, the contents of the upper accumulator, the contents of the lower accumulator, the contents of the distributor, and, if appropriate, the contents of the three index registers. (*Note:* These may be the contents either before or after the instruction is performed, depending upon the trace used. See the program write-up in your lab for complete details about the trace being used.)

The simplest (and most wasteful) trace is one which starts at the first instruction of your program and traces each step. Since a card is punched for each step of the program, the 650 can proceed at only 100 program steps per minute—a slow walk compared to its usual more than 60,000 steps per minute. For this reason it becomes imperative to speed up the tracing routine. One of the simplest methods is not to trace subroutines, which are assumed to be correct. This is one reason many subroutines in your 650 library have "negative instruc-

tions.'' By properly setting the sign knob on the console switches, the trace can be made to trace only positive instructions. The next most obvious requirement is to have a trace which will run at high speed until it gets to a certain address, then begins the slower punching until a second address is reached, etc. Further refinements include traces that trace a loop only once or twice, and then let the machine run at high speed until the looping is finished, whereupon it begins tracing again.

A ''logical trace'' which punches cards out only for branch instructions is available. The snapshot trace and flow trace will punch cards only for certain key drum locations which the programmer designates as ''bus stops.'' These locations are punched on cards and loaded with the trace routine. There are many variations available, and others will spring up. Usually it is good policy to begin with a ''bus-stop'' type trace and then do a ''+ only trace'' on that portion of the program where trouble seems to occur.

Your laboratory write-up will give specific directions for using each trace, but sample instructions are included here so that you may be familiar with what to expect.

Trace Routine: Uses 100 drum locations located in any two successive bands.

A maximum of 27 ''bus stops'' are permitted, *and these must not be instructions which are modified in the program.*

Contents of the distributor and both accumulators *prior* to the execution of the ''bus-stop'' instruction are punched along with the card number, instruction location, and instruction specified by the bus stop.

Send the program to the drum location of the first address FFFF in the trace routine, either by setting the console to 00 0000 FFFF or by using a transfer card. Push the start button.

6-8. EXPERIENCE.

Frankly, what you need now is *experience!* Hop to it. The remainder of this book is devoted to a brief discussion of various more advanced techniques, but what you need now is not more reading, but about two week's experience on the 650. Then you can read the rest of this book at your leisure, and also the related IBM manuals. There is much to learn, but your most urgent need is experience. The following set of problems will get you started.

Problem Set 6-8

In each case compare your results with those of other students in your class.

1. The following questions are ones which *you* can answer by actual experiment with the 650.

(a) If the upper accumulator contains a positive number, what will the *sign* of the accumulator be after the instruction 11 8003 9000 is executed?

(b) Same as (a) but with a negative number in the upper accumulator.

(c) If your 650 is equipped with index registers, determine what sign the register has when k-k is registered, for $k > 0$, and for $k < 0$ (i.e., RAA 000k, SXA 000k).

(d) How can you load a constant into the drum using the storage entry switches? (Find two ways—one using Read In Storage, and one not using this feature.)

(e) What happens if you try to read a SOAP program into the 650 with a standard load board in the 533? Why?

(f) Does the overflow light come on if an overflow occurs with the overflow switch in (1) sense position? (2) stop position?

(g) If an instruction has a positive sign, what sign will show on the display of the program register?

(h) Same as (g) for negative sign.

(i) Same as (h) for no sign. (Booby trap.)

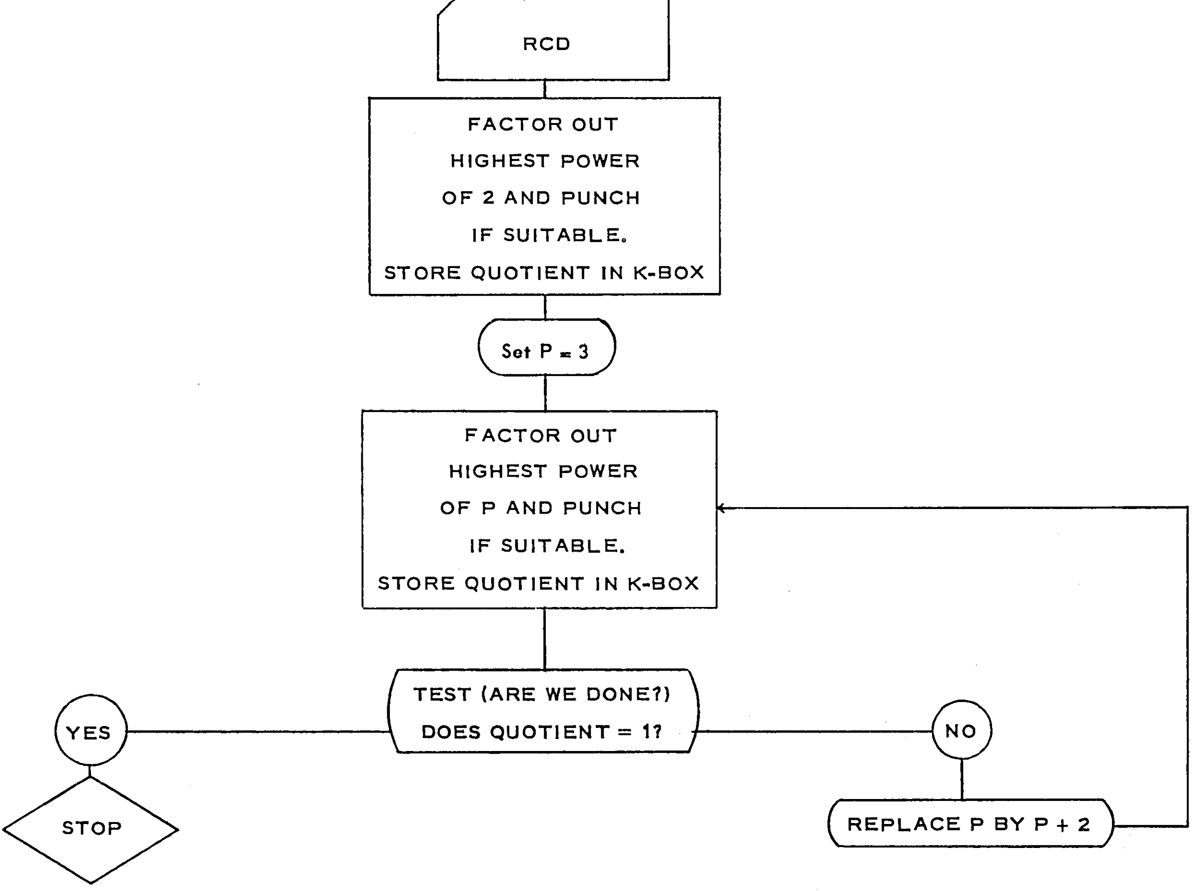
2. (a) Let drum locations 1101 to 1110 contain ten positive integers (whole numbers). Design a program which will place the difference between the largest and the smallest of these ten integers in drum location 1100. This technique is very valuable in problems involving handling and scaling of statistical data.

(b) Rework part (a) with the assumption that the ten integers may contain both positive and negative numbers.

3. (a) Devise a program which will take the ten integers of Problem 2 and rearrange them in descending order of size in drum locations 1127 to 1136.

(b) Write a program which will arrange 700 integers from drum locations 0001 to 0700 in descending order of size in drum locations 1001 to 1700. Be careful that your program does not use any data position or 1951–1960.

4. Write a program which will read a 10-digit integer K from a card and factor it into prime factors. Punch out cards containing P_i and n_i where P_i is a prime number which divides K and n_i is the highest power of P_i dividing K. Do not attempt to store primes on the drum. Instead:



5. (a) Write a program which will compute the coefficients of the straight line of best fit through a set of 10 points.

(b) Do the same for an unspecified number of points where the last point is identified by an 8-punch in Column 80 of the input card. Each input card is to contain x in word 1 and y in word 2 as fixed-point numbers.

(c) Read the IBM 650 manual on the use of the "Branch on 8 Distributor" codes 90–99 and use this in programming (b).

6. (a) Define $K! = 1 \cdot 2 \cdot 3 \cdot 4 \ldots K$ for an integer $K > 0$. Devise a program which will read a card containing two integers M and N with $M < N$, and will punch out cards containing K! for $M \leq K \leq N$. Discuss the *limits* on the size of N.

(b) Reprogram the problem of part (a) so that at least four answers are punched on each output card. Did you use 10 or 20 digits for K!? Discuss the limits of the size of K in either case.

(c) If your 650 has automatic floating-point arithmetic and/or index registers, use them in reworking (a) and (b). Again discuss limitations on the size of N.

7. Write a program which will read eight fixed-point constants x, a_0, a_1, a_2, a_3, a_4, a_5, a_6 (you decide on permissible range of size for x and for the a's), and will compute and punch $f(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4 + a_5x^5 + a_6x^6$.

8. (a) Write a program which will read a card giving x min, x max, Δ, A, and B, and will punch out a card giving this data followed by cards giving x and $f(x) = A \sin x + B e^{-x}$ for $x = x$ min, x min + Δ, x min + 2Δ, . . . , x max. Discuss the restrictions on the input data necessary in your program.

(b) Reprogram (a) so that at least four sets of answers are punched on each output card.

(c) Reprogram for floating point.

9. (a) A method for computing cube roots is the following: Let the zeroth approximation be 10. Then

$$y_{n+1} = \frac{1}{3}\left[\frac{A}{y_n^2} + 2y_n\right]$$

Write a flow chart for a subroutine which will compute the cube root of a floating-point number and exit to the instruction originally in the distributor. Assume that A is in floating point and is in the upper. The cube root should be in the upper when the routine exits. The answer should have at least 6-digit accuracy.

(b) Read the two articles on variations of Newton's method which appeared in the *American Mathematical Monthly* Vol. 51, pp. 36–8, 1944, and Vol. 52, pp. 212–4, 1945, from the viewpoint of digital computer programming.

10. Assume that eleven floating-point coefficients for a polynomial are stored sequentially in SOAP Region C (C0001 through C0011). The following SOAP program is supposed to evaluate the polynomial for the value of X (in floating point) in the upper accumulator. There are two errors in the program and one unnecessary line. The program exits to the instruction originally in the distributor (like a normal subroutine). The method for evaluating the polynomial is called "nesting." For example, to evaluate a polynomial, $a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4$, with 5 coefficients, one writes:

$$y = a_0 + x[a_1 + x\{a_2 + x(a_3 + xa_4)\}]$$

In this way one repeats a sequence of two steps—"multiply by x, add a new coefficient, repeat."

LOC	OP	DATA ADD	TAG	INSTR ADD
POLYX	STD	LEAVE		
	STU	X		
	RAU	C0011		
	RAA	ELEVN		RETRN

LOC	OP	DATA ADD	TAG	INSTR ADD T
RETRN	FMP	X		
	AUP	COOOO	A	
	SXA	0001		
	NZA	RETRN		LEAVE
ELEVN	00	0000		0011

Debug the program.

11. Write a flow chart and a SOAP program to evaluate the following series

$$\frac{2}{1^2 \cdot 3} + \frac{3}{2^2 \cdot 4} + \frac{4}{3^2 \cdot 5} + \dots + \frac{i+1}{(i^2)(i+2)} + \dots$$

to at least 3 significant figures. How many terms will this take? You can estimate the error you make by cutting off at the nth term as follows: The sum of the terms left out is

$$E = \sum_{i=n+1}^{\infty} \frac{(i+1)}{i^2(i+2)}$$

If n is large, a good approximation to E is made by replacing the sum by an integral:

$$E \approx \int_{n+1}^{\infty} \frac{1}{x^2}\,dx$$

Since you want $E < .001$, you can determine n. Note that it is much larger than your first guess!

12. Ten numbers are stored in locations 0100–0109. Write a flow chart and SOAP program which will punch these numbers, one to a card, in descending order.

13. Write a flow chart to solve a set to three simultaneous linear equations. Assume that the twelve coefficients are stored sequentially on the drum, and that the three constants b_i are read from a card.

Let the equations be:

$$A_{11}X_1 - A_{12}X_2 - A_{13}X_3 = b_1$$
$$A_{21}X_1 - A_{22}X_2 - A_{23}X_3 = b_2$$
$$A_{31}X_1 - A_{32}X_2 - A_{33}X_3 = b_3$$

(*Suggestion:* Decide on a method and try a sample calculation on paper, noting each step you must take. Then make a rough flow chart and from it make the final one.)

14. Two polynomials representing truncated power series are given by the following equations:

$$A(x) = \sum_{i=0}^{n} a_i x^i \qquad B(x) = \sum_{i=0}^{m} b_i x^i$$

The product of these two series is

$$C(x) = A(x) \cdot B(x)$$

$$\text{If } C(x) = \sum_{j=0}^{m+n} c_j x^j$$

then

$$c_j = \sum_{i=0}^{j} a_i b_{j-i}$$

For example: $c_0 = a_0 b_0$, $c_1 = a_0 b_1 + a_1 b_0$, $c_2 = a_0 b_2 + a_1 b_1 + a_2 b_0$
Assume that the coefficients of A(x) and of B(x) are stored in consecutive storage locations. There are (n + 1) a's and (m + 1) b's. Note that, in the summation for c_j, any a_i with subscript greater than n is considered zero, as is any b_i with subscript greater than m. Write a flow chart to evaluate C(x) for x = 0, 1, ... 10, punching a card for each value.

Result card format: Word 1 = x
Word 2 = C(x)

15. Write a SOAP II program to find the difference between the largest and smallest of the 1011 numbers stored in locations 0783–1793. Write comments to indicate what each step or small group of steps does. Be sure to include your flow chart.

16. Discuss a significant application of digital computers to your major field. Outline the problem, the nature and sources of data, the method of solution, and write a general flow chart. Indicate the storage and speed required to make the application practical. The application should be something which is not being done at present, to your knowledge.

17. Devise an "alternator" which will send the 650 alternately to drum locations 0602 and 0702 on successive passes through the loop. Try to invent several ways to accomplish this.

18. Read "Computation of Common Logarithms by Repeated Squaring," by O. E. Brown, *American Mathematical Monthly,* Vol. 65, pp. 118–19 (February, 1958), and write a program. Note that the article contains a suitable flow chart.

19. (a) Read "The Solution of Equations by Continued Fractions," by J. S. Frame, *American Mathematical Monthly,* Vol. 60, pp. 293–305 (May, 1953).

(b) Read some of the references given at the end of the above article.

20. Consult recent issues of the *Journal of the Association for Computing Machinery* or *Communication of the Association for Computing Machinery* and report on one article.

21. Devise two different methods of obtaining "20 zeros and a minus sign" in the accumulator. Try your proposals on an actual 650 to be sure that they will work.

22. Devise a short program which will permit you to specify a number N, $0 < N \leqq 1999$, using the four Address Selection switches on the console, and will place this number into index register A.

23. (For mathematicians only.) Use the available matrix inversion routines in your computer library to form A^{-1} for the following (ill-conditioned) matrix

$$A = \begin{bmatrix} 10 & 7 & 8 & 7 \\ 7 & 5 & 6 & 5 \\ 8 & 6 & 10 & 9 \\ 7 & 5 & 9 & 10 \end{bmatrix}$$

Now form $(A^{-1})^{-1}$ and compare it with A, or form both $A \cdot A^{-1}$ and $A^{-1} \cdot A$ and compare each with the identity matrix. Life is now always as simple as it appears.

24. Same as problem 23, using $A = \begin{bmatrix} 1 & 1/2 & 1/3 & 1/4 & 1/5 & 1/6 \\ 1/2 & 1/3 & 1/4 & 1/5 & 1/6 & 1/7 \\ 1/3 & 1/4 & 1/5 & 1/6 & 1/7 & 1/8 \\ 1/4 & 1/5 & 1/6 & 1/7 & 1/8 & 1/9 \\ 1/5 & 1/6 & 1/7 & 1/8 & 1/9 & 1/10 \\ 1/6 & 1/7 & 1/8 & 1/9 & 1/10 & 1/11 \end{bmatrix}$, the so-called Hilbert matrix of order six.

CHAPTER 7

INTERPRETIVE SYSTEMS

7-1. INTERPRETIVE SYSTEMS.

If a number of subroutines are used frequently, it becomes natural to leave them on the 650 drum more or less permanently. The next step is to load certain subroutines, along with programs to get into and out of these subroutines, onto the drum, and soon an "interpretive system" comes into existence. It is not our purpose to discuss all interpretive systems, but rather to discuss one in detail. After this, the reader should have little difficulty adjusting to other interpretive systems. One of the nicest interpretive systems available to date is the "653 Rocket Package" programmed by V. N. Huff, D. N. Turner, and O. N. Reese for the Lewis Flight Propulsion Laboratory, Cleveland. The "653 Rocket Package" makes use of the automatic floating-point and index-register features of the 650, as well as using immediate-access (core) storage. It is, therefore, not available for use on all 650's.

A year ago the author favored interpretive systems. Now, with the availability of For transit, interpretive systems no longer seem as attractive. For transit offers even more ease of programming than does an interpretive system. Furthermore, while an interpretive system is slow each time it is used, For transit uses extra machine time only the first time it is run and proceeds at full machine speed on subsequent runs. Interpretive systems will probably continue in favor for "one-time problems," but the author prefers the use of For transit or IT even for these. Chapter 8 gives further information on For transit, and may be read next if desired.

7-2. BELL SYSTEM (FLOPS).

The Bell Interpretive system, and its many variants, converts the 650 into a different machine which is slower than the 650, but much easier to program. The Bell Routine is loaded into drum locations 1000–1999. Drum locations (0) 001 to (0) 999 are available to the programer for use in storing instructions and data. Location (0) 000 has a special use as a "previous result box" in which the result of the last operation is also available. If this "previous result" is needed on the next operation, time is saved in getting it from 000 rather than from another storage location. Note that since there are only 1000 storage locations, three digits 000–999 suffice as addresses. A Bell instruction consists of a signed 10-digit number (word) which is broken up as follows:

±	OP	ADDRESS A	ADDRESS B	ADDRESS C
±	1	2 3 4	5 6 7	8 9 10

For example, the instruction

±	OP	ADDRESS A	ADDRESS B	ADDRESS C
+	3	057	621	483

says "Multiply the number in A drum location, 057, by the number in B drum location, 621, and store the result in C drum location, 483." All three numbers are in floating-point form and hence have eight significant digits and a 2-digit (mod 50) exponent. The old Bell system uses floating-point numbers of the form $x.xxxxxxx \cdot 10^{yy-50}$ = xxxxxxxxyy rather than having the decimal point before the first X as does machine automatic floating point.

To make the discussion clearer, the convention of using a bar over a drum location to indicate the 10-digit-sign contents of that drum location will be adopted. Thus:

B = 3-digit drum location
$\overline{B}$ = the floating-point number located in drum location B.

Some of the more important Bell operations are:

	OP				
ADD	1	A	B	C	Add, in floating point, the number, $\overline{A}$, stored in drum location A to the number, $\overline{B}$, stored in drum location B, and store the result, $\overline{C}$, in drum location C. This is abbreviated: $\overline{A} + \overline{B} = \overline{C}$.
SUB	2	A	B	C	$\overline{A} - \overline{B} = \overline{C}$
MPY	3	A	B	C	$\overline{A} \cdot \overline{B} = \overline{C}$
DIV	4	A	B	C	$\overline{A} / \overline{B} = \overline{C}$
NG MPY	5	A	B	C	$-\overline{A} \cdot \overline{B} = \overline{C}$ negative multiply
Go to 0_2	0	A or 0_2	B	C	

If the 1-digit operation code 0 is used, this indicates that only two addresses are needed, so a 3-digit operation code can be given in address A (now called 0_2) with B and C as addresses.

	O_1	O_2	B	C	
SQRT	0	300			$\sqrt{\overline{B}} = \overline{C}$
EXP E	0	301			$e^{\overline{B}} = \overline{C}$
LOGE	0	302			$\text{LOG}_e\ \overline{B} = \overline{C}$
SINR	0	303			SIN $\overline{B} = \overline{C}$, $\overline{B}$ IN RADIANS
COSR	0	304			COS $\overline{B} = \overline{C}$, $\overline{B}$ IN RADIANS
ART R	0	305			ARC TAN $\overline{B} = \overline{C}$, $\overline{C}$ IN RADIANS, $\lvert\overline{C}\rvert < \frac{\pi}{2}$
ABS	0	350			$\lvert\overline{B}\rvert = \overline{C}$

The system contains similar codes for exponents and logarithms to the base 10, and for functions in decimal degrees rather than radians. These will be listed later.

We illustrate a portion of a program which computes:

$$f(x) = \frac{\sin x}{1 + e^{-x^3}}$$

x, in radians, is assumed to be in drum location 500 and the constant, 1, in location 600 (1 = 1000000050)

f(x) is to be placed in location 502, with 501 and 400 used to store e^{-x^3} and $\sqrt{1 + e^{-x^3}}$ respectively. Note that the "previous results box," 000, is used as much as possible.

LOCATION OF INSTRUCTION		0_1	$A_N\,0_2$	B	C	COMMENTS
401	MPY	3	500	500	000	x^2
402	NG MPL	5	000	500	000	$-x^3$
403	EXP E	0	301	000	501	e^{-x^3} STORED IN 501
404	ADD	1	000	600	000	$1 + e^{-x^3}$
405	SQRT	0	300	000	400	$\sqrt{1 + e^{-x^3}}$ STORE TEMPORARILY IN 400
406	SIN R	0	303	500	000	SIN X
407	DIV	4	000	400	502	f(x) INTO 502

The extensive use of the "previous results box," 000, which has a quicker access time than other storage, is worthy of note. A proper "read" and "punch" instruction is all that is needed to complete the above program.

Another noteworthy thing about the program given above is the absence of a "next instruction address." An integral part of the functioning of most interpretive routines is that once the program is started, the next instruction executed is the one in the next higher drum location. (This is also true of the larger IBM-700 series computers.) If it is desirable to go from drum location k to a drum location other than k + 1, it is necessary to specifically command the machine to do so. The command

	OP	0_2	B	C
TR	0	203	000	429

transfers control to C (i.e., the next instruction executed will be $\overline{C}$ which is stored in C). The B address is ignored, but must be filled in with something, or the 650 will stop on a validity check.

7-3. READ AND PUNCH.

The card form used on Bell is not the same as that usually used for machine-language programming. Six (or sometimes 5) words are read or punched at a time, in addition to certain identifying numbers. The arrangement is as follows:

Columns	Contents
1-6	Identification
7-9	Drum location into which first word on card is to be stored

(Continued on next page)

Columns	Contents
10	Number of words on card
11	Sign of word 1
12-21	Word 1
22	Sign of word 2
23-32	Word 2
33	Sign of word 3
34-43	Word 3
44	Sign of word 4
45-54	Word 4
55	Sign of word 5
56-65	Word 5
66	Sign of word 6
67-76	Word 6
77-80	More identification

Both input (Read) and output (Punch) cards have the same form. Cards may be read into or punched out *from any location*, not merely read and punch locations. The operation codes are:

	0_1	0_2	
READ	0	400	A block of consecutive storage locations beginning at B and ending at C is read into the drum from one or several cards. Numerous checks are included here. For example, the sum of the word counts of all cards (Col. 10) must be C – B + 1, and the cards must be in consecutive order.
PCH	0	410	A block of consecutive words beginning at B and ending at C (inclusive) is punched. Six words (some programs use 5 words) and the corresponding word count are punched into each card, except possibly the last. Words remain in drum storage after punching.

7-4. ADDITIONAL BELL INSTRUCTIONS.

Alpha.	*Numer.*	*Function*
Set A	$0_2 = 500$	Set the A-address. The A-address of the instruction (B) specified by B is set equal to C.
Set B	$0_2 = 050$	Set the B-address. The B-address of the instruction (B) specified by B is set equal to C.
Set C	$0_2 = 005$	Set the C-address. The C-address of the instruction (B) specified by B is set equal to C.
Add A	$0_2 = 600$	Add to the A-address. The A-address of the instruction (B) specified by B is increased by C.
Add B	$0_2 = 060$	Add to the B-address. The B-address of the instruction (B) specified by B is increased by C.
Add C	$0_2 = 006$	Add to the C-address. The C-address of the instruction (B) specified by B is increased by C.
MOVE	$0_1 = 9$	Move. If $A \neq 000$, the block of A consecutive words beginning at B is moved into the set of A consecutive locations beginning at C. The words in the original locations are not destroyed, except where the two regions overlap. The number in location 000 ("previous result") is not affected when $C \neq 000$. Both $C > B$ and $C < B$ are permissible. An error stop occurs if $C + A - 1 > 1000$. If $A = 000$, the word (B) specified by the B-address is moved into location C and into 000. It also remains in location B.

Alpha.	*Numer.*	*Function*

MOVE with A = 000 differs from MOVE with A = 001 only in that the execution time with A = 000 is shorter and the previous result location is affected.

COND STOP	$0_2 = 200$	Stop conditionally and transfer. The machine stops if the Programmed Stop switch on the console is in the Stop position. The number 1120 is displayed on the address lights and B on the display lights. When the Program Start key is depressed, control is transferred to C. If the Programmed Stop switch is in the Run position, control is transferred to C without stopping. (*Caution:* If the Programmed Stop switch is on Run, the stops for loss of accuracy in sine, co-sine and logarithm, and the stop in the Cond Stop operation will not occur.)

This operation may be used for stopping at check points in the early running stages of a problem, with the option of avoiding the stops during later runs.

7-5. LOOP OPERATIONS.

A highly repetitive character is required of any problem to be economically handled on an automatic computer. In certain instances, such as Newton's iteration procedure for the solution of equations, a repetitive process or "loop" is conveniently programmed, merely using conditional transfer operations. In many cases, however, some of the instructions to be repeated must be slightly modified in a systematic way before each new repetition. For example, in the evaluation of a linear expression $\sum_{i=1}^{N} a_i x_i$ with the a_i and the x_i stored in blocks of consecutive locations, the addresses of a_i and x_i must be increased by 1 each time a new term is to be computed. To facilitate programming of this kind, the system provides two methods of so-called address modification. The simpler—but less general—of these methods employs a special 10-digit index register called the *loop box,* which is stored in a location normally inaccessible to the programmer. If an instruction carries a minus sign, the current contents of the loop box will be added to the instruction (in fixed-decimal arithmetic and without regard to the sign) before it is executed. If, for example, the instruction −1 531 600 901 is given and the loop box contains +0 009 000 009, the instruction actually executed by the machine would read 1 540 600 910. The original instruction remains unchanged in its storage location. At the end of a calculation, an 0_2 instruction called LOOP enables the programmer to increase the contents of the loop box by 1 in one or several address positions and to transfer control back to the beginning of the calculation. Hence, the calculation may be carried out repeatedly, each time with different addresses used in the execution of instructions with minus signs. A test provision included in the LOOP order stops the repetition after a specified number of executions and resets the loop box to zero for future use. An example will be given after the following list of LOOP operations.

Alpha.	*Numer.*	*Function*
LOOP A	$0_2 = 100$	Loop on A. The contents of the loop box are increased by 0 001 000 000. In other words, the A-segment of the loop box is increased by 1. *After* the increase, the A-segment of the loop box is compared to the B-address of the LOOP instruction. If the A-segment is less than B, control is transferred to C. If the A-segment is equal to B (or greater, which will

(Continued on next page)

Alpha.	*Numer.*	*Function*
		never be the case in normal use), the loop box is reset to zero and control proceeds to the next instruction.
LOOP B	$0_2 = 010$	Loop on B. Analogous to LOOP A, with the B-segment of the loop box now being increased and compared to the B-address of the LOOP instruction.
LOOP C	$0_2 = 001$	Loop on C. Analogous to LOOP A, with the C-segment of the loop box being increased and compared to B.
LOOP AB	$0_2 = 110$	Loop on A and B. Analogous to LOOP A. The A- and B-segments of the loop box are increased by 1 and the A-segment is compared to B.
LOOP AC	$0_2 = 101$	Loop on A and C. Analogous.
LOOP BC	$0_2 = 011$	Loop on B and C. Analogous. The B-segment is used for the comparison.
LOOP ABC	$0_2 = 111$	Loop on A, B and C. Analogous. The A-segment is used for the comparison.

To illustrate the use of a LOOP order, consider the evaluation of the linear expression $L(x) = \sum_{i=1}^{20} a_i x_i$, where the a_i and the x_i are stored in memory. In choosing storage locations for numbers, it is wise to plan in advance how they are to be used in the program. In this case, since the a_i and the x_i are to be reached using the LOOP operation, it is advantageous to store them in blocks of consecutive locations, say the a_i in 800 + i and the x_i in 900 + i, (i = 1, 2, ..., 20). Suppose L(x) is to be stored in 700. For simplicity, assume that register 700 contains zero at the beginning of the calculation and that the loop box has been reset. The entire program for this calculation might be written as follows:

INSTR. NO	ALPHA.	SING.	0_1	A OR 0_2	B	C
101	MPY	–	3	801	901	000
102	ADD	+	1	000	700	700
103	LOOP AB	+	0	110	020	101
104	NEXT INSTRUCTION IN THE PROBLEM.					

Note that the B-address of the LOOP order simply indicates the number of times the arithmetic calculation is to be performed, including the first time when the addresses are actually unmodified (modified by adding zero). The practice of starting the instruction numbering at, for examples 101, rather than 001, facilitates later additions to the beginning of a program.

The loop box is automatically reset at the beginning of a new problem and whenever a transfer out of a loop is effected by a loop order (as stated in the definitions above). Hence, the resetting of the loop box need not concern the programmer under normal conditions. If the need for resetting the loop box should arise, however, this is easily done by giving, for example, the order LOOP A with the B-address 000. According to the definition of LOOP A, this will cause control to proceed to the next instruction with a resetting of the loop box.

7-6. EXAMPLE 1.

As an introductory example, the summation in the section on LOOP operations will be programmed again using address change methods. *This would be an inefficient choice in an actual problem,* but it will illustrate the difference, as well as the analogy, between the methods. It is again assumed that register 700 contains zero at the start, but the steps analogous to the resetting of the loop box will be included.

INST.	ALPHA.	SIGN	0_1	A OR 0_2	B	C
101	SET A	+	0	500	103	801
102	SET B	+	0	050	103	901
103	MPY	+	3	[ ]	[ ]	000
104	ADD	+	1	000	700	700
105	ADD A	+	0	600	103	001
106	ADD B	+	0	060	103	001
107	TR A	+	6	103	821	103
108	NEXT INSTRUCTION IN THE PROBLEM.					

The brackets in the A- and B-addresses of instruction 103 are used to indicate that these addresses are variable and will be supplied by the program before the instruction is executed, hence what is written there when the program is loaded into the machine is irrelevant. At the end of the program when instruction 108 is reached, storage location 103 will contain +3 821 921 000. It is assumed that the summation just programmed is part of a larger problem in which it is used repeatedly. This is the reason for the SET A and SET B instructions. If 801 and 901 were simply loaded into their respective positions in instruction 103 initially, the summation would be performed correctly the first time it is used, but the next time the summation is called for, instruction 103 would read + 3 821 921 000 and erroneous calculations would result. The SET instructions could, of course, have been inserted after the completion of the summation, restoring instruction 103 to its proper value for the next application. This procedure is not recommended, however, because, in case of an interruption (e.g., error stop) during the loop, it makes it more difficult to restart the problem from the beginning without reloading the program.

7-7. EXAMPLE 2.

A more realistic example of the use of address-change methods would be a calculation involving more than one summation index or parameter. In such a case, one of the fast and convenient LOOP orders would normally be used in the "inner loop" (i.e., the loop occurring most frequently), with address-change operations controlling the "outer loop" or loops. Suppose, for example, that it is desired to calculate $S_j = \sum_{i=1}^{10} a_{ji}x_i$ for j = 1, 2, ... 5, where the a_{ji} are stored in $800 + 10_j + i$ (i.e., a_{11} is in 811, a_{12} in 812, etc.; a_{21} in 821, a_{22} in 822, and so forth), the x_i in 900 + i, and the S_j are to be stored in 700 + j. It will be assumed that register 500 contains zero. For completeness, the setting of all variable addresses to their initial values for repeated use of the summation program will be included.

INSTR.	ALPHA.	SIGN	0_1	A OR 0_2	B	C	COMMENTS
101	SET A	+	0	500	104	811	SET VARIABLE ADDRESSES TO THEIR INITIAL VALUES
102	SET C	+	0	005	107	701	
103	MOVE	+	9	000	500	400	SET REGISTER 400 TO ZERO
104	MPY	−	3	[ ]	901	000	"INNER LOOP"— I.E., SUMMATION ON I
105	ADD	+	1	000	400	400	
106	LOOP AB	+	0	110	010	104	
107	MOVE	+	9	000	400	[ ]	STORE THE RESULT
108	ADD A	+	0	600	104	010	INCREASE ADDRESSES FOR NEXT REPETITION IN THE OUTER LOOP (J-LOOP)
109	ADD C	+	0	006	107	001	

(Continued on next page)

INSTR.	ALPHA.	SIGN	0_1	A OR 0_2	B	C	COMMENTS
110	TR C	+	8	107	706	103	TEST FOR END OF J-LOOP
111	PCH	+	0	410	701	705	

7-8. EXAMPLE 3.

In conclusion, a problem will be programmed in order to illustrate the use of many of the operations and methods described above.

First, suppose it is desired to evaluate the "error function"

1. $\phi(x) = \dfrac{2}{\sqrt{\pi}} \displaystyle\int_0^x e^{-t^2}\,dt$

for a set of values $x = a, a + \Delta, a + 2\Delta, \ldots, a + 10\Delta$, using the RAND approximation

2. $\phi^*(x) = 1 - (a_1 n + a_2 n^2 + a_3 n^3 + a_4 n^4 + a_5 n^5)\, \phi'(x)$,

3. $n = 1/(1 + px)$ (p is a numerical constant),

and

4. $\phi'(x) = \dfrac{2}{\sqrt{\pi}}\, e^{-x^2}$,

and to punch out the results as well as to store them for later use. The evaluation of the polynominal in n will be faster if (2) is written in the form

5. $\phi(x) = 1 - (n(a_1 + n(a_2 + n(a_3 + n(a_4 + na_5)))))\, \phi'(x)$.

To make it possible to use the loop order in evaluating $\phi^*(x)$ this way, the coefficients a_i will be stored in consecutive locations in decreasing order. The LOOP program will be given a form applicable to an arbitrary polynomial by including a "dummy" coefficient $a_0 = 0$. Storage locations will be chosen as follows:

LOCATION	CONTENTS		
101–119	INSTRUCTIONS	(CARDS 1–4)	
200	$2/\sqrt{\pi}$	CONSTANTS (CARD 5)	
201	1		
202	a		
203	Δ		
204	P		
221	x	TEMPORARY STORAGE ("ERASABLE")	NOT ON CARD
222	$\phi'(x)$		
223	n		
301	a_5	COEFFICIENTS IN $\phi^*(x)$ (CARD 6)	
302	a_4		
303	a_3		
304	a_2		
305	a_1		
306	$a_0 = 0$		
401–410	$\phi^*(x)$	RESULTS	

CARD	LOC.	ALPHA.	SIGN	0_1	A OR 0_2	B^8	C	COMMENTS
1	101	SET C	+	0	005	114	401	SET ADDRESS OF FIRST $\phi^*(x)$
	102	MOVE	+	9	000	202	221	FIRST x IS x = a
	103	NG MPY	+	5	000	000	000	$-x^2$
	104	EXP E	+	0	301	000	000	e^{-x^2}
	105	MPY	+	3	000	200	222	$\phi'(x) = \frac{2}{\sqrt{\pi}} \cdot e^{-x^2}$

CARD	LOC.	ALPHA.	SIGN	0_1	A OR 0_2	B^8	C	COMMENTS
2	106	MPY	+	3	204	221	000	px
	107	ADD	+	1	000	201	000	1 + px
	108	DIV	+	4	201	000	223	n = 1 /(1 + px)
	109	MOVE	+	9	000	301	000	a_5 INTO 000 FOR LOOP
	110	MPY	+	3	000	223	000	PREV. RES. n
3	111	ADD	−	1	000	302	000	ADD NEXT COEFF.
	112	LOOP B	+	0	010	005	110	LOOP IN POLYNOMIAL EVAL.
	112	NG MPY	+	5	000	222	000	POLYN. $\phi'(x)$
	114	ADD	+	1	000	201	[ ]	$\phi'(x) = 1 +$ PREV. RES.
	115	ADD	+	1	221	203	221	$x + \Delta =$ NEXT x
4	116	ADD C	+	0	006	114	001	NEXT $\phi^*(x)$ ADDRESS
	117	TR C	+	8	114	412	103	TEST FOR END
	118	PCH	+	0	410	401	410	PUNCH TWO CARDS
	119	COND STOP	+	0	200	221	500	END; STOP, DISPLAY LAST x, GO TO 500 ON PROGRAM START

500 NEXT INSTRUCTION IN THE PROBLEM.

An important remark should be made: If there is no shortage of storage locations and if the programmer does not mind writing a somewhat larger number of instructions, the running time for many problems can be decreased and the logic simplified by "unwinding" the innermost loop,—i.e., by writing out the mathematical instructions in the loop in a straight sequence instead of using the LOOP operation. In the present problem, a sequence containing five MPY and four ADD instructions could replace the instructions 109–112 and also eliminate the use of the dummy coefficient a_0. The execution time for the polynomial loop would be reduced by nearly one third, and the LOOP operation could be used to replace the address-change operations in the outer loop. The polynomial evaluation accounts for about half of the total running time of this problem! In many large problems, the innermost loop consumes an even larger fraction of the running time, making it important to program the innermost loop efficiently even at the expense of apparent inefficiencies elsewhere.

7-9. SUMMARY OF BELL OPERATIONS.

BELL OPERATIONS

0_1 OPERATIONS		0_2 OPERATIONS			
NUM.	ALPHA.	NUM.	ALPHA.	NUM.	ALPHA.
0	GO TO 0_2	000	UNC STOP	300	SQRT
1	ADD	200	COND STOP	301	EXP E
2	SUB	201	TR SGN	302	LOG E
3	MPY	202	TR EXP	303	SIN R
4	DIV	203	TR	304	COS R
5	NG MPY	204	TR SUBR	305	ART R
6	TR A	205	TR PUT	350	ABS
7	TR B			351	EXP 10
8	TRC	100	LOOP A	352	LOG 10
9	MOVE	010	LOOP B	353	SIN D°
		001	LOOP C	354	COS D°
		110	LOOP AB	355	ART D°
		101	LOOP AC		
		011	LOOP BC	400	READ
		111	LOOP ABC	401	CONS
				410	PCH
		500	SET A		
		050	SET B	450	START TR

(Continued on next page)

0_1 OPERATIONS		0_2 OPERATIONS			
NUM.	ALPHA.	NUM.	ALPHA.	NUM.	ALPHA.
		005	SET C	451	STOP TR
		600	ADD A	452	ST TR ERAS
		060	ADD B		
		006	ADD C	454	NOOP
		700	SUB A		
		070	SUB B		
		007	SUB C		

A programming sheet of the following type is convenient for use with the Bell interpretive system.

BELL SYSTEM

650 PROGRAM FORM

PROGRAMMER: ______________________

PROJECT NO.: ____________ DATE: ________

DEPT. NO.: ______________________

LOC	NO. WD	MNE	±	0_1	0_2/A	B	C	REMARKS

7-10. REMARKS ON INTERPRETIVE SYSTEMS.

It should be remembered that there are over a dozen variations of the Bell system alone, not to mention SIR, MITILAC, SIS, and similar independent routines. Each has advantages, and each is slow on the machine. The use of For transit seems likely to replace interpretive routines for scientific and engineering work in many established laboratories, although there will certainly be inertia to the change.

CHAPTER 8

COMPILERS

8-1. EASY PROGRAMMING.

In the discussion of looping (Chapter 3) you discovered that it was quite feasible and desirable to let the 650 take over some of the drudgery of writing a long program by instructing it to modify its own instructions on each loop. Chapter 5 was a discussion of the Symbolic Optimum Assembly Program SOAP, in which the 650 was programmed to do the tedious work of optimizing a program and also the job of translating from an alphabetic to a machine-language code. In using SOAP it is still necessary to write one instruction in the SOAP program for each instruction in the final program. In Chapter 7 interpretive systems which sacrificed speed for ease in programming were discussed. Essentially, an interpretive system uses ingenious internal programming to convert the 650 into a machine considerably different in nature—a three-addresses machine, in the case of the Bell system. We now turn our attention to two types of compilers whose purpose is to further simplify programming by making programs more nearly parallel to the normal mathematical language used in stating problems.

8-2. IT.

There are a number of compilers in existence. Perhaps the best known of these is the Internal Translator, IT, composed by A. J. Perlis, J. W. Smith, and H. R. VanZoeren at the Computation Center of Carnegie Institute of Technology. It is based on an earlier compiler developed for the *Datatron* at Purdue. Having this machine-independent language, IT, for the two most popular medium-speed computers should help solve the current problem of babble of machine languages. In use, a programmer writes his program in the IT language, observing the necessary conventions and restrictions. An IT program deck is then read into the 650 and the program, written in IT language, is placed in the READ hopper. When the Start button is punched, the IT statements which the programmer has written are taken into the 650 and a resulting SOAP program is punched out. Each IT command may yield several SOAP commands. After the SOAP program is obtained as the output of the 533, this is run through in a normal fashion and SOAPed to obtain an optimized machine-language program which is the final program used. Two runs were necessary on the original IT compiler, but a "one pass compiler" was released at the time this book was being printed (late 1958). Since excellent manu for the IT compiler are now available, it seems unnecessary to describe IT here in more de If your laboratory uses IT, they will have a write-up describing the version used.

8-3. FOR TRANSIT.

Fortran is the language used on the IBM 700 series computers. A simplified version of Fortran, called *For transit*, is available (in five forms) for use on the 650. The basic advantages of For transit is twofold:

1. It is even easier to program in For transit than in IT.
2. A program written in For transit has the advantage that it will also run on a 700 level machine, if such a machine becomes available to the writer.

For transit is essentially a compiler's compiler. For transit translates the programmer's statement into the language of the Perlis compiler, IT. IT then translates the statement into a series of SOAP statements. The SOAP program then translates the SOAP statements into machine-language statements and, at the same time, optimizes the program. Since each For transit program requires three passes on the 650 to produce a machine-language program, the reader may feel this is a lengthy process. For a program in which the programmer merely writes fifteen or twenty steps tying together several subroutines, this may be the case, and SOAP language is probably the best tool to use in such an instance. However, in actual practice, it turns out that the For transit language is an excellent one to use. Furthermore, the amount of machine time spent in translation is actually quite negligible because once the For transit cards are punched, they are placed in a For transit box somewhere in the lab. Sometime when the 650 is not being used otherwise, a laboratory employee translates all the programs from For transit to IT with one loading of the For transit deck. Then the output, the entire group of programs, will be translated into SOAP with one loading of the IT deck. Again one loading of the SOAP deck is sufficient to produce machine-language programs from all of the original For transit programs which were in the box. This is usually done at a time when the 650 would otherwise be idle, and the programmer has no connection with it. He puts his cards in the to-be-translated box and later picks up his program in machine language along with the original For transit cards from a translated box, without having bothered about them in the interim. Most laboratories also list the programs, if requested. The programmer then schedules 650 time in the normal way. The biggest advantage of the For transit program is the ease with which programs may be written and *the reduction in the amount of machine time needed for debugging.* It may well be that, in a few years, the casual programmer will learn only For transit and never have any great understanding of the machine-language program. It seems unlikely, however, that this will extend to serious programmers who wish to take full advantage of the machine, since this comes only by a thorough understanding. If the 650 in your installation does not have the optional features necessary for use of any of the available varieties of For transit, you should inquire about other programs from your representative. In the meantime, secure a copy of the internal translator, IT, which is available in the 650 library program under file number 2.1.001, and use IT, or the newer one-pass compiler.

8-4. PROGRAMMING IN FOR TRANSIT.

If you will watch your parentheses and commas, it is fairly easy to learn to write 650 programs in For transit language. Three important conventions must be understood:

1. The symbol * denotes multiplication. That is, A * B means "A times B."
2. The symbol ** denotes raising to a power. That is, A ** 5 means A^5.
3. The equals sign, when used in For transit, has nothing to do with an equation. It means "compute the value of the right-hand member and store that value in the storage location which is designated by the left-hand member."

With these conventions in mind, the reader can see that the For transit statement

```
ROOT = (- B + SQRTF (B ** 2 - 4.0 * A * C))/(2.0 * A)
```

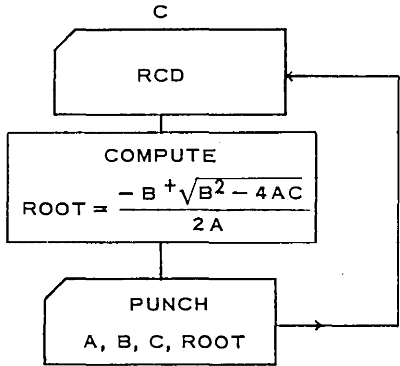
will eventually, after triple translation, give a machine-language program which will compute the value of the expression

$$[-B + \sqrt{B^2 - 4AC}]/2A$$

and will store this in the drum location called ROOT. This is, of course, not a complete program since there is no way to get the result from the machine after it has been computed. A more reasonable problem would be:

EXAMPLE 1:

Read a card from which A, B, and C are obtained, then compute the larger root of $Ax^2 + Bx + C = 0$, punch A, B, C, and the root on an output card, and repeat. The following flow chart illustrates the problem in mind.



The For transit program given below will accomplish this.

```
1  READ 1, A, B, C
2  ROOT = (- B + SQRTF (B ** 2 - 4.0 * A * C))/(2.0 * A)
3  PUNCH 1, A, B, C, ROOT
4  GO TO 1
5  STOP
```

The first statement causes the computer to read in a data card on which values of A, B, and C have been punched in floating-point form. The second statement computes the root. The third statement punches A, B, C, and the root. Notice that there is a comma after the phrase PUNCH 1. This is essential. The fourth statement causes the program to return to the statement numbered 1, and hence completes the loop. Each statement must have a statement number, but if it is not referred to in the program the number can be zero unless the preceding statement is a "DO" statement (not described here—see the For transit programmer's manual). The last statement, STOP, is required at the end of each For transit program, even though it is never used in the program. It is essential in the translation. The last line of every For transit program will read STOP. Placing a 1 after the word punch and before the comma is unnecessary as far as the basic 650 goes, but is essential if compatibility with the 700 series is desired. If more than one result is to be punched, they can be listed in the same punch command and separated by commas. Fixed-point variables are indicated by expressions involving from one to five letters and/or numbers, the first of which is I, J, K, L, M, or N. Floating-point variables are indicated by expressions involving from one to five letters and/or numbers, the first of which is a letter, but *not* I, J, K, L, M, or N. Either fixed- or floating-point variablos may be subscripted, but it is well to avoid this complication in your first few programs.

The For transit command

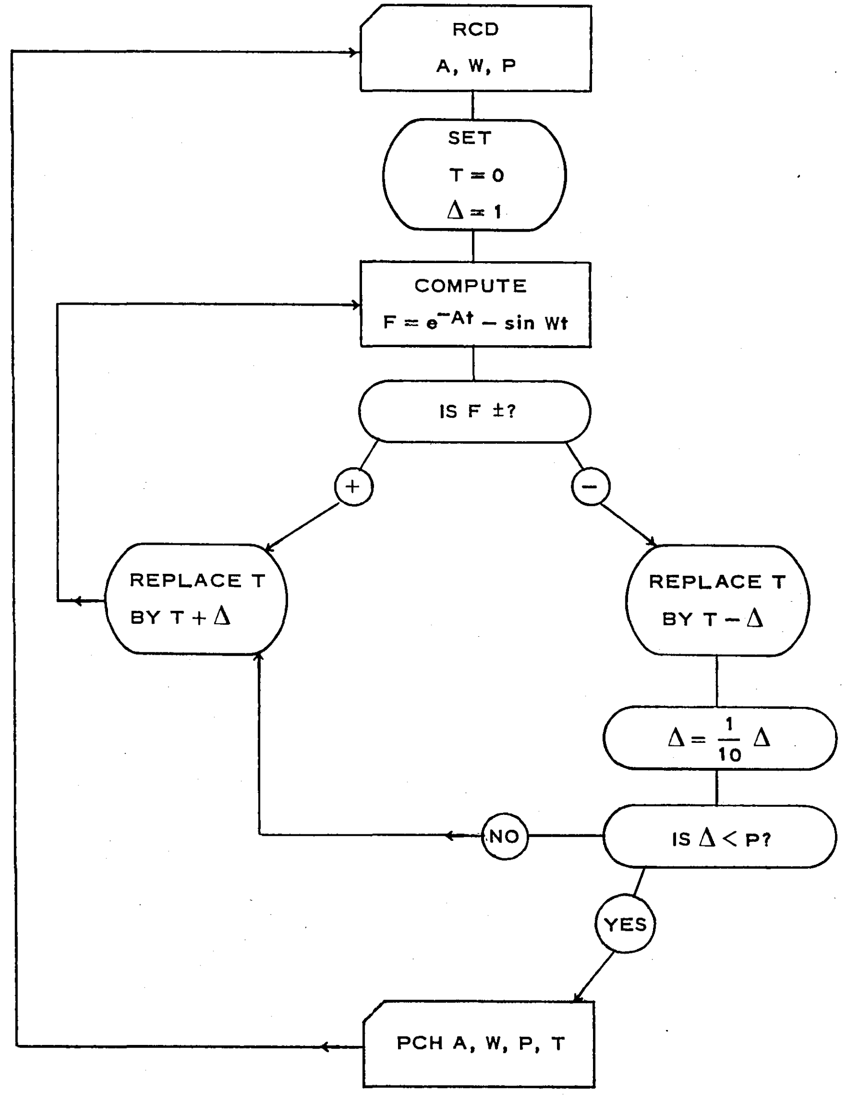
$$\text{IF } (A * T)\ m, z, p$$

is a branch command. It is translated into a series of program steps which computes and tests the sign of the expression in parentheses, $A \cdot T$ in this case. If $(A \cdot T)$ is less than zero, the next command executed will be the command with statement number m. If $(A \cdot T)$ equals zero, the next command executed will be the command with statement number z. If $(A \cdot T)$ is greater than zero, the next command will be the command with statement number p, where m, z, and p are any integers which are used as statement numbers. The parentheses may contain any desired expression.

If you have a good flow chart, the For transit program is fairly easy to write in most cases. Let us consider a refinement (since variable accuracy is permitted) of the example of Section 4-4.

EXAMPLE 2:
Read a card giving, in floating point, A, W, and P where $.0000000001 \leq P \leq .1$ represents the number of decimal places desired in the answer. Compute one solution of $e^{-At} - \sin wt = 0$, obtaining a result to as many decimal places as indicated by P.

A possible flow chart is:



The corresponding For transit program could be

```
1  READ, A, W, P
0  T = 0.0
0  DELTA = 1.0
2  F = EXP F(- A*T) - SIN F(W*T)
0  IF(F) 4, 4, 3
3  T = T + DELTA
0  GO TO 2
4  T = T - DELTA
0  DELTA = 0.1* DELTA
0  IF(DELTA - P) 5, 3, 3
5  CONTINUE
6  PUNCH, A, W, P, T
7  GO TO 1
8  STOP
```

The 650 will translate this into a suitable machine-language program for use. The resulting machine-language program contains over 180 program steps exclusive of the subroutines. The reader is now ready to turn to the For Transit *Programmer's Reference Manual* (32-7842) for additional instruction. Try a few programs. They are fun!

As you become more sophisticated you may wish to write a program to evaluate an integral using Simpson's Rule (see any calculus text). The following program purports to evaluate $\int_0^1 \frac{dx}{1 + x^2}$ by Simpson's Rule using h = .05.

Will it work? If not, why not?

```
1  K = 1
0  X = 0
0  SUM = 1.5
0  H = .05
2  X = X + H
0  IF (X - 1.0) 3, 6, 6
3  Y = 1.0/(1 + X ** 2)
0  GO TO (4, 5) K
0  PAUSE
4  SUM = SUM + 4 * Y
0  K = 2
0  GO TO 2
5  SUM = SUM + 2 * Y
0  K = 1
0  GO TO 2
6  Z = SUM * H/3
7  PUNCH, H, Z
   STOP
```

CHAPTER 9

ODDS AND ENDS

9-1. COMMON COURTESY.

Devoting space to remarks concerning the etiquette of a computer laboratory may, at first, seem unusual, but certainly such comments are not out of place in a book on programming. As in all of life, courtesy is an oil which makes the machinery run more smoothly. In any laboratory there are bound to be certain rules which must be obeyed. The number of these rules can be decreased sharply by observing a little common courtesy. For example, it is quite usual in a 650 laboratory to have one key punch located reasonably close to the console of the 650. It is common courtesy and indeed almost an unwritten rule that *no one* uses this key punch without the permission of the person who is signed up for the 650. A moment's thought will make the reason fairly obvious. Time on the 650 is worth between $70–$100 per hour. An unmanned key punch is worth perhaps a $1.25 per hour. It doesn't make economic sense to have the 650 idle while its user is forced to wait for you to remove your cards from the key punch, change the drum roll, insert his own cards, and punch a correction for a program which he is debugging. If the user of the 650 is on a production run rather than a debugging session, he will almost always give permission to use the key punch, but it is certainly boorish to use the key punch adjacent to the 650 without permission. Another item of common courtesy is not to leave your cards lying around on the 533 or the 650 or anywhere else in the laboratory after you have finished your turn. Many laboratories enforce an automatic rule that before starting a session with the 650 any loose cards found lying around are discarded in the wastebasket or the "used-card box" for salvage sale.

Certainly no one would be so unthinking as to change a wire on a public board for the 533 or some other machine. Nevertheless, the author has seen it happen. Usually the person who makes the change has every intention of replacing the wire in its original hole, but sometimes the press of other tasks makes him forget to replace it, or he forgets which is the correct hole. It is perhaps more common sense than courtesy which dictates that no beverages be placed on the console shelf or on any of the auxiliary equipment. An upset cup can cause untold damage.

There is another unwritten convention among users of IBM equipment which is both commendable and remarkable. That is the practice of sharing programs. If you are visiting a laboratory which has a program you desire, they will almost always reproduce a program deck

and give you a write-up. Experience suggests that the time to acquire such a program is when you are on the scene, rather than writing later for it, although most installations and particularly IBM World Headquarters are willing to send out programs through the mail. Requesting programs you do not actually need, or which your laboratory already has, will not make you popular. If you go to a nearby installation with the intention of acquiring several programs from them, naturally you take blank cards with you.

These items, as in the case of any matter of true courtesy, are each so obvious that it should not be necessary to mention them. Nevertheless, experience suggests that a word is in order.

9-2. SOURCES OF INFORMATION.

The field of high-speed computing is undergoing so many changes that it becomes rather difficult to secure up-to-date information in book form. Your laboratory will certainly have the *Journal of the Association for Computing Machinery*, a quarterly publication, the *Communications of the Association for Computing Machinery*, a monthly publication, and the *American Mathematical Monthly*, which is published ten times a year. If you are seriously interested in computing, you should join the Association for Computing Machinery ($6.00 a year) and the Mathematical Association of America ($5.00 a year). You will then receive the three above-mentioned journals without additional cost. The journal *Mathematical Tables and Other Aids to Computation* may also be consulted by interested persons. Most companies that make computing machines have fairly regular publications available. For example, the International Business Machines Corporation puts out a *Quarterly Journal of Research and Development*, the *Technical Newsletters*, and *The 650 Data Processing Bulletins*, all of which you may read at the laboratory.

A readily available source of information on Boolean algebra, number systems (including binary numbers), matrices, systems of linear equations, groups, etc., is the author's *Selections from Modern Abstract Algebra* (Holt, 1958)—a text designed primarily for undergraduate students.

9-3. ORGANIZATION OF A 650 LABORATORY.

The organization of a 650 laboratory, or any other computing center, will naturally depend upon the type of installation in which it is used. Two basic types of organization are widely used. These have come to be known as the "Open-shop" and the "Closed-shop" systems. In the open-shop system, everyone who uses the computer is expected to write his own program and do his own debugging and running of the programs. Usually the laboratory has key-punch operators and a machine-room supervisor available, in addition to laboratory assistants who can give you aid. In a closed-shop system, the programs are written by employees of the laboratory and the 650 is run only by laboratory employees. Obviously, many installations function somewhere between the two extremes. Most university computing centers operate on the open-shop system, since the variety of problems is almost too great to enable the laboratory to have competent personnel in the various fields. Irrespective of what system is used, *programs should always be written by a person who is quite familiar with the problem*, since additional information is often available without cost, if the programmer knows the problem. Furthermore, by cutting out the punching of intermediate results and by complete knowledge of the data (Scaling), as has been emphasized previously, a considerable saving in machine time may sometimes be realized.

The physical set-up of the laboratory should include programming rooms and conference rooms which are *quiet* and not affected by the noise of a busy laboratory. Usually the machine-room supervisor will take care of scheduling time on the 650 as well as on the auxiliary equipment. One method which has been found quite successful is to give precedence during the morning hours to the debugging of programs, and not permit any one person to reserve more than 30 minutes during the morning. Afternoon priority is given to production runs, and the maximum advanced reservation for any one person is one hour. Longer production runs are usually scheduled during the evening or Saturday afternoon and Sunday. Laboratory schedules for instructional purposes must, of course, be fitted into local requirements.

9-4. THE 650 LIBRARY.

A 650 installation will actually have two libraries. One consists primarily of books, pamphlets, and journals. The second is a library of punched cards containing subroutines and programs for the 650. It is usually the latter which is meant when one speaks of a "computer library." A small library of books can become quite jumbled and still be useful because the prospective reader can browse if he wishes. However, a library consisting of sets of punched cards is of very little use unless it is properly catalogued. A helpful practice is to have a loose-leaf notebook containing both abstracts and complete write-ups of each program and subroutine. Programs and subroutines which are used frequently should have duplicated descriptions for use in the programming room or the quiet of an individual's study. The IBM abstracts of programs provide a useful guide, along with their more complete write-ups. The write-ups furnished by IBM are master copies, suitable for reproduction by standard methods.

9-5. SUGGESTED GROUP DEMONSTRATIONS.

Every laboratory will develop its own policies concerning group demonstrations. This brief paragraph is merely a plea that:

1. You do give such group demonstrations for interested organizations, school groups, and industrial groups
2. You make such demonstrations informative and realistic rather than merely "cute and clever."

It is a fairly simple matter to program the 650 so that it will answer questions when asked. This merely requires a branching operation on some specific columns in the input data. For example, the 650 can be programmed so that whenever the first letter in card column 1 is a W the answer will be "yes"; if the letter in card column 1 is H, the answer will be "no"; if the letter in card column 1 is D, the answer will be "not at present but probably in the near future"; if the letter in card column 1 is M, the answer will be "Of course the best team will win"; etc. Then by carefully phrasing the first word of the questions when they are typed onto input cards, the desired answer can be obtained from the output. This can be modified and improved considerably by listing a dozen different affirmative answers and having the 650 select them in sequence, thus giving different appearing, but equivalent, outputs for the answer "yes." *This is "cute and clever,"* but it is *not informative*. I sincerely hope that your laboratory will *not* be guilty of using this type of demonstration for neophytes. It is a lot of fun for experienced programmers, but it gives a very wrong impression to the general public who, perhaps not surprisingly in view of the current science-fiction Sunday-supplement type of literature, show considerable awe and fear of the power possessed by a computer. Instead of reinforcing this fear, let us rather explain that the 650, or any other computer, is in essence a supersonic moron, not a giant brain. It has truly remarkable power. Its accuracy and phenom-

enal rapidity do merit awe, but there is no reason to endow it with mental properties it does not possess. Let the programmer have the credit he deserves.

There are available, through the IBM library, several rather interesting programs of a non-mathematical nature, such as the playing of tic-tac-toe or of Nim. It is also simple to devise a drum-count routine which will count the number of revolutions which the 650 drum makes in one minute by adding 1's to the accumulator. This short program can be placed on a single load card for convenience in demonstrating (the observer always receives a thrill out of being permitted to start and stop the 650 by punching buttons). A similar one-card program can be designed (see Problem 7, Set 2–5) which will place its instruction in the upper and do additions in the lower, taking the added 1 from the distributor. This program is of course independent of the drum, and will form a sum in excess of 100,000 in one minute. The solution of a set of six simultaneous linear equations in six unknowns, or the evaluation of a polynomial of high degree for a value of x selected by the observer also provides an easily understood demonstration. It is desirable to have a short statistical problem, say the computation of the means and standard deviations and possibly also the correlations for a set of 100 observations on four variables. These latter three provide understandable "workhorse" problems which are meaningful to many observers who are potential users of the 650.

9-6. SUBROUTINES AND PROGRAMS FOR YOUR LIBRARY.

The International Business Machines Corporation distributes cards and write ups for programs and subroutines. If your installation develops a particularly useful or interesting program, you are invited to submit a copy of the program and write up to IBM World Headquarters in New York. It will then be made available to anyone desiring it. If you desire a certain subroutine or program, your laboratory can write to World Headquarters giving the file number. A complete write up and a deck of program cards will be mailed to you by IBM. This is merely one of many services which IBM makes available to you without additional charge. All requests from a given laboratory or institution should be channeled through one person rather than having three or four people write for the same program. Once your laboratory has a copy of the program, it is simple to run off duplicates on your reproducing punch.

9–7. WELL.

Why are you looking here? Get busy, *write programs*. You are mature enough to be on your own now. Consult the various IBM programmers' manuals when you want information. *Experience is what you need now!* Have fun!

INDEX

The numbers refer to pages

Accumulator, 6, 18
Accuracy, 60, 70
Address
  data, 7
  instruction, 7
  invalid, 27
  stop, 33, 83
  symbolic, 75
Algebra
  abstract, 61, 83
  Boolean, 61, 105
  modern, 61, 83
Alphabetic code key, 4
American Mathematical Monthly, 105
Analysis
  multiple regression, 65
  numerical, 61, 70, 83
Approximate numbers, 60
Association for computing machinery, 105

"B-box," 44, 57
Bell system, 89
Bessel functions, 65
Biquinary, 25
Block diagramming, 8, 22, 76, 77
Boards, 4, 24, 31
  "standard," 31
  wiring for load cards, 31
Boolean algebra, 61, 105
Box
  loop, 93
  "previous results," 91
  "remarks," 81
Branching, 15, 21, 30, 48, 58
  on code number, 48
  on index register, 58
  on load card, 30
  on overflow (47 BOV), 21
"Bus stops," 84
Button
  "power-off," 25
  "power-on," 25

Canned programs, 107
Capacitor storage, 6, 27
Card, IBM, 4
  7-per-card load cards, 31, 82
Chart
  flow, 8, 21, 22, 76, 77
  planning, 9
Checking of the 650, internal, 27
Checklist, 82
Chi square, 65
"Closed-shop," 105
Code key, alphabetic, 4
Codes, operation, 11, 12, 20, back cover
  see Operation codes for specific page references
Common courtesy, 104
Communications of the A.C.M., 105
Computer laboratory, etiquette of, 104
Console, front cover, 2, 18, 24, 36
  operation, 36
  switches (8000), 18
Continued fractions, 88
Continuous function, 67
Control
  panel wiring, 4
  switch, power, 24
Cookbook directions
  to load and start program, 34
  to read contents of given drum location, 34
Count box, 43, 46, 57

Data, free, 82
Data Processing Bulletins, The 650, 105
DATATRON, 21
Debugging, 81, 82, 83
Diagram, block, 8, 21, 22, 76, 77
Directions, cookbook
  drum location, 34
  to load and start program, 34
  to read contents of given drum location, 34
Display lights, 25
Distributor, 6, 10, 18
Division, 52
  by zero, 27
Divisor, improperly positioned, 27
Drum, 5
  clear, 33, 35
  location address, 5
  to read contents of given location, 34
  zero routines, 33, 35

Equations, simultaneous linear, 87
Error
  analysis, 53, 60, 70, 82, 87
  estimation of, 87
  "function," 96
  program, 82
  "round-off," 70
  sense-light, 27
Etiquette of a computer laboratory, 104

First-reading, 31
Fixed-point variables, 101
Floating-point
  arithmetic, 69
  operations, 71
  variables, 101
Flops, 89
Flow
  chart, 8, 22, 76, 77
  of instructions, 25, 28
FORTRAN, 100
"For transit," 21, 89, 100
Fractions, continued, 88
Free data, 82
Function, error, 96

Group demonstrations, 106

h-k numbers, 49, 51
High punch, 30
Housekeeping, 16
Huff, V. N., 89

IBM card, 4
Improperly positioned divisor, 27
Improving speed, 38
Index register, 44, 57
Information, sources of, 105
input-output unit, 533, 2
Instruction, sign of, 7, 83
Instructions, 7
  flow of, 25, 28
  stepping, 38, 81
Integral, normal probability, 69
Integration, numerical, 65
Internal checking of the 650, 27
Interpretive systems, 89
IT, 99

Journal of the Association for Computing Machinery, 105
Laboratory, 650, organization of, 105
Library, the 650, 106
Light
  accumulator, 31, 32
  auxiliary, 32
  checking, 32
  clocking, 32
  control unit, 32
  data address, 31
  display, 25
  distributor, 32
  error sense, 27, 32
  input-output, 32
  instructions address, 31
  operating and checking, 31
  overflow, 32
  program, 31, 32
  storage selection, 32
Linkage, 62
List of available programs and subroutines, 65
Load cards, 30
  board wiring for, 31
Load routine
  one-instruction-per-card, 31
  4, 5, 6, 7, and 8-per-card, 31, 82
Location address, drum, 5
Locations, addressable, 18

Logical
  test, 15
  "trace," 84
Loop, 32, 40, 42, 46, 81, 93
  box, 93
  unwinding innermost, 81, 97
Looping and stepping, 42
Lower accumulator (8002), 18

Magnetic drum layout, 5
Mathematical Association of America, 105
Mathematical Tables and Other Aids to Computation, 105
Matrix, 65
Memory (or storage), 2
MITILAC, 98
Modern algebra, 61, 83
Modifying addresses, 40, 46, 57, 81
Monte Carlo, 83
Multiple regression analysis, 65
Multiplication, 18

"Negative instructions," 83
Newton's
  iteration, 93
  method, 86
Nim, 107
Normal probability integral, 69
Numbers
  approximate, 60
  h-k, 49, 51
Numerical analysis, 61, 70, 83

"Open-shop," 105
Operating and checking lights, 31
Operation, shift, 49
OPeration, 7
Operation code, invalid, 27
Operation codes, list (see back cover for complete)
  15 ALO (Add to LOwer), 12, 59
  10 AUP (Add to UPper), 20
  50 AXA (Add to indeX register A), 57
  90–99 BD (k) (Branch on 9 in Distributor), 62
  41 BMA (Branch on Minus in index register A), 59
  46 BMI (Branch on MInus), 20
  47 BOV (Branch on OVerflow), 18, 21
  32 FAD (Floating ADd), 71
  37 FAM (Floating Add Magnitude), 71
  34 FDV (Floating DiVide), 71
  39 FMP (Floating MultiPly), 71
  33 FSB (Floating SuBtract), 71
  38 FSM (Floating Subtract Magnitude), 71
  01 HLT (HaLT) or stop, 21, 33
  69 LDD (LoaD Distributor), 12
  00 NOP (No OPeration), 21
  40 NZA (branch on Non Zero in index register A), 58, 59
  45 NZE (branch on Non Zero Entire accumulator), 20
  44 NZU (branch on Non Zero in Upper half of accumulator), 14, 15, 20
  71 PCH (PunCH), 11, 24, 59
  80 RAA (Reset and Add to index register A), 58, 59
  65 RAL (Reset and Add into Lower), 12
  60 RAU (Reset accumulator and Add to Upper half), 20, 59
  70 RCD (Read CarD), 11, 24
  81 RSA (Reset and Subtract from index register A), 58
  66 RSL (Reset and Subtract from Lower), 20
  61 RSU (Reset and Subtract from Upper), 20
  36 SCT (Shift and CounT), 50, 62
  16 SLO (Subtract from LOwer), 12
  35 SLT (Shift LefT), 50
  31 SRD (Shift and RounD), 50
  30 SRT (Shift RighT), 50
  24 STD (STore Distributor), 12
  20 STL (STore Lower onto drum), 12
  21 STU (STore Upper), 20
  11 SUP (Subtract from UPper), 20
  51 SXA (Subtract from indeX register A), 58, 59
  84 TLU (Table Look Up), 62
  02 UFA (Unnormalized Floating Add), 71
Operation codes, new, 20
Optimum programming, 56, 72
Organization of a 650 laboratory, 105
Overflow, 18
  switch, 27

Perlis, A. J., 99
Permissible values, range of, 66
planning chart, 650, 47
Power control switch, 24
"power-off" button, 25
"power-on" button, 25
Power series, truncated, 87
Power unit, 2
"Previous results" box, 91
Program
  errors, 82
  to load and start, 34
  optimum, 56, 72
  start, 33
Programming
  linear, 65
  sophistication in, 76
Programs, sharing of, 104
Programs and subroutines, list of available, 65
Punch, 11, 24, 59
  "high," 30
  summary, 38
Punched hole, 27

Quarterly Journal of Research and Development, 105

Read, 24
Reese, O. N., 89
Register,
  address, 25
  operation, 25
Relocatable subroutines, 66
"Remarks box," 81
"653 rocket package," 89
"Round-off error," 70
Routine,
  drum zero, 33
  trace, 82, 83
5–10 rule, 56, 72

Scaling, 49, 51, 59, 65,
  in multiplication, 51
SENSE, 18
Sharing programs, 104
Shift operation, 49, 50
Significant digits, 70
Simpson's Rule, 103
Simultaneous linear equations, 87
SIR, 98
SIS, 98
Smith, J. W., 99
SOAP (Symbolic Optimum Assembly Program), 7, 72ff
  card, type 1, 75
Sources of information, 105
Standard board, 4
Stepping instructions, 38, 81
"Stop code," 33
Storage or memory, 2
  buffer, 24
  capacitor, 6, 27
  unit, 2, 5
Subroutines, 62, 65, 83, 107
  list of available, 65
Summary punch, 38
Switch
  "master power," 25
  overflow, 27
  power control, 24
Switches, console (8000), 18
Symbolic addresses, 75

Taylor, M. A., 83
Taylor's series, 61
Technical Newsletters, 105
Terminating technique, 43, 45
Test, logical, 14
Tic-tac-toe, 107
Time, saving of, 81, 97
Trace routine, 66, 82, 83, 84
Trapezoidal rule, 69
Truncated power series, 65, 87
Turner, D. N., 89
Type 1 SOAP card, 75

"Unwinding innermost loop," 81, 97

Van Zoeren, H. R., 99
Variables,
  fixed-point, 101
  floating-point, 101

Wiring, control panel, 4
"WORD," 4

Z-box, 57

IBM
TRADE MARK

Problem Name ______________________

INTERNATIONAL BUSINE
650 PLAN

Storage Entry Words	1	2	3	4
Memory Address				
Load Card				
Input Card				
Storage Exit Words	1	2	3	4
Memory Address				
Output Card				

Block Number	Card Number	Ref.	Location of Instruction	Instruction			Operation Abbrev.	8003 Upper Accumulator
				OP	Data	Instruction		

STANDARD REFERENCE CODES
O—Normal instruction
1—Instruction developed by 650 and Temporary Storage
2—Constant
3—Sheet reading
4—Additional description

ACHINES CORPORATION
G CHART

Problem No.______Page______of______

Form No. 22-6151-2
Printed in U.S.A.

	6	7	8	9	10
	6	7	8	9	10

8002 Accumulator	Sign	8001 Distributor	Sign	Remarks: Indexing Accumulators A	B	C	

IBM

650 S.O.A.P. PROGRAM SHEET

41	42	43 44- -47	48 49,50	51 52- -55	56	57 58- -61	62	63- -72	S.O.A.P. II COL.		
T P	S N	LOCATION	OPER. CODE	DATA ADDRESS	TAG	INSTR. ADDRESS	TAG	REMARKS	ACCUMULATOR		DISTRIBUTOR 8001
									UPPER 8003	LOWER 8002	

PROBLEM ______ PROGRAMMER ______ DATE ______ PAGE ____ OF ____

IBM®

650 PROGRAM SHEET

Form No. 22-6181-3
Printed in U. S. A.

PROBLEM: ______________________ WRITTEN BY: ______________________

INSTR NO.	LOCATION OF INSTRUCTION	OPERATION		ADDRESS		REMARKS
		ABBRV.	CODE	DATA	INSTRUCTION	

PAGE ______ OF ______

IBM

FORTRAN

CODING FORM

PROBLEM		
CODER	DATE	PAGE

C ← FOR COMMENT / STATEMENT NUMBER (1–5)	CONTINUATION (6)	FORTRAN STATEMENT (7–72)	IDENTIFICATION (73–80)

ARITHMETIC CODES

Add Upper	10	AUP
Add Lower	15	ALO
Subtract Upper	11	SUP
Subtract Lower	16	SLO
Multiply	19	MPY
Divide	14	DIV
Divide Reset Upper	64	DVR
Reset Add Upper	60	RAU
Reset Add Lower	65	RAL
Reset Subtract Upper	61	RSU
Reset Subtract Lower	66	RSL
Add Absolute (Magnitude) to Lower	17	AML
Subtract Absolute (Magnitude) from Lower	18	SML
Reset Add Absolute (Magnitude) to Lower	67	RAM
Reset Subtract Absolute (Magnitude) from Lower	68	RSM

INDEXING REGISTER CODES

REGISTER "A"

Add	50	AXA
Subtract	51	SXA
Reset Add	80	RAA
Reset Subtract	81	RSA
Branch Non-Zero	40	NZA
Branch Minus	41	BMA

REGISTER "B"

Add	52	AXB
Subtract	53	SXB
Reset Add	82	RAB
Reset Subtract	83	RSB
Branch Non-Zero	42	NZB
Branch Minus	43	BMB

REGISTER "C"

Add	58	AXC
Subtract	59	SXC
Reset Add	88	RAC
Reset Subtract	89	RSC
Branch Non-Zero	48	NZC
Branch Minus	49	BMC

BRANCHING CODES

ACCUMULATOR

Branch Non-Zero Upper	44	NZU
Branch Non-Zero	45	NZE

BRANCHING CODES CONT'D

Branch Minus	46	BMI
Branch on Overflow	47	BOV

DISTRIBUTOR

Branch on 8 in Position 10	90	BDO
Branch on 8 in Position 1	91	BD1
Branch on 8 in Positions 2-8	92-98	BD2-8
Branch on 8 in Position 9	99	BD9

INDEXING REGISTER "A"

Branch Non-Zero	40	NZA
Branch Minus	41	BMA

INDEXING REGISTER "B"

Branch Non-Zero	42	NZB
Branch Minus	43	BMB

INDEXING REGISTER "C"

Branch Non-Zero	48	NZC
Branch Minus	49	BMC

TAPE—RAMAC

Branch No Tape Signal	25	NTS
Branch No End of File	54	NEF
Branch on Inquiry	26	BIN

INPUT—OUTPUT CODES

SYNCHRONIZER 1

Read	70	RD1
Write	71	WR1
Read Conditional	72	RC1

SYNCHRONIZER 2

Read	73	RD2
Write	74	WR2
Read Conditional	75	RC2

SYNCHRONIZER 3

Read	76	RD3
Write	77	WR3
Read Conditional	78	RC3

IMMEDIATE ACCESS STORAGE

Load IAS Block	08	LIB
Load IAS	09	LDI
Store IAS Block	28	SIB
Store IAS	29	STI
Set IAS Timing Ring	27	SET

MISC. OPERATION CODES

STORE

Store Lower Accumulator	20	STL
Store Upper Accumulator	21	STU
Store "D" Address of Lower Accumulator	22	SDA
Store "I" Address of Lower Accumulator	23	SIA
Store Distributor	24	STD

SHIFT

Shift Right	30	SRT
Shift Right and Round	31	SRD
Shift Left	35	SLT
Shift Left and Count	36	SCT

MISC.

No Operation	00	NOP
Stop (Halt)	01	HLT
Load Distributor	69	LDD
Table Look-Up	84	TLU

TAPE

Read Tape Numeric	04	RTN
Read Tape Alphameric	05	RTA
Write Tape Numeric	06	WTN
Write Tape Alphameric	07	WTA
Read Tape for Checking	03	RTC
Branch No Tape Signal	25	NTS
Branch No End of File	54	NEF
Rewind Tape	55	RWD
Write Tape Mark	56	WTM
Backspace Tape	57	BST

RAMAC®—DISK STORAGE

Seek Disk Storage	85	SDS
Read Disk Storage	86	RDS
Write Disk Storage	87	WDS
Branch on Inquiry	26	BIN
Reply to Inquiry	79	RPY

FLOATING DECIMAL ARITH.

Floating Add	32	FAD
Floating Subtract	33	FSB
Floating Multiply	39	FMP
Floating Divide	34	FDV
Unnormalized Floating Add	02	UFA
Floating Add Absolute (Magnitude)	37	FAM
Floating Subtract Absolute (Magnitude)	38	FSM