October, 1957

CODING
for the
MIT-IBM 704 COMPUTER

F. Helwig, editor

Prepared at the MIT Computation Center

by

D. Arden
S. Best
F. Corbato
F. Helwig
J. McCarthy
A. Siegel
F. Verzuh
M. Watkins
M. Weinstein

The Technology Press
Massachusetts Institute of Technology
Cambridge 39, Mass.

TABLE OF CONTENTS

Appendixes

PREFACE

This is a new and slightly revised edition of a set of notes prepared by staff members of the M.I.T. Computation Center specifically to serve as a basis for a two-week course in coding for the IBM 704 given at the M.I.T. Computation Center during August, 1957. The notes are a drastic revision of a similar set of notes prepared by the staff for use during August, 1956. They are being issued in their present (rather unpolished) form as the result of a large demand for such material at the Computation Center.

The notes are written for the novice and do not assume any previous knowledge of digital computers. It is not intended, however, that these notes replace the IBM 704 Manual of Operation. Indeed certain topics, such as input and output, are treated briefly in the notes, and the manual must be referred to for complete descriptions.

Basically our topic is coding, and since there is more to coding than description of a digital computer, we have provided the reader with many illustrative examples of codes. We have also included material describing the operational systems presently available at the M.I.T. 704. This includes a brief description of the SHARE organization; descriptions of the SHARE assembly program, which provides a common language for 704 users; and a description of a post-mortem program written at M.I.T. for SHARE distribution.

A listing of subroutines distributed by the SHARE organization is also included. This list was reasonably complete at the time of publication, but will certainly become incomplete as new subroutines are developed by SHARE members.

In addition, the reader's attention is called to the FORTRAN programming system which is already described in IBM publications. These include the FORTRAN Programmer's Reference Manual and the new FORTRAN Introductory Programmer's Manual, which is to be published shortly.

Frank C. Helwig

17 October, 1957

ERRATA

Page I - 8, line 8: should read "numerical addresses whenever..."
instead of "numerical adress whenever..."

II - 4, line 6: comment should read "x → C(MQ)."
line 7: comment should read "x → C(101)."

II - 9, line 7: comment should read "a/b → C(R)."
line 10: comment should read "c/d → C(MQ)."
line 12: comment should read "(a/b) (c/d) → C(R)."

II - 10, line 8: comment should read "$(a_3x+a_2)\ x + a_1 \rightarrow C(R).$"

II - 11, line 1: replace "DVP" by "FDP."
line 3 from bottom: replace "DVP C" by "FDP C."

IV - 12, line 9: replace "-8190" by "-8191."
line 10: replace "-8191" by "8192."

V - 4, line 14: insert the following sentence: "Columns 8 to 10 make up the operation field."

V - 5, line 9 from bottom: replace "assmelby" by "assembly."

V - 8, line 6 from bottom: there should be no blanks after the commas, so that it should read:
B DEC 23178195, -251 + 251, 48

V - 13, line 9: delete the word "he."
line 15: replace "one" by "once."

VI - 3, line 10: replace "location" by "variable."

VIII - 5, line 8 from bottom: replace "if" by "it."

X - 7, line 16: comment should read "C(NUM)/C(DENOM) → C(MQ)."

XII - 7, line 12: should read "AMTST TQO* + 1" instead of
"AMTST TQO + 1."

XII - 10, last line: should read "reenter the trapping mode again after the transfer." instead of "reenter the trapping mode."

Appendix B Page 1, line 4 from bottom: replace "is" by "and is."

ERRATA (Continued)

Appendix C

Page 2, line 9: The following instruction should be inserted as a new line between "Store Logical Word" (line 9) and "Store Left-Half MQ"

Instruction: Store Zero

Mnemonic Code: S TZ α, β

Octal Value: +0600

AC	:	$A' = A$
MQ	:	$M' = M$
IR β	:	$I' = I$
R	:	$\omega' = O$

(no comments)

Page 3, 7th entry in column headed "mnemonic code" should be "CAD α, β" instead of "CPA α, β"

14th entry in column headed "Octal Value" should be +0761 instead of -0761

Page 3, line 18 from bottom: octal value for REW i should read "+0772...200+i."

Page 3, line 17 from bottom: octal value for WEF i should read "+0770...200+i."

Page 3, line 16 from bottom: octal value for BST i should read "+0764...200+i."

Page 4, line 3 from bottom:

The following instruction should be inserted between "Leave Trapping Mode" and "Redundancy Tape Test"

Instruction: End of Tape Test

Mnemonic Code: ETT

Octal Value: -0760...011

Comments: $L' = L+2$ if tape indicator is off, tape must still be selected.

INTRODUCTION

The MIT Computation Center, which was established in July, 1956, is an interdepartmental activity located in the new Karl T. Compton Laboratory (Building 26). The principal objective of the Center is to increase the number of students, staff members, and scientists qualified to use modern computing machines to further their research efforts.

The Computation Center is an activity which has many assets: qualified staff, modern computing equipment, and a brand new physical plant. The participating personnel in the Center program are located at MIT, IBM, or one of the participating New England Colleges or Universities. Specifically, the Center represents a cooperative activity involving MIT, the IBM Corporation and, at present, 25 New England Colleges and Universities.

Participating Colleges

The following New England Colleges and Universities -- in addition to MIT -- are currently participating in this program:

Amherst College
Bates College
Bennington College
Boston College
Boston University
Bowdoin College
Brandeis University
Brown University
Connecticut, University of
Dartmouth College
Harvard University
Maine, University of
Massachusetts, University of
Middlebury College
Mount Holyoke College
New Hampshire, University of

Participating Colleges (Continued)

Northeastern University
Rhode Island, University of
Tufts University
Vermont, University of
Wellesley College
Wesleyan College
Williams College
Worcester Polytechnic Institute
Yale University

An active participating by the staffs of the New England Colleges in the Computation Center program was initiated by the appointment of 24 Research Assistants and Associates at these institutions during the academic year 1956-1957. These appointees provide active liaison between the staff at the Center and the students and staff at their individual institutions. Appointments of this type will be made each year -- to insure a widespread and dynamic participating program.

Physical Plant

The physical plant of the MIT Computation Center consists of 18,000 square feet located in the recently-erected Karl T. Compton Laboratory. Specifically, the Center occupies part of the basement, the entire first floor, and part of the second floor of the Compton Laboratory. In addition, a two-story annex is used to house the IBM Type 704 Electronic Data Processing Machine (EDPM) and the associated Electric Accounting Machine (EAM) equipment.

The first floor contains adequate space for the headquarters staff, the operations staff (analysts, programmers, machine operators, etc.), IBM Institutional Representatives, New England University Research Assistants and Associates, MIT Research Assistants and Associates, classroom and seminar room, as well as the 704 computer. The basement provides space for the EAM machines, the systems research laboratory, dark room facilities, the electrical power plant, and the air conditioning equipment. The second floor provides space for the programming research staff, the visiting professors, and the library and document room.

All this area has been furnished in a first-class manner to facilitate the progress of research at the Center.

The 704 Computer and Associated Equipment

The computational facilities in the Center are supported in large measure by the IBM Corporation. Specifically, IBM is providing the 704 computer, the associated EAM equipment, and the associated maintenance personnel on a gratis basis. The following machine complement is available in the Center:

MACHINE COMPLEMENT IN THE MIT COMPUTATION CENTER

Quantity	Type	Description
1	704	Analytical Control Unit
1	711	Punched Card Reader
1	716	Alphabetic Printer
1	721	Punched Card Recorder
1	733	Magnetic Drum Unit (8192 words)
1	736	Power Frame No. 1
2	737	Magnetic Core Storage (8192 words)
1	740	CRT Output Recorder
1	741	Power Frame No. 2
1	746	Power Distribution Unit
1	753	Magnetic Tape Control Unit
10	727	Magnetic Tape Units
1	780	CRT Display Unit

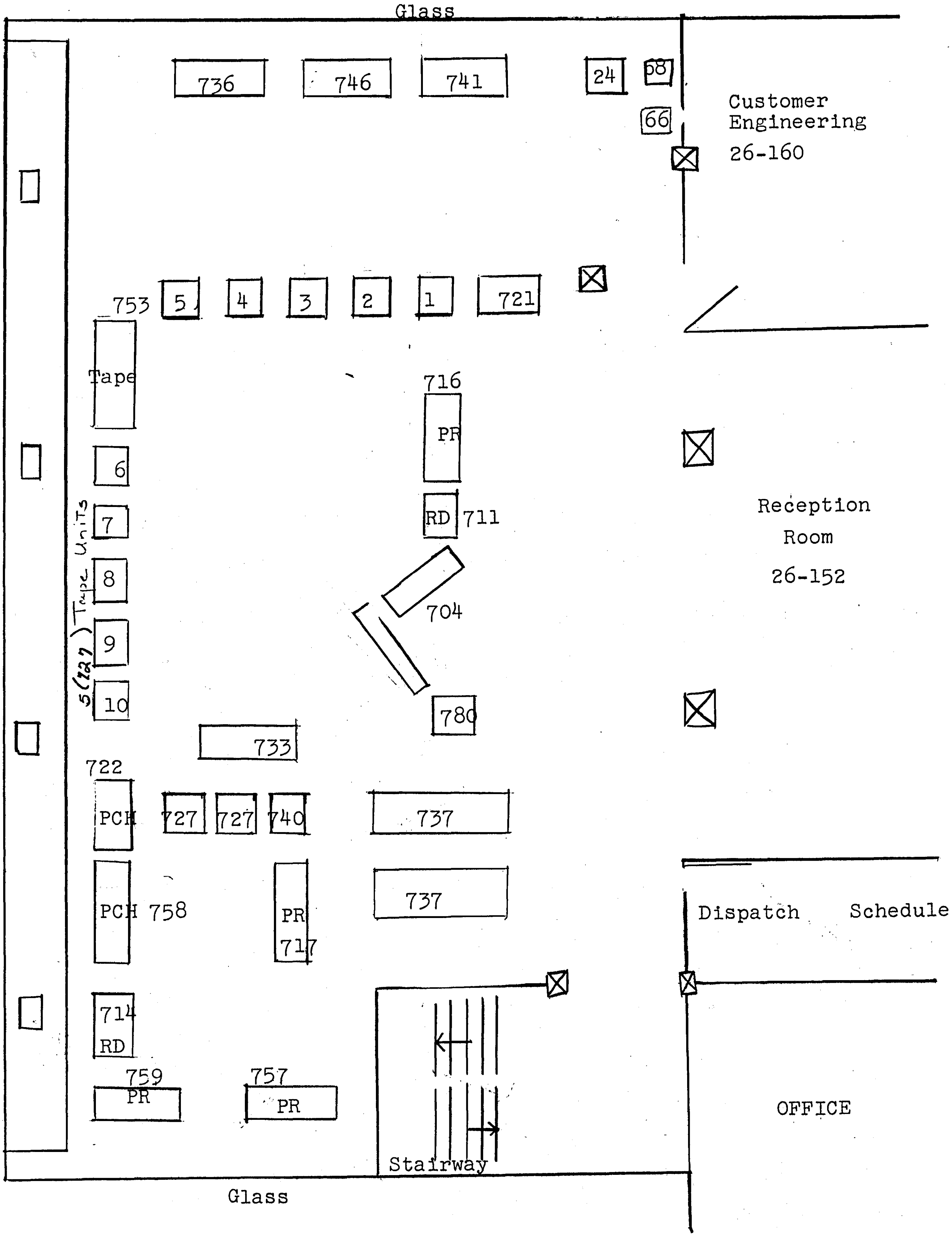
Off-Line Equipment

Quantity	Type	Description
1	714	Card Reader
1	717	Alphabetic Printer
1	722	Card Punch
2	727	Magnetic Tape Units
1	757	Printer Control Unit
1	758	Punch Control Unit
1	759	Card Reader Control Unit

Auxiliary Machines

1	024	Key Punch
5	026	Key Punches
3	056	Verifiers
1	066	Printing Card Unit)Data Transceiver
1	068	Telephone Signal Unit)and Receiver
1	077	Collator
1	082	Sorter
1	407	Accounting Machine
1	519	Reproducer
1	552	Interpreter

The actual location of the machines in the 704 Room is shown on the attached physical layout sheet.


Glass
736
746
741
24
68
66
Customer Engineering
26-160
753
5
4
3
2
1
721
Tape
716
PR
6
7
8
9
10
5(727) Tape Units
RD
711
704
780
Reception Room
26-152
733
722
PCH
727
727
740
737
737
PCH 758
PR
717
Dispatch
Schedule
714
RD
759
PR
757
PR
Stairway
OFFICE
Glass

Additional Description of 704 Components

The Type 66-68 IBM Transceiver equipment will permit remote programming for the Type 704 computer. Specifically, the Type 66 Printing Card Unit will receive approximately 14 card columns of information per second over *telephone* lines. The received information is simultaneously *printed* along the top of the card while it is being *punched* into the same card. At this transmission speed an average of ten (10) fully-punched 8-column cards may be received each minute -- more if fewer than 30 columns are punched in each card.

Four independent transmissions can be made simultaneously over the same telephone wires, provided each independent transmitter has its own transceiver at each end of the line. Simultaneous transmission is accomplished by use of the following four channel frequencies: 800, 1300, 1800, and 2300 cycles per second.

Initially, the MIT Computation Center will use only *one* transceiver operating at 1300 cycles per second on a 4-wire signal unit. The initial telephone circuit will connect the 704 Computer installation at Poughkeepsie, New York to the Center in Cambridge, Massachusetts.

Use of Dual-Purpose Equipment

There are only 12 magnetic tape units at the Center and ten (10) of these are directly connected to the main frame and are available to the programmer. Since there are three (3) additional sets of peripheral or off-line equipment, namely:

1. Magnetic tape-to-punched card converter,
2. Magnetic tape-to-printer converter,
3. Punched card-to-magnetic tape converter,

there is need for *dual* use of one of the magnetic tape units.

Accordingly, the physical layout of the equipment and cables has been designed to permit use of tape unit No. 10 on a dual basis, either as on-line tape unit No. 10, or as an off-line unit with the off-line card punch Type 722. (The change from on-line to off-line usage is effected by manually changing the signal cable connector on tape unit No. 10.)

Personnel at the Center

The personnel of the Computation Center may be roughly classified into the following groups:

1. Administrative and Supervisory Staff,
2. Members of the Teaching Staff,
3. Members of the Operations Unit,
4. Members of the Programming Research Unit,
5. Members of the IBM Research and Associate Program.

The core of the above groups was obtained by selecting key individuals from the staff of the Office of Statistical Services and the staff of the Scientific and Engineering Calculation Group at the Digital Computer Laboratory.

The composition of the IBM Research Assistant and Associate program will naturally vary from time to time, since these appointments are made on an annual basis. Some of these appointments are renewed for a second year; however, the principal purpose of the appointment -- that of indoctrination in computer application and programming -- is accomplished the first year. At the end of the first year, these men are well-qualified to transmit their knowledge to other students and staff at their respective universities.

CHAPTER I

AN INTRODUCTION TO THE 704

The modern computer is really a large, but elementary device -- at least in principle. An understanding of a computer is perhaps best given by listing the major components of a particular computer, the IBM 704, and then describing how these components interact with each other. Briefly, these components are:

1. A large, fast-access memory or information storage device
2. An arithmetic element
3. An electronic control element
4. Input and output equipment
5. Auxiliary memory devices to supplement items 1 and 4.

The first item, a large memory unit, is a device capable of storing (although not necessarily all at once) all the information required to perform a computation. This information is stored in convenient units by words. Thus in the IBM 704 computer at MIT there is an 8192 word high-speed magnetic core memory. Although it need not unduly concern the user at present each word consists of 36 binary digits (bits), each bit capable of having a value of one or zero. Finally each of the one-word storage locations in the memory unit, (often called a register or cell), has an arbitrary numerical address from 0 to 8191 which is permanently wired into the machine. In effect then the memory unit of the computer is a collection of labelled pigeon holes which will hold all the numerical values of a problem before, during, and after computation.

The second item, the Arithmetic Element, consists of several special registers: the Accumulator Register (AC) the Multiplier-Quotient Register (MQ) and a Storage Register (SR). Each of these registers can contain one word and will respond to signals from the Control Element, described shortly. Usually the SR will contain a word which is to be combined in some definite manner with a word in the AC or the MQ according to signals sent from the Control Element. For example the simple addition of two numbers, one in the AC and the other in the SR, will result in the sum being left in the AC.

The third item, the Control Element,is analogous to a "central nervous system" in the computer. An important part of this system is two registers: the Instruction Location Counter (ILC) and the Instruction Register (IR).

Having established in this way the more important terms, it is now possible to clarify their meaning by considering the process of computer operation. The most basic operation consists, in general, of information being brought from memory to the arithmetic unit, processed by means of a standard operation and the resultant information perhaps being stored in the memory; to accomplish this operation an instruction (i.e. a number code for the process desired) is given to the Control Element which then selects from the memory the specified information and places it in the SR, impulses the Arithmetic Element to perform the operation and then stores the result whenever the instruction so specifies.

Now clearly if instructions were to be given to the Control Element by a human machine operator, the execution of a sequence of instructions could be no faster than the human operator. A possible solution would be to prepare the sequence of instructions in a loop of perforated coded paper tape, but this too would be limited by the speed of mechanical rotation and reading of the tape; (some of the earlier computers did just this). An ingenious solution to this problem is to place the sequence of number-coded instructions in the memory unit of the computer itself, for then the execution of the instructions is only limited by the speed of the electronic circuitry and suffers from neither mechanical nor human intervention. This latter concept, often called that of the stored-program, is one of the important distinctions of the modern high-speed digital computer. A second distinction and a very important feature of a stored program computer is that since both the instructions and data are stored in the same memory unit, it is quite possible for sequences of instructions to actually modify themselves. The ramifications of this second distinction will be explored in later chapters.

Let us consider as an example the execution of an elementary sequence of instructions arbitrarily located in memory locations 127, 128, etc.

Location	Operation	Address (of word to be operated on)
127	CLA	199
128	ADD	198
129	STO	200
130	TRA	353
	. . . .	
198	(Contains value of x)	
199	(Contains value of y)	
200	(Contains value of sum)	

As implied here, the 704 computer is a single-address computer so that each instruction consists of an operation, (usually abbreviated by 3 letters) and an address referring to one word in the memory. (Many other computers for reasons of design efficiency use multi-address instructions). A second implication in the example shown is that the Control Element performs the instructions in the sequence of their location in memory. There are a few instructions, which cause exceptions to this rule, but these discrepancies are considered part of the instruction definition. In fact these exceptional instructions which cause jumps in the instruction execution sequence will be seen in later chapters to play a vital role in the decision and repetition capabilities of the computer.

Returning to the example given, the computer operation now will be traced to ensure that the basic concepts are established. The assumption made is that the Control Element is manually started with the Instruction Location Counter (ILC) preset to the value 127. The first step the Control Element performs is to copy the instruction in memory location 127 into the instruction register (IR). Examination by the Control Element of the address section of the IR reveals that the word located at address 199 is to be operated on so this word is copied into the Storage Register (SR). Next the Control Element carries out the operation indicated by that section of the instruction in the IR. In the particular example here, CLA means "clear and add (to the AC)" so the effect of the operation is to copy the contents of the SR into the AC. The final step performed by the Control Element is to increase the ILC by one (to 128 in this case), and then repeat the pattern described by placing the instruction located in 128 in the IR, placing the word stored at location 198 in the SR and so forth. It should be clear from this description that the computer can operate at high speeds in a fully automatic fashion. It should also be clear that all sequences of instructions were pre-arranged inside the computer. The practical use of a computer hinges on this latter accomplishment which is called programming if it involves the totality of computer operation or coding if it concerns only the sequences of instructions.

Having completed the basic operating description of a computer it is now possible to finish discussing the major computer components. The fourth item listed previously, input-and-output equipment, serves to transmit information to-and-from the outside world and the memory unit. Thus for input devices on the 704 computer there are a card-reader or magnetic tape units. Similarly for output equipment there is a printer, a card punch, magnetic tape units and a photographing oscilloscope. It is an important feature that all the input and output devices can be actuated and controlled whenever special instructions are executed in the computer; thus the devices are said to be under "program control."

The auxiliary storage devices mentioned previously as item five are of two types. The first is the use of magnetic tape also as a supplement to the storage capacity of the memory unit. The second device is a rotating magnetic drum. The drum units on the MIT 704 offer another 8192 words of storage, any word of which may be brought into the main core memory unit in a time bounded by that of one drum rotation. Thus for some purposes the drum as a storage device is inferior to core memory but superior to magnetic tape where the time required to bring a word into core memory depends on the position of the word on the tape.

This concludes the broad brush-stroke description of a computer. The remaining chapters will discuss various aspects of the essential details. As a general introduction,though,a quick survey will be made of some of the conventions involving computer words.

It was already noted earlier that there were two broad categories of words used in the 704. These were instructions and data words, each composed of 36 binary bits and indistinguishable except by usage. However there are several convenient word usage conventions which are strongly favored by the instruction codes available on the 704. Thus the binary bits of an instruction are divided into standard sections. In most of the instructions, the first 18 bits give the operation code, the next 3 bits the tag value (the use of this is described under the chapter on indexing), and the last 15 bits give the address section of the word that the instruction refers to. In a few instructions the first 18 bits of the operation section are further divided into a 3 bit prefix and a 15 bit decrement section, again described in the chapter on indexing.

No attempt has been made to describe in any detail the binary nature of the computer because in practice there are standard "translation" procedures always available. Hence when a person writes down CLA as an instruction, this when read into the computer is translated into an 18 bit operation code. An additional and similar convenience is to be able to avoid the use of numerical ~~address~~ addresses whenever writing down sequences of instructions. This is done by using what are known as symbolic locations or more generally symbols. These are merely arbitrary 5-character (or less) names for specific locations or addresses. Thus the previous example of coding might have been written as:

```
Location      Instruction      Address

 START            CLA            Y
                  ADD            X
                  STO            ARG
                  TRA            NEXT
                . . . .
    X           (Contains value of x)
    Y           (Contains value of y)
    ARG         (Contains value of sum)
```

It is important to realize that this algebra-like convenience produces exactly the same numerical values for instructions and locations inside the computer as the previous numerical example; all that has changed is the convention for describing these instructions and locations.

The other major category of words used in the 704 is that of words used to represent arithmetic quantites. There are two major types, those for fixed-point numbers and those for floating-point numbers. Again it should be emphasized that these conventions are only useful because there are explicit 704 instructions which manipulate words according to these conventions. In fixed-point words, the first bit is used to describe the sign (0 is positive, 1 is negative) and the remaining 35 bits give the magnitude of the significant figures. Inasmuch as the binary point is not a tangible thing inside the computer, a fixed-point number can either be an integer or a fraction depending on whether one interprets the binary point as being at the left-hand end or the right-hand end of the magnitude.

In a similar way, floating-point numbers, that is, numbers which are represented by a fraction multiplied by 2 raised to a power, are represented in the following way: The first bit is the sign of the fraction, the next 8 bits are the always-positive characteristic (by definition, the exponent plus 128), and the remaining 27 bits are the magnitude of the fraction.

Just as in the instructions, where the convenient abbreviations and symbols are translated whenever instructions are placed in the computer, there are convenient ways of writing fixed-and floating-point numbers in normal decimal form for the computer. For example, simply writing down the pseudo-instruction DEC -5, will translate (because there is no decimal point) into the computer as the fixed-point integer minus five. Similarly DEC -5.translates (because there is a decimal point) into a floating-point minus five, and DEC -.5B translates (because there is a B) into the fixed-point fraction minus one-half. Further discussion of this translation process (often misleadingly called <u>assembly</u>) and the translation syntax or rules are given in the chapter describing the SHARE Assembly Program.

The foregoing chapter briefly describes the basic word structure used in the 704 computer. For clarification of details and definitions the IBM 704 manual will be found useful as a reference. In particular binary arithmetic and conversion are described in an Appendix.

CHAPTER II

FLOATING POINT ARITHMETIC IN THE 704

In this chapter we will show how the 704 can be made to evaluate simple numerical expressions, as for example (a + b)c.

Some Conventions

The reader already knows that in order to have the computer do any computations a program must be written in terms of the elementary instructions which the machine can obey. When we write programs down on paper, we represent the instructions by three letter abbreviations which are derived from the name of the instruction. Clear and add is represented by the abbreviation CLA. Such abbreviations we call operation codes. In general, the instruction will have an address. Usually the address determines which storage location the instruction refers to, and accordingly it may be the integer number of that storage location. For example: CLA 100 refers to storage location 100.

We will more often want to write some symbol instead of an integer with the understanding that the symbol represents an integer. For example:

```
CLA    A
```

where A stands for a permissible integer.

Furthermore it is often convenient to write comments on the same line with the instruction to explain its purpose or define it, like this:

```
    CLA   A       This is a 704 instruction
```

When composing a program, we arrange the instructions in a vertical column and imagine that the computer obeys them in sequence reading down. Thus:

```
    CLA   A       First this one
    STO   B       then this one
    CLA   C       then this one
    STO   D       etc.
```

Conventions Used in Comments and Definitions

The two most important registers in the arithmetic element, the accumulator and the multiplier-quotient registers, we will abbreviate by (AC) and (MQ) respectively.

Often we will want to talk about the contents of a certain storage location. We will write

$$C(100)$$

for "the contents of storage location 100," and

$$C(A)\ ,\ C(AC)$$

for "the contents of storage location A" and "the contents of the accumulator" respectively. Also we will use the symbol "→" to mean "replaces." Thus

$$C(A) \rightarrow C(AC)$$

will mean "the contents of storage location A replaces the contents of the accumulator."

Now we are ready to begin.

The Administrative Instructions

First we will consider some instructions, which do no computing, but are very important. They are used to transmit words between storage and arithmetic element. We call them administrative instructions. They are:

Definition

1. Clear and Add

   CLA A $C(A) \rightarrow C(AC)$

2. Load MQ

   LDQ A $C(A) \rightarrow C(MQ)$

3. Store

   STO A $C(AC) \rightarrow C(A)$

4. Store MQ

   STQ A $C(MQ) \rightarrow C(A)$

Some Simple Examples of Programs

We will write the following computer instruction at the end of sample programs which we exhibit:

5. Halt and Proceed

   HPR causes computer to halt; it will proceed to the next instruction if then started manually.

Example I: If C(100) = x, then either of the following programs may be used to place x in location 101.

```
         | OPER |   Address        Comments
---------+------+----------------------------------
         | CLA  |     100         x  → C(AC)
         | STO  |     101         x  → C(101)
         | HPR  |                   HALT
```

or

```
         | LDQ  |     100         x → C(MQ)
         | STQ  |     101         x → C(101)
         | HPR  |                   HALT
```

In both cases C(100) remains undisturbed so that x ends up in both locations 100 and 101.

Example II: Suppose it is desired to exchange C(A) and C(B), Let:

$$C(A) = x \quad ; \quad C(B) = y$$

```
| LDQ | A             x → C(MQ)
| CLA | B             y → C(AC)
| STO | A             y → C(A)
| STQ | B             x → C(B)
| HPR |               halt, x and y are inter-
                      changed
```

Next, we shall consider how the administrative instructions can be combined with arithmetic instructions to do simple calculations; but first we will briefly discuss a kind of number that the 704 is designed to deal with.

Floating Point Numbers

In many of the computational problems that arise in the sciences and engineering one encounters numbers that vary greatly in magnitude. To save writing and to save paper such numbers are usually written, for example, in this way:

$$5.213 \times 10^{-6} \quad , \quad 3.213 \times 10^{10}$$

rather than in the equivalent forms:

$$.000005213 \quad , \quad 32130000000.$$

The first way of writing these numbers is an example of what we shall call floating point notation. As a convenience for doing calculations where the magnitudes of the numbers do vary widely, the 704 has instructions which do arithmetic with numbers of a similar form. We call numbers of this kind floating point numbers. Since the 704 is a binary machine, these numbers are of the form

$$N = x \cdot 2^{i}$$

(rather than $x \cdot 10^{i}$). The integer i is called the exponent and is restricted to lie in the range

$$-128 \leqslant i \leqslant +127$$

and x is called the fraction and is restricted to lie in the range

$$-1 < x < 1.$$

If x also satisfies either of the two conditions

$$\frac{1}{2} \leqslant |x| < 1 \qquad \text{or} \qquad x = 0$$

then we say that $N = x \cdot 2^{i}$ is a normalized floating point number. In what follows, we shall assume that all floating point numbers are normalized unless a specific statement to the contrary is made.

The fraction, x, is not a continuous variable but can assume only integral multiples of the number 2^{-27}. This fact we usually express by saying that x (and therefore N) has a precision of 27 binary digits. This is a precision slightly greater than 8 decimal digits.

How Floating Point Numbers are Written When Programming

When we are writing a program, we may write floating point numbers in ordinary decimal notation since there is an assembly program which can translate this notation into the internal binary floating point numbers of the 704.

To be specific, if we wanted to have the number $.51 \times 10^{+2}$ stored as a floating point number, we would write on our coding sheet:

|DEC| 51.

(The decimal point is essential because we wish to reserve the notation

|DEC| 51

to mean something quite different). However, we may also write:

|DEC| .51E+2 notice E+2 means 10^2

Now it may be helpful to restate two properties of the 704's floating point numbers in decimal notation.

1. The absolute value of a floating point number must either be 0 or must lie between

the approximate limits $(10^{-38}, 10^{+38})$.

2. The maximum precision of a floating point number is slightly more than 8 decimal digits.

Thus we see from (1) that the number 5.0×10^{-41} cannot be stored as a floating point number because its magnitude is too small; and from (2) we see that it would be silly to write

```
|DEC| -1.234567890123E+2
```

because nothing beyond the 9th significant digit could possibly affect the stored result.

The Floating Arithmetic Instructions

We are now ready to introduce the four basic floating arithmetic instructions. In every case, if the operands are normalized floating point numbers, the results will be also*

1. Floating Add

```
FAD  B        C(AC) + C(B) → C(AC)
```

2. Floating Subtract

```
FSB  B        C(AC) - C(B) → C(AC)
```

* The 704 also has some floating-point instructions which do not produce normalized results. In practice these instructions (UFA,UFS,and UFM) are used only rarely and in rather tricky and obscure ways. The interested reader may consult the IBM 704 Manual under the topic of "Fixing a Floating-Point Number." We advise him to first study the 704 fixed point instructions, however.

3. Floating Multiply

FMP B $\qquad C(MQ) * C(B) \longrightarrow C(AC)$

4. Floating Divide or Proceed

FDP B $\quad$ if $C(B) \neq 0$; $C(AC)/C(B) \longrightarrow C(MQ)$

The instructions FAD, FSB, and FMP do not leave the MQ undisturbed. In fact, these instructions leave a value in the MQ such that the number

$$C(AC) + C(MQ)$$

is a better approximation to the true result than C(AC) is. In "single precision" work, however, the C(MQ) is ignored.

The instruction FDP leaves the remainder in the AC. This is also usually ignored.

Examples of Programs Using the Floating Arithmetic Instructions

Example III: If

$$C(A) = a \quad \text{and} \quad C(B) = b$$

then the following program computes

$$3a - 2b$$

and stores the result in location C:

```
CLA     A ]
FAD     A ]   3a ──> C(AC)
FAD     A ]
FSB     B ]
FSB     B ]   3a - 2b ──> C(C)
STO     C ]
HPR
```

<u>Example IV</u>: If locations A,B,C, and D contain the numbers a,b,c, and d respectively, then the following program computes

$$(a/b)(c/d)$$

and stores it in location R:

```
| CLA |    A ]
| FDP |    B ]    a/b ──> C(R)
| STQ |    R ]
| CLA |    C ]
| FDP |    D ]    c/d ──> C(MQ)
| FMP |    R ]
| STO |    R ]    (a/b)(c/d) ──> C(R)
| HPR |
```

The following equivalent program requiring fewer instructions can also be used:

```
| CLA |    A ]
| FDP |    B ]    a/b ──> C(MQ)
| FMP |    C      (a/b)c ──> C(AC)
| FDP |    D ]
| STQ |    R ]    ((a/b)c)/d ──> C(R)
| HPR |
```

<u>Example V</u>: Suppose that

$C(A3) = a_3$

$C(A2) = a_2$

$C(A1) = a_1$

$C(A0) = a_o$

$C(X) = x$

then the following program evaluates the polynomial

$$a_3x^3 + a_2x^2 + a_1x + a_o = ((a_3x + a_2)\ x + a_1)x + a_o$$

and stores it in location R:

```
LDQ   A3
FMP   X
FAD   A2
STO   R
```

$$a_3x + a_2 \longrightarrow C(R)$$

```
LDQ   R
FMP   X
FAD   A1
STO   R
```

$$(a_3x + a_2)\ x + a_1 \longrightarrow C(R)$$

```
LDQ   R
FMP   X
FAD   A0
STO   R
HPR
```

$$((a_3x + a_2)x + a_1)x + a_o \longrightarrow C(R)$$

Underflow, Overflow, and Division by Zero

If during the course of a floating point calculation an attempt is made to compute a result whose magnitude is too large or too small i.e. lies outside the approximate range

$$(10^{-38} \quad , \quad 10^{+38}) \ ,$$

then a very wrong answer will result!

In this chapter, we have been and will continue to ignore this complication. We only remark that there are two lights on the 704 console, the AC Overflow light and the MQ Overflow light, which are turned on by an overflow (or underflow) in the AC or MQ, and that there are 704 instructions which can be used to determine whether these lights are on.

Also, if division by zero is attempted, the FDP ~~DVP~~ instruction turns on the Divide Check light and goes on to the next instruction leaving C(AC) unchanged.

We will have more to say about these things later.

Some Instructions with only One Operand

The arithmetic instructions that we have just been considering each had two operands. That is, they combined two numbers by an arithmetic process to obtain a result. Now we wish to consider a few instructions which have only one operand.

1. Clear and Subtract

   CLS A $\quad -C(A) \longrightarrow C(AC)$

2. Change Sign

   CHS $\quad -C(AC) \longrightarrow C(AC)$

3. Set Sign Plus

   SSP $\quad +|C(AC)| \longrightarrow C(AC)$

4. Set Sign Minus

   SSM $\quad -|C(AC)| \longrightarrow C(AC)$

Notice that CHS, SSP, and SSM do not have addresses.

Example: If locations A and C contain the numbers a and c, then the following program computes

$$-a/c$$

and stores the result in R

CLS A $\quad -a \longrightarrow C(AC)$

FDP ~~DVP~~ C $\quad -a/c \longrightarrow C(R)$

STQ R

HPR

CHAPTER III

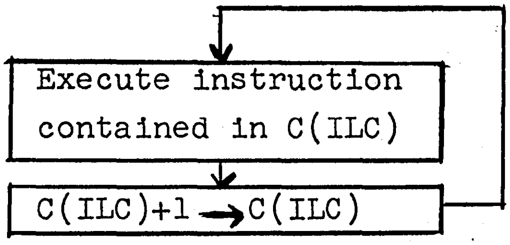
THE CONTROL INSTRUCTIONS

If the 704 could execute only the instructions which we considered in the last chapter, it would not be any more useful than a desk calculator. For if we had only the arithmetic and administrative instructions, there would be no way we could cause the computer to execute the same instructions more than once. Thus for every addition, subtraction, multiplication or division we might wish the computer to perform, we would have to write one or more instructions; and it probably takes more time to write down a 704 instruction than it does to do a multiplication on a desk calculator. In this chapter we will introduce some of the instructions with which we can cause the 704 to profitably execute the same instructions many times and with which we can program the computer to make decisions. We will call them control instructions.

Normal Sequence in Which the 704 Obeys Instructions

Let us first review briefly some basic facts about the computer. Both the numbers with which it computes and the instructions which it executes are stored in the memory. The instructions are stored one to a storage-location. When the computer has just finished executing an instruction in a certain storage location, say location N, it normally proceeds to next execute the instruction in the next storage location, i.e., location N+1.

As a matter of fact, there is a special register in the control mechanism of the 704 which always contains the location of the next instruction to be executed. This register is called the instruction location counter, abbreviated (ILC). Now what we just said about the normal sequence in which instructions are executed can be illustrated by this diagram:



Normal Sequencing

The Control Instructions

Any instruction which can cause the 704 to select some instruction for execution other than the one in the next storage location following the instruction it last executed, that is, any instruction which can change the normal sequence of execution just described, we will call a control instruction. The control instructions are of two types: unconditional control instructions and conditional control instructions

The Transfer Instructions

(1) Transfer

TRA A

is an unconditional control instruction. The next instruction the computer will obey after obeying this one is the instruction in location A. That is, the TRA instruction affects the contents of the instruction location counter. We summarize this as follows:

TRA A $\quad A \longrightarrow C(ILC)$

There are several conditional transfer instructions. Each of these has associated with it a condition which, if satisfied, causes the computer to take the next instruction from a specified storage location. If the condition is not satisfied, the computer takes the next instruction from the next storage location in normal sequence.

(2) Transfer on Minus

TMI A

causes the computer to take its next instruction from location A if the contents of the accumulator is negative and otherwise to execute the next instruction in normal sequence. We can summarize this as follows:

TMI A $\quad$ if (C(AC) negative) then $A \longrightarrow C(ILC)$

Some other conditional transfer instructions are:

(3) Transfer on Plus

TPL A $\quad$ if (C(AC) positive) then $A \longrightarrow C(ILC)$

(4) Transfer on MQ Plus

TQP A $\quad$ if (C(MQ) positive) then $A \longrightarrow C(ILC)$

(5) Transfer on Low MQ

TLQ A $\quad$ if ($C(MQ) < C(AC)$) then $A \longrightarrow C(ILC)$

If the C(AC) = 0 or C(MQ) = 0, the behavior of these conditional transfer instructions is indeterminant.* For the case where C(AC) = 0, we have the following conditional transfer instructions:

(6) Transfer on Zero

TZE A   if (C(AC)= zero) then A → C(ILC)

(7) Transfer on No Zero

TNZ A   if (C(AC) $\neq$ zero) then A → C(ILC)

None of the control instructions affect C(AC) or C(MQ).

Origin Pseudo-Instructions; Symbolic Locations

Up until now we have not worried about where in memory our little example programs were to be stored. We now adopt the convention that writing the operation code ORG with an integer address, say A, at the top of a program means that that program is to be stored in locations A, A+1, ... For example, if we wrote

```
ORG 50      Put this program in 50, etc.
LDQ A
CLA B
STO A
STQ B
HPR
```

* This arises because there are two representations of zero in the 704, +0 and -0, which have zero magnitude but differ in sign. Either zero may be obtained as a result of arithmetic computations. The usual rule of signs holds for results obtained by multiplication or division, but a zero obtained by addition or subtraction has the same sign as the original contents of the accumulator. The reader should also note that the computer considers +0 to be larger than -0 whenever the question arises.

we would mean that our little interchange program was to be stored in locations 50 through 54. The ORG is not a 704 instruction; we call it the origin pseudo-instruction.

Example 1:

Suppose $C(A) = a$

$C(B) = b$

then the following program computes

$\min(a,b)$

and stores it in R:

```
        ORG 50
        LDQ A  |
               |     a → C(R)
        STQ R  |
        CLA B
        TLQ 55 ─┐
        STO R   │    if b < a  then  b → C(R)
        HPR ←───┘
```

Notice that our program is dependent upon where it is stored. It obviously wouldn't work if we changed the origin instruction to ORG 100. The trouble is that the TLQ has an absolute integer for an address. Let us instead use a symbol:

```
        ORG 50
        LDQ A
        STQ R
        CLA B
        TLQ STOP ─┐
        STO R     │
STOP    HPR ←─────┘
```

We call the symbol, STOP, written to left of the HPR a symbolic location, and it serves to indicate that we are letting the symbol, STOP, represent the storage address at which the HPR is stored. With this understanding, our program will work wherever we choose to put it. For example:

```
        ORG 1000
        LDQ A          This program will work!
        STQ R
        CLA B
        TLQ STOP
        STO R
STOP    HPR
 A                     here is a
 B                     here is b
 R                     here will be min(a,b)
```

Here we have indicated that a,b, and the result are to be in the 3 storage locations following the HPR.

Example 2:

Suppose we desire to compute

$$+1.0/C(A) \qquad \text{if } C(A) \neq 0.0$$

and $$+1.0\text{x}10^{38} \qquad \text{if } C(A) = 0.0$$

and store the result in ANSWER

```
CLA A          C(A)→C(AC)
TZE Z          if C(AC)=0, then Z→C(ILC)
CLA ONE ]
FDP A   ]      1.0/C(A)→C(MQ)
```

```
         TRA STOR
     Z   LDQ LARG          10^38 → C(MQ)
  STOR   STQ R             C(MQ) → C(R)
         HPR
   ONE   DEC 1.0              constants
  LARG   DEC 1.0E38
     A                     here is the argument
     R                     here is the result.
```

The Skipping Type Control Instructions

Some of the conditional control instructions do not transfer control to an arbitrarily specified location under certain conditions, but rather they skip one or more instructions under certain conditions. We will introduce one of them here and others will come up during a discussion of input-output and elsewhere.

(8) Compare Accumulator with Storage

```
CAS A       if C(AC) > C(A), go to the next
                             instruction
            if C(AC) = C(A), skip one instruction*
            if C(AC) < C(A), skip two instructions
```

For convenience, we also will introduce this instruction:

(9) No Operation

```
NOP         Do nothing; go to the next instruction
```

The NOP instruction has an address but it is ignored.

* Recall the previous footnote concerning +0 and -0.

Example 3:

Let us write another little program to compute

$$\min (C(A), C(B))$$

and put it in R:

```
          CLA A           C(A)→C(AC)
          CAS B
          CLA B           if C(B)< C(AC) then C(B)→C(AC)
          NOP             they are equal
          STO R           store the minimum
          HPR
      A
                Arguments
      B
      R         Result
```

Other Control Instructions

We shall meet other control instructions in the next chapter on indexing instructions; others will be discussed under miscellaneous topics and one or two may not get discussed at all. For these, consult the 704 Manual.

CHAPTER IV

INDEXING: COUNTING, ADDRESS MODIFICATION

The instructions which we shall consider in this Chapter are called indexing instructions and are extremely useful for coding repetitive computations. They help in two ways: first, they help in address modification, that is, they help in making a sequence of instructions operate on different numbers each time they are executed. Secondly, they help in counting. A typical example of a computation where the indexing instructions are useful would be the formation of the scalar product of two vectors. We will use this as an example later.

Index Registers

The 704 has three registers in its control element each of which is capable of storing any of the integers 0, 1, 2, ..., S-1 where S is the number of storage locations in the memory.* These registers are referred to as index registers 1, 2, and 4; or for short, IR1, IR2, and IR4.

Tag; Effective Address

To see how the index registers can help us in address modification, we must consider tagged instructions. Every 704 instruction may be tagged; and by this we mean that it may have appended to it, as a sort of second address, the

* This number varies from one 704 to another, but is always one of the powers of two: 2^{12}, 2^{13}, 2^{14}, 2^{15}. At present at MIT it is 2^{13} = 8192, but it may be increased later.

number of one of the three index registers. The following instruction is an example of an instruction with a tag of 4:

```
          CLA A,4            C(A-C(IR4))→C(AC)
```

The comment shows symbolically what the effect of the tag is. The contents of the index register with which the instruction is tagged is subtracted from the address of the instruction before the instruction is executed. Thus the instruction acts as if it had an address of

```
          A - C(IR4)
```

This value we call the effective address. All the instructions that we have considered in the last two chapters and which have addresses behave in the same way. Such instructions are called indexable instructions.

It is now easy to see that if we change the contents of an index register, we at the same time change the effective address of every instruction tagged with that index register. This would then provide us with the promised address modification facility.

Decrements

Before we can describe the indexing instructions, it is necessary to explain what a decrement is. Some of the indexing instructions have what amounts to still another address which we write separated from the tag by a comma. For example:

```
          TXI A,2,100
```

Here A is the address, 2 is the tag and 100 is the decrement. In the decrement of such an instruction, we can store any integer that could be stored in an address; that is, any of 0,1,2,...; 32767. The decrement is used to change or test the value contained in an index register, and does not normally refer to a storage location.

When we are writing a program, we will often find it necessary to have constants with integer values in the addresses and decrements. For this we use the plus zero operation code. For example:

PZE 1000, 0, 10

represents a 704 word with 1000 in the address and 10 in the decrement.

The Administrative Instructions for Index Registers

We have described a group of instructions which we called the administrative instructions and which did nothing more than move numbers in and out of the AC and the MQ. Now we are going to discuss some instructions which move integers in and out of the index registers. We will let K stand for any of 1, 2, or 4.

There are two instructions which move integers from storage locations to index registers:

(1) Load Index from Address

LXA A,K

This instruction loads into index register K the integer found in the address of storage location A. We can symbolize this as follows:

C(address of A) → C(IRK)

(2) <u>Load Index from Decrement</u>

```
LXD A,K           C(decrement of A)→C(IRK)
```

There is only one instruction which moves an integer from an index register to a storage location:

(3) <u>Store Index in Decrement</u>

```
SXD A,K           C(IRK)→C(decrement of A)
```

Only the decrement of A is disturbed by this instruction; the rest of the word is unchanged.

There is a similar set of three instructions for moving integers between the accumulator and the index registers:

(4) <u>Place Address in Index</u>

```
PAX 0,K           C(address of AC)→C(IRK)
```

(5) <u>Place Decrement in Index</u>

```
PDX 0,K           C(decrement of AC)→C(IRK)
```

(6) <u>Place Index in Decrement</u>

```
PXD 0,K           Clear the AC; then
                  C(IRK)→C(decrement of AC)
```

This last instruction has the interesting property that if it has no tag (i.e., if K is zero), then it clears the AC; that is:

```
PXD               +0→C(AC)
```

In these three "place" instructions the address is ignored.

It may be well to note that the pair of instructions

```
CLA A             C(A)→C(AC)
PAX 0,1           C(address of AC)→C(IR1)
```

puts the same number into IR1 that this single instruction does:

LXA A,1   C(address of A) → C(IR1)

Counting With the TIX Instruction

We said that the index registers would be a help in counting. To prove our point, we now introduce the most popular indexing instruction which is used to count. It both changes (subtracts from) the contents of the index register and acts as a conditional transfer. Here it is:

(7) Transfer on Index

TIX A,K,N

The action of this instruction depends upon C(IRK). If $C(IRK) > N$, then it decrements IRK by N and transfers control to A. Symbolically: if $C(IRK) > N$, then $C(IRK)-N \rightarrow C(IRK)$ and $A \rightarrow C(ILC)$. However, if $C(IRK) \leq N$, it does not change C(IRK) and it does not transfer control; it goes on to the next instruction in sequence without changing anything.

Example 1:

As an example, suppose we wish to write a program to evaluate the scalar product of two 3-dimensional vectors: $A \cdot B = A_1B_1+A_2B_2+A_3B_3$. Suppose A_1, A_2, A_3 are stored in storage locations VECTA, VECTA+1, VECTA+2, and B_1, B_2, B_3 are stored in VECTB and following locations. Then the following program will compute the scalar product and leave the result in ANSWER:

```
          LXA COUNT,1             +3→C(IR1)
LOOP      LDQ VECTA+3,1
          FMP VECTB+3,1
          FAD ANSWER
          STO ANSWER
          TIX LOOP,1,1            Count to 3
          HPR
COUNT     PZE 3                   a constant
ANSWER                            Originally contains 0.
```

Instructions Executed	IR1	Effective Address
LXA COUNT,1	3	
LDQ VECTA+3,1		VECTA+3-3 = VECTA
FMP VECTB+3,1		VECTB+3-3 = VECTB
FAD ANSWER		
STO ANSWER		
TIX LOOP,1,1	2	(since 3>1, we index and go back)
LDQ VECTA+3,1		VECTA+3-2 = VECTA+1
FMP VECTB+3,1		VECTB+3-2 = VECTB+1
FAD ANSWER		
STO ANSWER		
TIX LOOP,1,1	1	(since 2>1, we index and go back)
LDQ VECTA+3,1		VECTA+3-1 = VECTA+2
FMP VECTB+3,1		VECTB+3-1 = VECTB+2
FAD ANSWER		
STO ANSWER		
TIX LOOP,1,1	1	(since 1 = 1, we do not index; we proceed)
HPR	1	stop

FIGURE IV-1

Figure IV-1 shows a step-by-step history of this program. Notice how the effective address changes with the contents of the index register. The instructions themselves, of course, remain unchanged. This simple program illustrates several points worth remembering about such "TIX loops":

(a) The index register is set to the number of elements to be processed.

(b) The tagged instructions have an address equal to the sum of the location of the first element in the block to which they refer and the number of elements in the block.

(c) The effective address moves forward through the block.

(d) The instructions themselves don't change; only their effective addresses change.

The TIX instruction has a backward twin which acts exactly like the TIX except that it goes to the next instruction where the TIX would transfer and transfers control where the TIX would go to the next instruction:

(8) <u>Transfer on No Index</u>

TNX A,K,N

If $C(IRK) > N$, then $C(IRK) - N \rightarrow C(IRK)$ and go to next instruction
and $C(ILC)+1 \rightarrow C(ILC)$
If $C(IRK) \leq N$, then it leaves IRK alone, and $A \rightarrow C(ILC)$.

<u>Example 2</u>.

We can contrive to use the TNX instruction in our previous example.

```
          LXA  COUNT,1          +4 → C(IR1)
   LOOP   TNX  STOP,1,1←┐       count
          LDQ  VECTA+3,1 │
          FMP  VECTB+3,1 │
          FAD  ANSWER    │
          STO  ANSWER    │
          TRA  LOOP ─────┘
  STOP    HPR
  COUNT   PZE  4
  ANSWER
```

The reader should note that in this example the index register is set to one plus the number of elements to be processed. Another characteristic of this program is that counting and testing are done before the loop is entered.

The above example is somewhat forced; however, TNX does have some valid applications. In more complicated programs the number of times a loop is executed may be a variable computed by the program. If zero is an admissible value for this variable it may be possible to write neater loops using TNX than can be written using TIX.

In such program dependent loops, however, further complications arise in connection with the addresses of tagged instructions. In example 1 we noted that a tagged instruction must have an address equal to the sum of the location of the first element in the block and the number of elements in the block. For a program dependent loop this means that the addresses of all tagged instructions must be altered before execution. The physical modification of 704 instructions is possible (we do not know how to do it yet) but can be avoided in this case by an artiface: namely, instead of assigning a symbol to the location of the first element of the block we assign a symbol to the location of the last element of the block (or better still to this location plus 1).

Thus if we store A_1, A_2, and A_3 in VECTA-3,VECTA-2 VECTA-1, respectively, and we store B_1,B_2, and B_3 in VECTB-3, VECTB-2 and VECTB-1, respectively, then scalar multiplication could be performed by the following variation on example 1:

<u>Example 3</u>

```
              LXA  COUNT,1          +3→C(IR1)
      LOOP    LDQ  VECTA,1
              FMP  VECTB,1
              FAD  ANSWER
              STO  ANSWER
              TIX  LOOP,1,1         Count
              HPR
     COUNT    PZE  3
     ANSWER
```

It is suggested that the reader devise a similar variation on example 2.

There are two other conditional transfer instructions involving index registers:

(9) Transfer on Index High

TXH A,K,N if $(C(IRK) > N)$, then $A \rightarrow C(ILC)$

(10) Transfer on Index Low or Equal

TXL A,K,N if $(C(IRK) \leq N)$, then $A \rightarrow C(ILC)$

These two instructions act exactly like TIX and TNX respectively except that they don't change the index register.

One further unconditional transfer instruction will round out the picture.

(11) Transfer With Index Increased

TXI A,K,N $C(IRK)+N \rightarrow C(IRK)$ and $A \rightarrow C(ILC)$

Some Remarks About Notation

The reader may have noticed that:

a) All the indexing instructions have an X in their 3-letter operation codes.

b) Transmission of information from storage to the index registers is designated by an initial L. (Load). We have LXA and LXD.

c) Transmission of information from the index registers to storage is designated by an initial S. (Store). We have SXD but not SXA.

d) Transmission of information in either direction between the AC and index registers is designated by an initial P. (Place). We have PXD, PDX, PAX, but not PXA.

e) Transfer of control is designated as usual by an initial T. We have four conditional transfers involving index registers, TIX, TNX, TXH, and TXL, and one unconditional transfer, TXI*

* One more, TSX, will be discussed under subject of subroutines.

Arithmetic in the Index Registers and Machine Size

Since the index registers can only hold positive integers less than S, the number of words in the memory; and since S may be less than the largest integer which can be stored in the address or decrement of an instruction, the following pair of instructions might change the integer in the decrement of A:

```
LXD   A, 1
SXD   A, 1
```

In fact our description of the instructions, which move integers into the index registers, was not quite correct. For example, the description of LXD ought to have been:

Load Index from Decrement

LXD A, K $\quad$ C(decrement of A)(mod S) $\longrightarrow$ C(IRK)

where S is the number of words in the memory, sometimes called the machine size, and

X (mod S)

means the remainder obtained after dividing X by S.

Also, our description of the conditional transfer instructions suffer from the same defect. A precise description of TXH would be:

Transfer on Index High

TXH A, K, N $\quad$ if (C(IRK) $>$ N (mod S)),
then A $\rightarrow$ C(ILC)

There is a common convention according to which a negative integer, say -N, written in the address or decrement of an instruction is taken as an abbreviation for the positive integer:

$$2^{15} - N = 32768 - N$$

For the case, N $<$ C(IR1), this can be justified by the following equation:

$$\left[(2^{15} - N) + C(IR1)\right] \text{ (mod S)} = C(IR1) - N$$

which holds because S is always a factor of 2^{15}. It follows that the instruction:

```
TXI   A, 1, -N
```

acts as if it were adding a negative number to index register 1.

The address spectrum for the 8192 word machine is illustrated below:

Binary	Decimal
11.........11	8191 or -1
11.........10	8190 or -2
....	 8191
00.........01	1 or ~~-8190~~ 8192
00.........00	0 or ~~-8191~~

For example, if C(IR1) > 1, then the instruction

```
        TXI   A, 1, -1
```

acts exactly as the following instruction would

```
        TIX   A, 1, 1
```

Our convention for negative numbers leads us to a rather peculiar algebra, however, in connection with the TXH and TXL instructions. The usual algebra applies when we are comparing two numbers of like sign, however, if numbers of unlike sign are being compared we see from the table that negative numbers are frequently larger than positive numbers.

Thus, for example, the instructions:

```
        TXL   A, 1, -1  and TNX   A, 1, -1
```

would be unconditional control instructions.

Example 4. We rewrite example 1 to illustrate these points.

```
         LXA  COUNT, 1
LOOP     LDQ  VECTA +3, 1
         FMP  VECTA +3, 1
         FAD  ANSWER
         STO  ANSWER
         TXI  TEST, 1, -1       Decrease C(IR1) by 1
TEST     TXH  LOOP, 1, 0
         HPR
COUNT    PZE  3
ANSWER
```

Example 5. We again rewrite example 1 to illustrate a powerful technique for writing program dependent loops.

```
        LXA   COUNT, 1
LOOP    LDQ   VECTA, 1
        FMP   VECTB, 1
        FAD   ANSWER
        STO   ANSWER
        TXI   TEST, 1, -1           1   2
TEST    TXH   LOOP, 1, -3
        HPR
COUNT   PZE   0
ANSWER
```

The next example may take some study, but it illustrates the fact that coding loops in terms of the TXI and TXL or TXH instructions may be more convenient than using the TIX.

Example 6. Suppose we have 10 numbers stored in locations A + 1, A + 2, ..., A + 10. Then the following rather complicated routine will sort these numbers in order of increasing size. It does it by the so-called interchange method. First it scans through the list interchanging adjacent numbers if they are out of order. When it gets to the end of the list, the largest element is in last place. Then it repeats the process for the other 9 numbers, etc.

```
        LXA   CONST, 2          Set count of no. of passes
Pass    LXD   SKIP, 1           Consider 1st pair
        SXD   TEST, 2           Set test for end of pass.
NEXT    LDQ   A, 1
        CLA   A + 1, 1
        TLQ   SKIP              is the pair out-of-order?
        STO   A, 1              yes, interchange them
        STQ   A + 1, 1
SKIP    TXI   SKIP + 1, 1, -1   Consider next pair
TEST    TXH   NEXT, 1, -        go to NEXT if not end of pass
Q       TXI   Q + 1, 2, 1       Prepare for next pass
        TXL   PASS, 2, -2       Go back for next pass, or
        HPR                     stop
CONST   PZE   - 10              Address has 2^15-10 = 32758
```

Some Pathological Points about the Index Registers

Any of the indexing instructions may be written without a tag. If this occurs the instruction behaves as if there were an imaginary index register, numbered 0 by convention, whose contents is always 0. An application for

PXD 0,0

has already been described. Another useful case is the instruction

TXL A,0

which now becomes an unconditional control instruction and can almost always be used in place of TRA. The advantage in doing so is that TXL can have a decrement and decrements are useful for storing integers needed in the program. (What is the point in ever using TRA?)

An indexable instruction may also (in a certain sense) refer to more than a single index register. For an explanation of this the reader is referred to p11 of the 704 manual and to the definition of SXD as given on p26. Such multiple reference is indeed tricky and must be done with care. The following two examples illustrate some uses for reference to multiple index registers. The reader may, if he wishes, defer study of these examples until he has a better feel for the binary nature of the machine.

Example 8. Here we multiply two n by n matrices

$A = (a_{ij}) \qquad i, j = 1,\ldots,n$

$B = (b_{ij}) \qquad i, j = 1,\ldots,n$

to obtain the matrix

$C = (c_{ij}) \qquad i, j = 1,\ldots,n$

We assume that the matrices are stored in "row by row" form, i.e.

a_{ij} is in register MATA + (i-1)n + (j-1)

b_{ij} is in register MATB + (i-1)n + (j-1)

c_{ij} appeares in register MATC + (i-1)n + (j-1)

The rule for matrix multiplication is

$$c_{ij} = \sum_{1}^{n}{}_{k}\, a_{ik}b_{kj}$$

The program follows:

```
            LXA   SETUP, 7            n^2→C(IR1), C(IR2), C(IR4)
   1 LOOP   PXD
            STO   MATC+N*N,4          N*N means N^2
   LOOP     LDQ   MATA+N*N,1  ]
            FMP   MATB+N*N,2  ]
            FAD   MATC+N*N,4  ]       Form C_ij
            STO   MATC+N*N,4  ]
            TXI   NEXT,1,-1   ]
   NEXT     TIX   LOOP,2,N    ]
            TNX   STOP,4,1    ]       Stop when finished
            TNX   2 LOOP,2,1  ]       Count within row
            TXI   1 NEXT,1,N     ]    Same row
   1 NEXT   TXI   1 LOOP,2,N*N-N ]
   2 LOOP   TXI   1 LOOP,2,N*N-1 ]    New Row
   STOP     HPR
   SETUP    PZE   N*N
```

An interesting exercise for the reader would be to extend this program to handle arbitrary conformable matrices.

Example 9. Here we sort n numbers

$$a_0, a_1, \ldots, a_{n-1}$$

which are stored in registers DATA, DATA +1,..., DATA +n-1 respectively into ascending order. We do this by a variation of the interchange method as follows: First we compare C(DATA) with C(DATA+1), C(DATA+2), etc. and place the smallest number in DATA. Next we compare C(DATA+1) with C(DATA+2), C(DATA+3) etc., and place the next smallest number in DATA +1. We repeat the process n-2 more times and sort the numbers.

```
            LXA   COUNT, 3
   LOOP     CLA   DATA +n,2   ]      Exchange if necessary
            LDQ   DATA +n,1   ]
            TLQ   SKIP        ]
            STQ   DATA +n,2   ]
            STO   DATA +n,1   ]
   SKIP     TXI   NEXT, 2, -1 ]      Count one pass through data
   NEXT     TXH   LOOP, 2, 0  ]      data
```

```
         TIX  LOOP, 3, 1      Count passes.
         HPR
COUNT    PZE  n
```

CHAPTER V

The SHARE Assembly Program (SAP).

This introduction contains enough information to enable one to write programs in the SAP language that will be correctly translated by SAP. Not all of the features of SAP are described since some of them are useful mainly to the experienced programmer, but the more important features are described more fully than in the main writeup.

It is assumed that the reader has been introduced tc most of the 704 instructions and also to the basic idea of coding with symbolic addresses.

The Purpose of SAP

The purpose of the SHARE assembly program is to translate programs written in the SHARE symbolic language to a binary form which can be obeyed by the IBM 704. This symbolic language is standard for 704's throughout the country, and programs exchanged between 704 computing centers will usually be in this form.

SAP was written by Roy Nutt of the United Aircraft Corporation to the specifications of the SHARE organization and became available around the beginning of 1956. A revised version has been written at United Aircraft and will replace the original version although all programs written for the original will be correctly translated by the new version. A complete description of the new program is not yet available so this introduction is mainly based on the older version which contains all the most important features of the language. We shall mention some features of the new SAP, and when we do so will indicate that they belong to the new SAP.

What SAP Does

SAP takes a program written in the language to be described and punched onto cards and does the following things to it:

1. Starting at a register specified by the programmer in the program it assigns numerical addresses to the symbolic addresses written by the programmer.

2. It translates the mnemonic operation codes (like CLA) written by the programmer into the binary code which can be obeyed by the 704.

3. It translates numbers written in decimal form by the programmer into binary fixed or floating point numbers.

4. It incorporates into the program routines taken from the library tape.

5. It does not run the program it translates.

6. It punches a deck of absolute binary cards with 22 instructions per card. (The original language is written one instruction per card.)

This deck can be loaded into the 704 if prefixed with a loading program (furnished by the machine operator). The last binary card punched by SAP gives the address of the first instruction to be obeyed in the program. There are other optional forms of output which we will not discuss in this introduction.

7. It provides an assembly listing which describes the translation performed in printed form. This listing also tells about any mistakes in your program which SAP has detected.

8. It also punches a symbol table giving the numbers assigned to symbols. This is useful for making additions to the program.

How to Use SAP

1. Write your program in the SAP language and punch it on cards. The deck of cards produced is called the symbolic deck.

2. Give the symbolic deck to the scheduler with a performance request form asking for SAP assembly.

You will get back the binary deck, the assembly listing and the symbol table.

3. Examine the listing to see if SAP has found any errors in your program. If there are errors correct them in ways to be described later. This may or may not require a new assembly.

4. Give the corrected binary deck to the scheduler with a performance request asking that the problem be run.

5. Look at the answers.

(When the MIT operator program is available there will be some changes in this procedure.)

The SAP Language*

In order to be translatable by SAP a program must be punched on cards in a particular form. To facilitate punching

* This language is not used in the 704 manual which was written before SAP. In the back of the manual another assembly program (NYAP1) is described which is not in general use.

it is usual to write the program on a "SHARE symbolic coding form", pads of which are available in the computing center.

After cards have been punched a printed listing may be made on the 407 accounting machine. This is useful for checking the key punching.

Each line of the coding form is punched onto a single card and represents either 704 words or an instruction to SAP on how to make the assembly.

Each time a key on the punch is struck certain holes are punched in one of the 80 columns of the card and the punch spaces to the next column. The set of columns of the card is divided into fields for the purposes of this language. Columns 1 to 6 make up the location field (column 1 has an additional special significance). Columns 7 and 11 are not used. Columns 12 to 72 make up the variable field. Finally, columns 73 to 80 make up the identification field which has no programming significance. Columns 8 to 10 make up the operation field.

The way a card is interpreted by the compiler is determined by the 3 letter operation field. Now we describe the meaning of the various 3 letter codes in the operation field.

REM (Remarks)

If the operation code is REM the assembly program ignores the card except that the contents of the variable field of such a card appears in the assembly listing. It is a good idea to put an REM card with one's name on it at the beginning of the symbolic deck so that the operator will deliver the listing to the right person. Otherwise REM cards are used to label sections of program and should be used liberally as aids to one's memory in re-reading the program. The letters REM themselves are suppressed in the assembly listing.

ORG (Origin)

This card is used at the beginning of every program to specify the register at which the program begins. This number must be large emough so that there is room at the beginning of memory for the loading program which brings the binary cards into the machine. The space required varies from about 30 registers to several hundred depending on the feature of the loader but if the programmer is not pressed for memory space 512 = 1000_8 is a good register to start in. The earlier registers can be used for intermediate results since it does not matter if they overlay the loader after it has served its purpose.

<u>The address at which the program is to start is written (as a decimal integer) starting in column 12</u>.

An ORG card can also occur in the middle of the program. In this case one writes an <u>expression</u> starting in column 12 each <u>symbol</u> of which must have previously been <u>defined</u>. ($T_h$is last sentence will be clear when we describe what symbols and expressions are and what it means for the symbols of an expression to be defined.)

<u>704 Instructions</u>

a. <u>Operation field</u>: A 704 instruction is written with the 3 letter SHARE nmemonic code (e.g. CLA) in the operation field. The 3 letter SHARE codes include those of the 704 manual but some additional ways are provided for writing input-output selection and sense instructions that place less burden on the memory of the programmer.

b. <u>Location field:</u> If the instruction is not referred to by other instructions the location field should be left blank to save assmebly time. If the instruction is to be referred to a <u>symbol</u> should be written in the location field.

<u>Symbols</u>: A symbol may be any combination of 6 or fewer Hollerith characters none of which are the special characters:

+ - * / , $

and not all of which are digits. The Hollerith characters are: the capital letters, the digits 0 thru 9, and the following special characters:

+ - * / , $. ( ) =

There are two - signs on the keypunch, and the one on the key also marked SKIP (which gives rise to an 11 punch) is used in SAP. The one on the same key with the = sign is not used.

The above list does not agree with that given in the 704 manual which gives the characters used for commercial purposes.

Examples of <u>legal</u> symbols are A, AB, COMMON, START, DONE, 3.4, A40. The symbols ELEPHANT, A+B, 17, are illegal. If the symbol has fewer than 6 characters it may be placed anywhere in the location field.

c. <u>The variable field</u>: If the instruction has an address, tag, and decrement these are written in that order starting in column 12 and separated by commas with no intervening blanks. Examples of the way 704 instructions are written are the following:

```
B      CLA  A
AB     CLA  A,1
4.1    CLA  382
       TIX  B,1,1
       PAX  0,1
       TXL  A
       TXL  A,1,1
       TXL  A,0,1
       HPR
```

Notice that if the instruction has no decrement part nothing need be written for it. This is the reason why SAP instructions are written with the address, tag and decrement in a different order from the order of these parts in a 704 word. The first blank after column 12 signals the end of the instruction to SAP. Anything beyond this blank is ignored in the translation but is reproduced verbatim in the listing so it is usual to put comments about the instruction after the blank. It is recommended that the program be liberally sprinkled with comments.

Expressions

In the address, tag, and decrement fields one can write not only integers and symbols, but certain arithmetic combinations of integers and symbols called expressions. In these expressions the characters +, -,*, and / stand for addition, subtraction, multiplication and division respectively. Examples of expressions are A+3, A+B, A+B* C, 391AB+A* B, -A and A*B + A*C + A*D. No parentheses are allowed. The arithmetic involved is integer modulo 2^{15} so that, for example, -1 is the same as 32767 and 32769 is the same as 1. This arithmetic is more fully described in the SAP writeup but the information in the above paragraph should be sufficient for most purposes.

It is important to note that the arithmetic involved has to do with the addresses and not with their contents (in contrast with FORTRAN). Thus if register 900 is assigned to the symbol A, the expression A+9 refers to register 909.

The new SAP provides the additional feature that the symbol * can be used to refer to the location of the current instruction. Thus TXI*+ 1, 1, -1 refers to a TXI to the register following the TXI instruction. It turns out that no ambiguity results from using the same character for a multiplication sign and also to refer to the current location.

Data Storage

If one wants to reserve a register for an intermediate or final result, one merely writes a symbol to name the register in the location field and leaves the operation and variable fields blank.

END (End of Program)

The last card in any program has END as its operation part. This is very important because if the END card is omitted your assembly may be mixed up with the next man's and he is unlikely to be grateful.

The variable field of an END card contains an expression which tells the assembly program the address from which the first instruction is to be taken, when the program is run. SAP uses this information to punch as the last card in your program a transfer card which tells the loader the register to which it should transfer control when it has finished loading the program into memory.

Data Numbers

If a program is to be run with only one set of data or if certain data are to be included with all runs of a program it is simplest to assemble the data into the program. SAP is capable of converting various kinds of data into the binary form used by the 704.

DEC (Decimal Data)

This pseudo operation causes the decimal numbers in the variable field to be converted to binary and stored in registers starting at the point in the program where the DEC card appears. The first data word may be referred to by putting a symbol in the location field of the card.

The numbers to be converted are written starting in column 12 separated by commas and with no blanks. Whether the conversion is to fixed or floating binary depends on the way the numbers are written. We now describe these ways:

a. Integers: If a positive or negative decimal integer is written with no decimal point, it is converted to a binary integer. The + sign is optional with positive integers. Of course, the absolute value of the integer must be less than 2^{35}.

Examples of decimal cards with integers are:

```
A   DEC 17
B   DEC 23178195, -251, +251, 48
```

The number of integers that can be put on a card depends on their size but they must not extend beyond column 72.

b. Floating Point Numbers: If a decimal number like 3.481, -2.0 or even -2. is written, it is converted to floating binary.

`. Numbers can also be written in a floating decimal notation: 3.4E10 is equivalent to $3.4x10^{10}$ and -3.4E-10 is equivalent to $-3.4x10^{-10}$. These numbers are converted to floating binary and must of course be of a size that allows them to be represented as normalized floating binary numbers.

c. There is another notation for producing fixed point numbers which is described in the main SAP writup. The first blank (as usual) indicates that all punching to the right is a remark.

OCT (Octal Data)

For some purposes, in particular for describing masks to be used in logical operations, it may be convenient for the programmer to think of his data in binary form. However, since it is hard to copy even 36 consecutive binary bits without error it is customary to write binary data in octal. (Each 3 bits are summarized into one octal digit.)

If OCT appears in the operation field then the octal numbers written on the card starting in column 12 and separated by commas are converted to binary integers. If the word has 12 digits the first bit may be regarded either as a sign or as part of the leftmost digit so that -0 = 4, -1 = 5, -2 = 6 and -3 = 7. Either form may be used.

The first blank to the right of column 12 indicates that all punching to the right is to be considered a remark.

BCD: (Hollerith Data or Binary-Coded-Decimal)

The 60 columns, 13-72, are regarded as consisting of 10 six-character words. If column 12 contains a digit V ($0\leq V\leq 9$), the first V words are converted to binary and assigned to V successive registers. If column 12 is blank or contains some other character, 10 words are converted. The words converted may contain blanks.

Data Having the Format of an Instruction

It is often desirable to include in a program words for which the address, tag, decrement, and prefix are described. (The prefix is the first three bits.) This is done as follows: the address, tag, and decrement are written as in 704 instructions and may be expressions. If the operation field is left blank the prefix will be zero. To give it another value one writes one of the following operation codes.

Code	Name	First 3 bits
MZE	Minus zero	100 or -00
MON	Minus one	101 or -01
MTW	Minus two	110 or -10
MTH	Minus three	111 or -11
PZE	Plus zero	000 or +00
PON	Plus one	001 or +01
PTW	Plus two	010 or +10
PTH	Plus three	011 or +11
FOR	Four	100 or -00
FVE	Five	101 or -01
SIX	Six	110 or -10
SVN	Seven	111 or -11

BSS (Block Started by Symbol) and BES (Block Ended by Symbol

It is frequently necessary to reserve a block of consecutive storage registers for an array of intermediate or final results. This can be done by either the BSS or BES operation with an address giving the number of registers to be reserved. The difference is that if the pseudo-operation is BSS, the symbol in the location field refers to the first of the registers reserved while if BES is the pseudo-operation, the symbol refers to the register right after the last of the reserved block.

For example, if we have a program beginning

```
          ORG   1000
COMMON    BSS   60
START     CLA   COMMON+5
```

the symbol COMMON is assigned the value 1000 and START is assigned the value 1060.

On the other hand in the program

```
            ORG 1000
    COMMON  BES 60
    START   CLA COMMON-60
```

both COMMON and START are assigned the value 1060.

When the program is loaded nothing is stored in the block of registers reserved by BSS or BES. Thus these registers will contain whatever was in them previously.

If we write, BSS 0, a symbol is assigned but no register is reserved.

One can write symbolic expressions as well as constants in the variable field of a BSS or BES instruction. The symbols occurring in such an expression must have previously been defined. (We discuss this in connection with the next section.)

SYN (Synonym) and EQU (Equals)

A symbol (placed in the location field) may be assigned the value of an expression (placed in the variable field) by using either of the pseudo-operations SYN or EQU. The symbols occurring in the expression must have been previously defined.

Thus

```
    N  EQU  20
```

assigns N the value 20 and

```
       ORG  1000
    A  BSS  0
    B  SYN  A
```

assigns B the value 1000.

The distinction between SYN and EQU is somewhat difficult to describe. It arises only when a coder requests SAP to punch out a very special type of binary card (called relocatable binary) which is described in the SAP write-up included as an appendix. The casual coder need not concern himself with this distinction.

When A Symbol Is Defined

In order to understand this question it is necessary to know something about how the assembly program works. It goes over the program twice. The first time is for the single purpose of assigning values to symbols. It sets a location counter equal to the value of the first ORG card and increases the counter by one for each instruction read. Every time it encounters a symbol in the location field of an instruction it makes an entry in the symbol table which assigns the current value of the location counter to the symbol. Naturally, the location is increased for DEC, OCT, and BCD cards by an amount equal to the number of words on the card and for BSS and BES cards by the value of the expression in the variable field. A SYN or EQU pseudo-operation also causes an entry to be made in the symbol table. In order for this to be possible the symbols in the expression in the variable field must already be in the symbol table, or, to use the customary terminology, must have been previously defined. This will be the case only if the symbol has previously appeared in a location field. The symbols appearing in the variable field expression of an ORG, BES, or BSS pseudo-operation must have been previously defined so the assembler can change the location counter properly. If such a symbol is undefined when the assembler encounters it, it is taken to have the value zero although if it is later defined subsequent uses of it will have the correct value. In the second pass over the program the assembler evaluates the expressions occurring in the variable field of 704 instructions translates the operation codes, punches the binary cards and prints the listing.

LIB (Library Search)

A list of the subroutines which are on the library tape is available in the computing center. To incorporate one of these subroutines it is only necessary to include a card in the program with the operation code LIB and the name of the subroutine in the location field. This name must be written and placed in the location field exactly as it appears in the list of available routines.

The routine will appear in the program wherever the LIB card occurs.

Additional Pseudo-Operations

The above are not all the pseudo-operations available with SAP. However, the programmer who has had some experience with the use of those so far defined ~~he~~ will probably prefer the condensed style of the regular writeup to the wordiness of this introduction.

Error Detection and Correction

The assembly program generally detects certain errors and misprints. First of all it prints at the head of each program any symbol that is defined more than ONCE ~~one.~~ Secondly, whenever a symbol or operation code is undefined it leaves a blank in the octal translation. Finally it prints at the end of the program a list of the undefined symbols. The new SAP has additional error detection features.

There are four main ways of correcting errors detected at assembly time. Which one should be used depends on the extent of the error.

1. Replace incorrect decimal cards and re-assemble. If the errors are very extensive this is desirable.

2. Make binary correction cards and include them before the transfer card when the program is loaded. Method 3 is preferred to this.

3. Make octal correction cards as follows:

The octal equivalents of the symbolic addresses in the SAP language program can be obtained from the assembly listing. The octal correction cards are placed after the program to be corrected and before the transfer card. The loader first loads the uncorrected program and then puts the numbers from the correction cards in the registers specified, replacing the erroneous instructions. A correction card may have up to four corrections on it which are punched in the following format:

Columns	
1-5	location where first correction is to be stored
6-17	corrected word (12 octal digits)
19-23	location of second correction (if any)
24-35	second correction (if any)
37-41	location of third correction (if any)
42-53	third correction (if any)
55-59	location of fourth correction (if any)
60-71	fourth correction (if any)

Leave all other columns blank except that 73-80 are ignored. If there are fewer than four corrections, fields can be left blank except that the first field cannot be left blank or else the whole card will be ignored by the loader.

The loader which loads absolute binary cards and octal correction cards is NYBOL1. It is described in SHARE distribution 215.

4. It is also possible to assemble corrections by using the symbol table of the original assembly.

CHAPTER VI

THE MIT POST-MORTEM PROGRAM

Program Debugging

The first time that a program is tried on the computer, it will probably fail. It can fail in any of the following ways:

(1) It can stop on a computer alarm.

(2) It can stop on an improper halt instruction (i.e. not the halt instruction on which the coder planned to stop).

(3) It can stop on the proper halt instruction and yield wrong answers.

Cases (1) and (2) can be recognized by the computer operator and he will (as a matter of course) copy whatever information about the failure can be obtained from the console lights. This will include the location of the instruction on which the machine stopped; the contents of the AC, MQ and index registers; and the condition of the various alarm lights. In some cases the error can be deduced from this information.

If the coder cannot deduce his error from the console lights he must then resort to more powerful diagnostic techniques. Perhaps the simplest of these is the post-mortem program. A post-mortem program prints the contents of specified storage registers after the program failure has occured. By examining such records the coder can almost always find his errors. The coder is urged to restrict his post-mortem requirements whenever possible since computer time is used to produce the post-mortem results.

In certain cases however, the coder may not be able to deduce his error from post-mortems obtained after the failure. He may then have to modify his program to store certain critical results (which he can obtain via post-mortems after failure) or insert instructions (called blocking instructions) at intermediate points in his program (so that he can obtain intermediate results via post-mortems). The clever coder will try to anticipate errors and will build such features into his program when he writes it originally. Such error anticipation features can be removed when the program is finally debugged.

For the latter (most difficult) case the coder also has available a class of programs (called tracing programs) which can be used to print out quantities (e.g. contents of storage registers, C(AC), etc.) during the time that his program is operating. Such programs may be selective (i.e. print out information only at selected instructions in the program called break points) or non-selective (i.e. print out information for every instruction executed). Tracing programs (particularly non-selective ones) use a lot of computer time and should be used only as a last resort. Several tracing programs have been written by members of SHARE and are described in the SHARE distribution material.

A final word of warning: too much information can be almost as bad as too little information. The coder is urged to be selective in obtaining data for diagnostic purposes.

The MIT Post-Mortem Program

The MIT post-mortem program allows the coder to print arbitrary ranges of storage in specified forms. The forms allowed as output are exactly those forms allowed as input in SAP language (i.e. instructions, floating-point numbers, fixed-point numbers, integers, octal numbers and BCD).

For each range of storage required the coder must prepare a request card. The deck of request cards should be prepared before the program is run and submitted to the operator along with the main program deck. The coder should tell the operator on the performance request that a post-mortem deck is included since the operator must set up certain magnetic tape units which are used by the post-mortem program.

The output of the post-mortem program looks exactly like SAP language. Thus if a coder understands SAP language he understands post-mortem results.

The coder may request that post-mortem results be recorded directly on the printer, directly on the punch or on a magnetic tape unit for printing or punching later on. (We shall have more to say about such off-line operation). In all cases the output obtained is the same (i.e. if punched output is printed on an accounting machine it looks exactly like output printed directly).

Request Cards

Every memory range requires a request card which has the same format as a SAP card and is identified by the letters, PMR (Post-Mortem Request), in the operation field. The ~~location~~ VARIABLE field must contain four expressions separated by commas with no intervening blank columns.

The first two expressions define the initial and final addresses of the range in memory to be recorded. Any legal SAP expressions are allowed here and symbols may be used if required.

The last two expressions designate the mode in which words are to be recorded (instructions, etc.) and the output device to be used (printer, etc.). These are designated by certain mnemonic 3 letter abbreviations defined in an appendix:

Some examples of request cards are:

```
PMR 150, 200, FLO, NPR
```

which means "record the contents of registers 150 to 200 (inclusive) as floating-point numbers on the printer which is directly connected to the computer" and:

```
PMR A1, B1 + 5, SYM, NPU
```

which means "record the contents of registers A1 to B1 + 5 as instructions with symbolic addresses on the punch which is directly connected to the computer.

Any characters punched in the variable field following the terminating blank column are not considered part of the request but are instead recorded as a remark preceding the request. In addition, special request cards (identified by the letters, REM, in their operation field) may be used to associate remarks (punched in their variable field) with PMR requests. The REM card must immediately precede the PMR card.

The end of a request deck is signaled by a termination card. A suitable termination card would be one having the 704 instruction, HTR, punched in its operation field. Some others are mentioned in the appendix.

An example of a complete post-mortem request deck is the following:

```
REM  THIS BELONGS TO JONES
PMR  A1, A1 +5, FLO, NPR FIRST REQUEST
PMR  100, C1, FLO, NPR SECOND REQUEST
HTR
```

Symbolic Requests

The coder may use symbols in specifying core memory ranges and he may also request that the words in a range be recorded as instructions with symbolic addresses. In either case a symbol table must be made available to the post-mortem program. This can be done by preceding the request deck with the binary symbol table produced by SAP during its assembly of the main program deck. Since both the symbol table and the request deck are read into the computer by the post-mortem program they should not be preceded by a binary loader.

The Machine Conditions

The post-mortem will record the contents of the various registers and alarm lights in the arithmetic element as remarks preceding the first post-mortem request. The program cannot however record C(MQ) and C(ILC) since these and storage registers 0-4 are used in loading the post-mortem program itself.

CHAPTER VII

SUBROUTINES

It often happens in a large program that a particular computation must be done many times. For example, the square root function or sin function must be evaluated several times or the scalar product of two vectors must be calculated repeatedly. In such cases it is clearly desirable to program and code the routine for such a computation only once. It is also usually but not always advantageous to have the instructions which perform the computation stored in the memory only once. Such a group of instructions which are used repeatedly by a given program or perhaps by any particular program only once but in many distinct programs, is called a subroutine. In fact a skilled programmer very often designs his program so that it is built out of subroutines by breaking the problem up into units which occur repeatedly or *may in subsequent revisions of the program be used repeatedly*.

Closed Subroutines

As a part of a larger program we may wish to tabulate the following function

$$f(x) = \sin x + \cos (\sin x)$$

for a set of values of the independent variable, x. If somehow we have written programs for evaluating sin x and cos x we can evaluate f(x) by the following sequence of steps.

1. Compute $z = \sin x$
2. Compute $y = \cos z$
3. Compute $f(x) = z + y$

The functions, sin x and cos x, are required only once in this sequence.

A slightly different function, say

$$g(x) = \sin x + \sin(\cos x)$$

would, however, give rise to the sequence

1. Compute $z = \sin x$
2. Compute $y = \cos x$
3. Compute $w = \sin y$
4. Compute $g(x) = z + w$

in which the function, sin x, is required at two different places.

Since programs for evaluating sin x and sin y will differ only slightly we can reduce our storage requirements by writing one program to do both jobs. Whenever we wish to compute a sine we will transfer control to this program. We must, however, convey two pieces of information to the sine program, namely

1. The location of the argument whose sine is to be evaluated.
2. The location in the main program to which control should be returned once the sine has been evaluated.

This can be done in many ways, the most convenient of which uses the following indexing instruction:

TSX x, t : Transfer and Set Index

The instruction, TSX x, t, copies -C(ILC) into index register t* and then transfers control to register x.

TSX x, t : $-C(ILC) \rightarrow C(IRt)$, $x \rightarrow C(ILC)$.

Let us suppose now that we have written a program (beginning in register SIN) which forms sin C(AC)

SIN $\sin C(AC) \longrightarrow C(AC)$

* More precisely, TSX copies 32768 - C(ILC) modulo machine size into index register t.

If we agree to enter this program by the instruction

```
MAIN  TSX   SIN,4
```

then we can make it a closed subroutine merely by terminating the program with the instruction

```
TRA 1,4
```

which returns control to register MAIN +1 with the required sine in the AC.

The program for computing sin x is called a closed subroutine since it is stored separate from the main sequence of control, but is entered from the main sequence and returns to the main sequence. The argument of the subroutine (C(AC) in this case) differs from use to use of the subroutine and is called a <u>program parameter</u>.

The program parameter and the TSX instruction required to link the subroutine to the main program are referred to as a <u>calling sequence</u> for the subroutine.

<u>Closed Subroutines having Many Program Parameters</u>

If a closed subroutine has only one program parameter it can be stored in the AC (or even the MQ). If a closed subroutine has more than two program parameters the AC and MQ no longer suffice. The most convenient solution in this case is to store the program parameters in the registers following the TSX instructions used to enter the subroutine. Within the subroutine we can refer to these registers by instructions of the form:

```
CLA  1,4
CLA  2,4 etc.
```

Before giving an example let us introduce two new 704 instructions which will be useful in interpreting calling sequences.

The reader is already aware of an indexing instruction, PDX, which copies the decrement of the AC into an index register. There also exist 704 instructions which copy the address and decrement of the AC to storage registers, namely

```
STA   x  : Store address
STD   x  : Store decrement
```

The instruction, STA x, copies the address of the AC into the address of storage register x.

STA x : C(address of AC) → C(address of x)

The instruction, STD x, copies the decrement of the AC into the decrement of storage register x.

STD x : C(decrement of AC) → C(decrement of x)

The reader should note that STA (STD) affects only the address (decrement) of x and does not disturb any other part of C(x) or C(AC).

Thus, for example, the sequence

```
PXD  0,4
STD  X
```

has the same effect on C(X) as the single instruction

```
SXD  X,4
```

An example of a subroutine requiring more than one program parameter, is the following subroutine which forms the scalar product of two vectors. The program parameters required are the location of the vectors in question and the number of components. The result is stored in the AC. It is most convenient to identify the location of the vectors by specifying the addresses of the registers immediately following the last component of each vector.

This routine can be entered by the calling sequence

```
        SXD    TEMP, 4    Save C(IR4)
        TSX    ENTRY,4    Enter Subroutine
        PZE    A1        ⎫Addresses of vectors
        PZE    A2        ⎭
        PZE    N          Number of components.
        LXD    TEMP,4     The subroutine returns control
                          to here and we restore C(IR4)
```

The program parameters (A1, A2 and N) are stored in the addresses of the registers immediately following the TSX instruction.

The subroutine itself is given below.

```
        ENTRY    CLA    1,4     Set addresses of vectors.
                 STA    LDQ
                 CLA    2,4
                 STA    FMP
                 SXD    SAVE,1  Save C(IR1)
                 CLA    3,4     Set to count in IR1
                 PAX    0,1
                 PXD            Set C(PROD) to zero
                 STO    PROD
          LDQ    LDQ    0,1     Form scalar product.
          FMP    FMP    0,1
                 FAD    PROD
                 STO    PROD
                 TIX    LDQ,1,1
                 LXD    SAVE,1  Restore C(IR1)
                 TRA    4,4     Return to program - 4th word after TSX
         PROD    PZE
         SAVE    PZE
```

An Example of a Subroutine Using an Interlude

It often occurs in practical routines that some program parameters are changed infrequently while others differ for every entry to the subroutine. For example, in the scalar product subroutine it is usually true that the number of components changes infrequently while at least one vector location changes with every reference to the subroutine. In this case, it is wasteful to include the final program parameter in the calling sequence every time the routine is entered.

Situations of this type are often handled by interludes. For example the calling sequence would be

```
      SXD   TEMP,4      Save C(IR4)
      TSX   ORDER,4     Enter interlude
      PZE   A1          Addresses of vectors.
      PZE   A2
      LXD   TEMP,4      Return here and restore C(IR4)
```

At ORDER would be stored the interlude

```
ORDER TRA   ENTRY       Enter subroutine
      PZE   N           Number of components
```

The subroutine would then be modified to

```
ENTRY SXD   SAVE,1      Save C(IR1)
      CLA   0,4         Obtain number of components
      STA   CLA         from interlude and set to count
      LXA   SAVE,1
  CLA CLA   0,1
      PAX   0,1
      CLA   1,4         Set addresses of vectors.
      STA   LDQ
      CLA   2,4
      STA   FMP
      PXD               Set C(PROD) to zero
      STO   PROD
```

```
LDQ    LDQ    0,1        Form scalar product
FMP    FMP    0,1
       FAD    PROD
       STO    PROD
       TIX    LDQ,1,1
       LXD    SAVE,1     Restore C(IR1)
       TRA    3,4        Return to program.
PROD   PZE
SAVE   PZE    -1
```

The SHARE Library of Subroutines

The nicest feature about using subroutines is that they may have been written (and debugged) by someone else. Having this in mind every computer installation begins immediately to assemble a collection of pre-tested programs into a subroutine library. Among 704 owners the SHARE organization serves as a collection and distribution agency for 704 subroutines.

A very important feature of a library subroutine is a detailed write-up describing exactly how to use it. Also very important to the library is the adoption of a certain set of conventions to prevent coders from working at cross-purposes. There follows a partial list of SHARE subroutine conventions which has been abstracted from the SHARE reference manual:

1. Subroutines shall always be entered by a calling sequence using IR4 as linkage.

2. The point transferred to shall always be the first instruction in the subroutine.

3. Index registers and sense lights when used by a subroutine shall always be restored to their original contents within the subroutine before exiting.

4. The six letter symbol, COMMON, is reserved for subroutines to represent temporary storage (i.e. a block

of registers used during operation of the subroutine but whose initial values do not matter to the subroutine.)

5. All other symbols used in the subroutine normally contain 5 or fewer characters (so that they can be headed).

Conventions having been established the SHARE organization has proceeded to generate an extensive library of subroutines which is described in the SHARE distribution material. This material should be consulted by the coder before he begins coding. To aid in this we have enclosed (as an appendex) an index to this material. If a coder finds a useful subroutine he can include the symbolic cards in his deck by requesting these at the Computation Center.

A certain small number of very useful subroutines have been recorded on a magnetic tape unit and can be automatically included in a program during assembly by using the (previously defined) LIB pseudo-operation. The list of subroutines available in this fashion will of course vary from installation to installation and even from time to time.

A word of warning: subroutines as submitted to SHARE can be as fully general in their symbol structure as programs. In using a subroutine a coder must carefully check what symbols are assigned within it and avoid conflicts (i.e. duplicate symbols). If he wishes the coder may avoid trouble by placing a unique heading character in front of each subroutine he uses.

In order to further clarify the above concepts we shall now describe in detail a particular SHARE subroutine, UABDC1, which can be used for printing floating-point and fixed-point numbers.

704 Generalized Print Program, UABDC1

UABDC1 is a subroutine which converts floating binary or integral binary numbers to binary-coded-decimal numbers arranged in a rather arbitrary format. It is described in SHARE distribution No. 72.

Using UABDC1 the coder may specify that a block of numbers (whose initial and final addresses may be specified) be printed according to a rather arbitrary format. Both these addresses and the format are program parameters of the subroutine. A format statement consists of a string of BCD characters formed according to certain rules which we shall omit here since they are exactly the same as the rules for forming FORMAT statements in FORTRAN language (see page 26 of the FORTRAN programmers reference manual).

To program a format specification using UABDC1 the coder must write an interlude of the form

```
SYMBOL   TRA   BLOCK
         BCD   VF
```

where F denotes an arbitrary format statement ending in at least one blank, V is the word count required by the BCD pseudo-operation, BLOCK is a particular symbol assigned at the beginning of UABDC1 and SYMBOL denotes an arbitrary symbol assigned by the coder.

A format specification interlude can be referred to from the main program by a calling sequence of the form

```
MAIN   TSX   SYMBOL,4
       PZE   A,0,B
```

where A and B are the initial and final addresses of the words to be printed.

UABDC1 occupies 405 storage registers and requires

60 or more additional registers of COMMON storage (i.e. a block of registers tagged at the beginning by the symbol, COMMON). If a column width, W> 30, appears in a format specification then W-30 additional registers of COMMON storage must be provided in the locations COMMON-1, COMMON-2..., COMMON - W+30.

UABDC1 is not complete in itself and requires a satellite subroutine (i.e. a subroutine which it uses) for actual printing. Two compatable subroutines are available:

1. UASTH1 : which records for off-line printing.
2. UASPH1 : which records for on-line printing.

Both of these subroutines are described in SHARE distribution No. 72. They are linked to UABDC1 by a symbol (namely WOT), which is used in UABDC1 and assigned in UASTH1 or UASPH1.

UASTH1 occupies 15 storage registers and requires no COMMON storage. UASPH1 occupies 109 storage registers and requires 33 registers of COMMON storage.

Since UABDC1 defines many symbols internally it is recommended that it and its satellite be prefixed by a heading card.

CHAPTER VIII

FIXED-POINT ARITHMETIC IN THE 704

Introduction

In the preceding chapters we have been describing 704 instructions which perform computations on floating-point numbers. These numbers are extremely convenient for solving most scientific problems. However, in many problems one encounters data which cannot conveniently be expressed in this form. Indeed this situation arises in almost every problem since, as we shall see, coding for the input-output devices connected to the computer cannot be done in terms of floating-point numbers..

In the next three chapters we shall describe some 704 instructions which are not designed for dealing with floating-point numbers. It should be emphasized, however, that all 704 instructions deal indiscrimanently with 36 binary digit words and have no way of telling what the coder means by these digits. Thus any of the instructions we are about to describe can operate formally on floating-point numbers, however the results obtained are difficult to describe in terms of such numbers. Such usage of these intructions is most often an error but may, in rare cases, be an ingenious move on the coder's part.

Fixed-Point Numbers

The 704 instructions defined in this chapter deal primarily with fixed-point numbers. In such numbers the entire 36 digits of the storage word are used to represent a single binary number. The sign digit is used to designate the sign of the number (with the convention that 0 denotes +) and the remaining 35 binary digits are used to represent the magnitude of the number. As we shall see, the binary point can be construed to lie anywhere within the number, however, for purposes of describing the instructions it is most convenient to assume that the binary point lies to the left of digit 1 (fixed-point fractions). Some examples of fixed-point fractions are given below:

S	1	2		34	35	
0	1	0	....	0	0	$+2^{-1} = 1/2$
1	0	1	....	0	0	$-2^{-2} = -1/4$
1	0	0	....	0	1	-2^{-35}
0	0	0	....	0	0	+0
1	0	0	....	0	0	-0

The reader should again note the existance of two zeros which differ only in the sign digit. Fixed-point fractions are available in the 704 register for expressing all integral multiples of 2^{-35} in the range:

$$-1 + 2^{-35} \leq x \leq +1 - 2^{-35}$$

The Fixed-Point Accumulator

In describing the fixed-point instructions, it is no longer convenient to gloss over the binary nature of the machine. The reader is already familiar with the fact that 704 registers can be used to store 36 digit binary numbers. We shall label these digits from left to right by the symbols:

S, 1, 2,...,35

For convenience we introduce the notation:

$C(x)_i$

to denote the "contents of digit i in register x". For example:

$C(x)_S$

An obvious extension of this notation is:

$C(x)_{1,2,5-7}$

to denote the contents of digits 1, 2 and 5 through 7 of register x.

The MQ register in the 704 is exactly like a storage register.

The AC, however, contains two extra digits which are important to the fixed-point instructions. These are called the Q bit and the P bit. The ordering of bits in the AC is as follows:

S, Q, P, 1,...,35

The Q and P bits are cleared by CLA and CLS. They are not transmitted to storage by STO. They may be affected by the floating-point arithmetic instructions but this need not concern us for the present.

The Ambidextrous Instructions

Certain of the instructions already introduced can be used in dealing with either fixed-point numbers or floating-point numbers. Proving this assertion requires a knowledge of exactly how floating-point numbers are stored in the 704. The reader is urged to acquire this knowledge from the 704 manual (p8).

For the sake of completeness these instructions are redefined below in terms of the notation of the preceding section.

Administrative Instructions:

(1) CLA x : $C(x) \rightarrow C(AC)_{S,1-35}$, $0 \rightarrow C(AC)_{Q,P}$

(2) LDQ x : $C(x) \rightarrow C(MQ)$

(3) STO x : $C(AC)_{S,1-35} \rightarrow C(x)$

(4) STQ x : $C(MQ) \rightarrow C(x)$

Control Instructions:

(1) TRA x : $x \rightarrow C(ILC)$

(2) TMI x : $C(AC)_S = 1 \Rightarrow x \rightarrow C(ILC)$

(3) TPL x : $C(AC)_S = 0 \Rightarrow x \rightarrow C(ILC)$

(4) TNZ x : $C(AC)_{Q,P,1-35} \neq 0 \Rightarrow x \rightarrow C(ILC)$

(5) TZE x : $C(AC)_{Q,P,1-35} = 0 \Rightarrow x \rightarrow C(ILC)$

(6) TQP x : $C(MQ)_S = 0 \Rightarrow x \rightarrow C(ILC)$

(7) TLQ x : $C(MQ) < C(AC) \Rightarrow x \rightarrow C(ILC)$

(8) CAS x : $C(AC) > C(x) \Rightarrow C(ILC) + 1 \rightarrow C(ILC)$
$C(AC) = C(x) \Rightarrow C(ILC) + 2 \rightarrow C(ILC)$
$C(AC) < C(x) \Rightarrow C(ILC) + 3 \rightarrow C(ILC)$

Arithmetic Instructions:

(1) CLS x : $C(x)_{1-35} \rightarrow C(AC)_{1-35}$, $\overline{C(x)}_S \rightarrow C(AC)_S$,
$0 \rightarrow C(AC)_{Q,P}$*

(2) SSP : $0 \rightarrow C(AC)_S$

(3) SSM : $1 \rightarrow C(AC)_S$

(4) CHS : $\overline{C(AC)}_S \rightarrow C(AC)_S$

If we consider the effect of the instruction, CLA x where register x contains a fixed-point fraction we see that we must construe the binary point of the AC to lie between digits P and 1. The AC can thus be used to store any integer multiple of 2^{-35} in the range:

$$-4 + 2^{-35} \leqslant x \leqslant + 4 - 2^{-35}$$

* The notation, $\overline{C(x)}_i$, is defined by the equations:
$\overline{0} = 1$ and $\overline{1} = 0$

Fixed-Point Addition and Subtraction

The following instructions can be used to add and subtract fixed-point fractions:

(1) ADD x : Add
(2) ADM x : Add Magnitude
(3) SUB x : Subtract
(4) SBM x : Subtract Magnitude

(1) The instruction, ADD x, algebraically adds C(x) to C(AC) and places the sum in the AC:

ADD x : $C(AC) + C(x) \rightarrow C(AC)$

(2) The instructions, ADM x, algebraically adds the magnitude of C(x) to C(AC) and places the sum in the AC.

ADM x : $C(AC) + |C(x)| \rightarrow C(AC)$

The magnitude of a number stored in the sign-magnitude convention may be obtained by setting its sign digit to 0.

(3) The instruction, SUB x, algebraically subtracts C(x) from C(AC) and places the difference in the AC.

SUB x : $C(AC) - C(x) \rightarrow C(AC)$

(4) The instruction, SBM x, subtracts the magnitude of C(x) from C(AC) and places the difference in the AC.

SBM x : $C(AC) - |C(x)| \rightarrow C(AC)$

None of the above instructions disturbs C(x).

If a sum or difference obtained using the above instructions is zero, its sign is the same as the sign of the original contents of the AC.

Overflows

The sum or difference of two fixed-point fractions may exceed unity. When this occurs ~~if~~ IT cannot be represented by 36 digits and thus cannot be placed in a storage register by the instruction:

STO x

which does not transmit the P and Q bits. Such a situation is called an overflow and the coder may have to know that it has occured to correctly interpret his results. The detection of overflows by the 704 is simplified by the presence of an AC overflow light which is turned on (by certain instructions) whenever a digit, 1, passes between digits 1 and P of the AC. In fixed-point addition and subtraction this is caused by carries and borrows.

The AC overflow light can be sensed and turned off by either of the following conditional control instructions:

(1) TOV x : Transfer on Overflow

(2) TNO x : Transfer on No Overflow

(1) The instruction, TOV x, directs the computer to turn off the AC overflow light and to take its next instruction from register x if the AC overflow light was on.

TOV x : ACOV on ⇒ x → C(ILC) and ACOV off.

(2) The instruction, TNO x, directs the computer to turn off the AC overflow light and to take its next instruction from register x if the AC overflow light was off.

TNO x : ACOV off → x → C(ILC) and ACOV off

The instructions, TOV and TNO, are the only means by which a program can turn off the AC overflow light. Thus if a coder wishes to detect that an overflow occured in a particular instruction he must turn off the overflow light immediately before executing the instruction and sense the overflow light immediately after executing the instruction (or at least before executing any other instruction which might turn the overflow light on).

In performing fixed-point additions and subtractions carries (borrows) can also occur between bits P and Q of the AC or out of bit Q. Such carries do not affect the status of the AC overflow light. Indeed carries out of the Q bit are simply lost to the computer so that numerical results obtained in such a case must be carefully interpreted.

Fixed-Point Multiplication

The following instructions can be used to multiply fixed-point fractions:

(1) MPY x : Multiply

(2) MPR x : Multiply and Round

(1) The instruction, MPY x, multiplies C(x) by C(MQ) and stores the result (a 70 digit exact product) in the AC and MQ. The 35 most significant digits of the product appear in AC_{1-35} and the 35 least significant digits of the product appear in MQ_{1-35}. The sign digit of the product appears both in AC_S and MQ_S. AC_Q and AC_P are set to zero.

The MPY instruction does not affect the status of the AC over flow light (since the product of two fractions is a fraction).

$$\text{MPY x:}\quad C(MQ)\cdot C(x)\rightarrow C(AC+MQ)$$
$$C(AC)_S = C(MQ)_S$$
$$0\rightarrow C(AC)_{Q,P}$$

Before describing the instruction, MPR x, we introduce the following instruction:

RND: Round

which cannot have an address section.

The instruction, RND, increases the magnitude of the fixed-point number in the AC by 2^{-35} provided that $C(MQ)_1 = 1$. If $C(MQ)_1 = 0$ nothing happens. In neither case is C(MQ) or $C(AC)_S$ affected by the instruction. The instruction can thus be used to round off a 72 digit number in the AC and MQ to a 37 digit number in the AC.

$$\text{RND: } C(MQ)_1 = 1 \Rightarrow C(AC)_{Q,P,1-35} + 2^{-35}\rightarrow C(AC)_{Q,P,1-35}.$$

The reader should verify that in certain cases the RND instruction will turn on the AC overflow light.

(2) The instruction, MPR x, is equivalent to the following sequence of instructions

MPY x
RND

The reader should verify that the MPR instruction cannot affect the status of the AC overflow light.

Fixed-Point Division

The following instructions can be used to divide fixed-point fractions:

(1) DVH x : Divide or Halt

(2) DVP x : Divide or Proceed

(1) The instruction, DVH x, considers C(AC) and $C(MQ)_{1-35}$ to be a 72 digit signed dividend ($C(MQ)_S$ is ignored) and C(x) to be a 35 digit signed divisor.

If $|C(AC)| \geq |C(x)|$ then division does not occur since the quotient in this case is not a fraction.* The computer turns on a light called the divide-check light and stops. The dividend remains undisturbed in the AC and MQ.

If $|C(AC)| < |C(x)|$ the quotient is a fraction and division occurs. The instruction forms a 35 digit signed quotient, q, which replaces C(MQ) and a 35 digit signed remainder, r, which replaces $C(AC)_{S,1-35}$ (recall that $C(AC)_P = C(AC)_Q = 0$). q and r are <u>fractions</u> having the following properties:

(a) The following equation holds exactly

$$C(AC+MQ) = q \cdot C(x) + r \cdot 2^{-35} \qquad 0 \leq |r| < C(x)$$

(b) The s.gn of the remainder, r, is the same as the sign of the dividend, C(AC+MQ).

The reader should verify property (b) guarantees that the magnitude of q is less than or equal to the magnitude of the true quotient. This means that 35 additional digits of the true quotient can be obtained if one divides r by C(x).

(2) The instruction, DVP x, executes a division as just described if $|C(AC)| < |C(x)|$. If $|C(AC)| \geq |C(x)|$ the divide-check light is turned on and the computer proceeds to the next instruction without disturbing C(AC) and C(MQ).

The divide-check light can be sensed and turned off by the following conditional control instruction:

* The reader should note that this automatically occurs unless $C(AC)_Q = C(AC)_P = 0$ so that describing the dividend as a 72-digit number is fiction.

DCT : Divide-Check Test

which cannot have an address section.

The instruction, DCT, directs the computer to turn off the divide-check light and to skip one instruction if the divide-check light was off.

DCT : DVCK off $\Rightarrow$ C(ILC)+2 $\rightarrow$ C(ILC) and DVCK off.

The DCT instruction is the only means by which a program can turn off the divide-check light.

CHAPTER IX

THE SHIFTING INSTRUCTIONS

Introduction

The shifting instructions are used to move the digits in the AC and MQ to the right or left of their original positions. They can be split into two classes: the numerical shifting instructions and the logical shifting instructions.

In the numerical shifting instructions the sign digits are not shifted and zeros are brought into digit positions which are vacated by shifting. Thus in most cases these instructions can be interpreted as multiplication or division by a power of two when the AC and MQ contain fixed-point fractions.

In the logical shifting instructions the sign digit may be shifted along with the numerical digits. A numerical interpretation in this case becomes difficult and the best viewpoint would seem to be that which regards the contents of a register as an array of binary digits. We shall have more to say of this later on.

The address section of a shifting instruction does not refer to a register in storage but instead specifies the number of positions that digits are to be shifted. The upper limit, 255, is placed on the number of positions that digits can be shifted but since the combined length of the AC and MQ is 74 digits this is more than adequate. We can state precisely what happens as follows: if n denotes the address of a shifting instruction, then the number of digit shifts implied by the instruction is n mod 256 (i.e. the remainder obtained when n is divided by 256.)

The Numerical Shifting Instructions

There are four numerical shifting instructions:

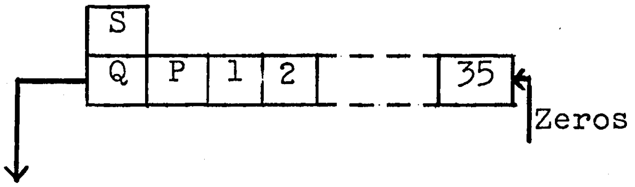
(1) ALS n : AC Left Shift

(2) ARS n : AC Right Shift

(3) LLS n : Long Left Shift

(4) LRS n : Long Right Shift

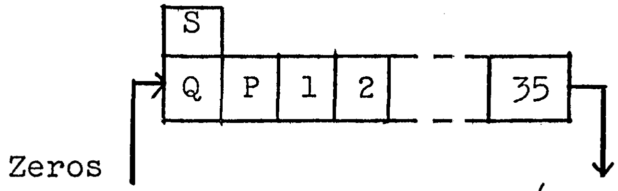
(1) The instruction, ALS n, shifts $C(AC)_{Q,P,1-35}$ to the left n mod 256 positions. Digits shifted from the Q bit are lost and zeros are introduced into digit 35 to fill digit positions vacated by shifting. $C(AC)_S$ and $C(MQ)$ are unaffected.



The AC overflow light will be turned on if a 1 is shifted into or through the P bit.

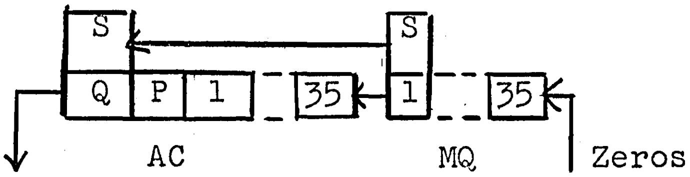
$$\text{ALS n : } C(AC) \cdot 2^{n \bmod 256} \longrightarrow C(AC)$$

(2) The instruction, ARS n, shifts $C(AC)_{Q,P,1-35}$ to the right n mod 256 positions. Digits shifted from digit 35 are lost and zeros are introduced into the Q bit to fill digit positions vacated by shifting. $C(AC)_S$ and $C(MQ)$ are unaffected.



$$\text{ARS n : } C(AC) \cdot 2^{-(n \bmod 256)} \longrightarrow C(AC)$$

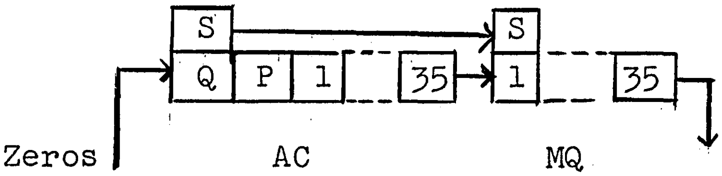
(3) The instruction, LLS n, copies $C(MQ)_S$ into AC_S and shifts $C(AC)_{Q,P,1-35}$ and $C(MQ)_{1-35}$ to the left n mod 256 positions. Digits shifted from AC_Q are lost and zeros are introduced into MQ_{35} to fill digit positions vacated by shifting. $C(MQ)_S$ is unaffected.



The AC overflow light will be turned on if a 1 is shifted into or through the P bit.

$$\text{LLS n : } C(AC + MQ) \cdot 2^{n \bmod 256} \rightarrow C(AC + MQ)$$
$$C(MQ)_S \rightarrow C(AC)_S$$

(4) The instruction, LRS n, copies $C(AC)_S$ into MQ_S and shifts $C(AC)_{Q,P,1-35}$ and $C(MQ)_{1-35}$ to the right n mod 256 positions. Digits shifted from MQ_{35} are lost and zeros are introduced into the Q bit to fill digit positions vacated by shifting. $C(AC)_S$ is unaffected.



$$\text{LRS n : } C(AC + MQ) \cdot 2^{-(n \bmod 256)} \rightarrow C(AC + MQ)$$
$$C(AC)_S \rightarrow C(MQ)_S$$

The reader is urged to consider the effects of these instructions for the case n mod 256 = 0.

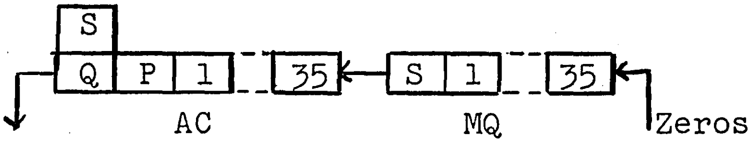
The Logical Shifting Instructions

There are two logical shifting instructions:

(1) LGL n : Logical Left

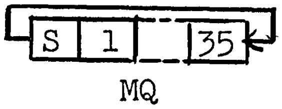
(2) RQL n : Rotate MQ Left

(1) The instruction, LGL n, shifts $C(AC)_{Q,P,1-35}$ and $C(MQ)_{S,1-35}$ to the left n mod 256 positions. Digits shifted from the Q bit are lost and zeros are introduced into MQ_{35} to fill digit positions vacated by shifting. $C(AC)_S$ is unaffected. This instruction differs from LLS since MQ_S is shifted like any of the numerical digits, i.e. digits shifted from MQ_1 enter MQ_S and digits shifted from MQ_S enter AC_{35}.



The AC overflow light will be turned on if a 1 is shifted into or through the P bit.

(2) The instruction, RQL n, rotates $C(MQ)_{S,1-35}$ to the left n mod 256 positions, i.e., digits shifted from MQ_1 enter MQ_S and digits shifted from MQ_S enter MQ_{35}.



Integer Arithmetic

The restricting of fixed-point numbers to be fractions is largely artificial and was adopted to simplify the explanation of the fixed-point instructions. As far as addition and subtraction are concerned, any consistent assumption for the location of the binary point yields correct results (provided, of course, the binary point of the result is construed to be in the same location). In cases where the binary points of the operands are not consistent they can easily be made consistent by using a shifting instruction. The coder must be careful, however, not to lose significant digits in the process.

In multiplication and division the locations of the binary points of the operands need not be consistent. The coder, however, must memorize certain rules for determining the location of the binary point in the result. These rules are easily derived.

(1) Multiplication: Consider a fixed-point number, A, having n digits to the left of its binary point and a fixed-point number,B, having m digits to the left of its binary point. This means that

$$A \cdot 2^{-n} \text{ and } B \cdot 2^{-m}$$

are fractions which if multiplied using MPY yield the fraction

$$P = AB \cdot 2^{-(n+m)}$$

Since the true product is

$$AB = P \cdot 2^{n+m}$$

the product has n+m digits to the left of its binary point.

(2) Division: If the above fractions are divided using DVH or DVP we obtain a quotient, q, and a remainder, r, where

$$\frac{A \cdot 2^{-n}}{B \cdot 2^{-m}} = q + \frac{r}{B \cdot 2^{-m}} \cdot 2^{-35}$$

Since the true quotient and remainder are

$$\frac{A}{B} = q \cdot 2^{n-m} + \frac{r \cdot 2^{n}}{B} \cdot 2^{-35}$$

the quotient has n-m digits to the left of its binary point and the remainder has n digits to the left of its binary point.

For integer multiplication we let n=m=35. The binary point of the product is thus located at the right-hand end of the MQ, e.g.,

```
          LDQ X
          MPY Y
          STQ PROD              C(X) • C(Y)→C(PROD)
```

suffices provided that C(X) • C(Y) can be expressed as a 35-digit integer.

For integer division we let n=70 and m=35 and obtain an integer quotient and remainder. The integer dividend must be placed in the MQ; however, in doing so the coder must guarantee that the AC contains a zero having the sign of the dividend (since $C(MQ)_S$ is ignored in division). The following program would suffice

```
          CLA X
          LRS 35
          DVP Y                 X/Y
          STO REM               r→C(REM)
          STQ QUOT              q→C(QUOT)
```

How Fixed-Point Numbers Are Written When Programming

The Share Assembly Program provides a fairly general notation for writing fixed-point numbers. Their most general form is the following

DEC $NEe_{10}Be_{2}$

where N denotes any mixed number (+ signs and decimal points are optional), e_{10} denotes any integer and

$e_2 = 0,1,2,\ldots,35$

The notation $NEe_{10}Be_{2}$ stands for the fractional part of the number

$$N \cdot 10^{e_{10}} \cdot 2^{-e_2}$$

rounded off to 35 binary digits.

The decimal scale-factor, Ee_{10}, is optional and may be omitted. The binary scale-factor, Be_2, must, however, always appear so that these numbers can be distinguished from floating-point numbers. If $e_2=0$, however, it may be omitted yielding

```
          DEC    NEe10B
```

Thus the fraction, 1/10, may be written in any of the following forms

$$.1B = 1E\text{-}1B = 10E\text{-}2B$$

If $e_2=35$ is used we obtain integers

```
          DEC    1B35 = 1 · 2^-35
          DEC    2B35 = 2 · 2^-35
                 .........
          DEC   1E1B35 = 10 · 2^-35
```

Since integers occur frequently they have been made a special case by SAP which allows integers to be denoted by simply writing their integer value

```
          DEC  1
          DEC  2
          ......
          DEC  10
```

Integers are characterized by the fact that neither E nor B nor a decimal point appears in their definition.

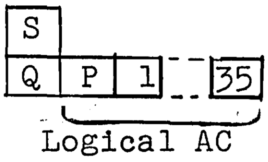
The binary scale-factor, e_2, may also be interpreted as specifying the number of binary places between the left-hand end of the register and the binary point of the fixed-point number.

CHAPTER X

THE LOGICAL INSTRUCTIONS

The Logical Word

The logical word in some ways is the simplest of words since all of its digits are treated the same. In storage (of course) the logical word occupies digits S and 1-35. In the AC the logical word will normally occupy digits P and 1-35.



Thus the CLA instruction will not suffice for copying a logical word from storage to the logical AC.

Logical words are of great importance in problems which are primarily non-numerical. The casual coder may, however, have little use for them except in programming for the input-output units (a job most often handled by library subroutines).

The Logical Administrative Instructions

The logical administrative instructions are the following:

(1)	CAL	x	:	Clear and Add Logical
(2)	SLW	x	:	Store Logical Word
(3)	STP	x	:	Store Prefix
(4)	STD	x	:	Store Decrement
(5)	STA	x	:	Store Address

(1) The instruction, CAL x, copies C(x) into the logical AC. $C(x)_S$ is copied into AC_P. Zeros are copied into $C(AC)_{S,Q}$. C(x) is unaffected.

CAL x : $C(x) \rightarrow C(AC)_{P,1-35}$

$0 \rightarrow C(AC)_{S,Q}$

(2) The instruction, SLW x, copies the logical AC into register x. $C(AC)_P$ is copied into $C(x)_S$. C(AC) is unaffected.

SLW x : $C(AC)_{P,1-35} \rightarrow C(x)$

(3) The instruction, STP x, copies $C(AC)_{P,1,2}$ into $C(x)_{S,1,2}$. C(AC) and $C(x)_{3-35}$ are unaffected.

STP x : $C(AC)_{P,1,2} \rightarrow C(x)_{S,1,2}$

(4) The instruction, STD x, copies $C(AC)_{3-17}$ into $C(x)_{3-17}$. C(AC) and $C(x)_{S,1,2,18-35}$ are unaffected.

STD x : $C(AC)_{3-17} \rightarrow C(x)_{3-17}$

(5) The instruction, STA x, copies $C(AC)_{21-35}$ into $C(x)_{21-35}$. C(AC) and $C(x)_{S,1-20}$ are unaffected.

STA x : $C(AC)_{21-35} \rightarrow C(x)_{21-35}$

The Logical Arithmetic Instructions

The logical arithmetic instructions are the following:

(1)	ANA	x	:	And to AC
(2)	ANS	x	:	And to Storage
(3)	ORA	x	:	Or to AC
(4)	ORS	x	:	Or to Storage
(5)	COM		:	Complement Magnitude
(6)	CLM		:	Clear Magnitude
(7)	ACL	x	:	Add and Carry Logical

(1) The instruction, ANA x, compares each digit in the logical AC with the corresponding digit in C(x). If both of these digits are 1, then a 1 replaces the corresponding digit

in the AC, otherwise a 0 is placed in this digit. $C(AC)_{S,Q}$ are cleared.

$$\text{ANA} \quad x : \quad C(AC)_i \wedge C(x)_i \rightarrow C(AC)_i \qquad i = P(S), 1\text{-}35$$

$$0 \longrightarrow C(AC)_{S,Q}$$

The table of combinations for the "and" function is given below:

		0	1	(d_x)
(d_{AC})	0	0	0	
	1	0	1	

(2) The instruction, ANS x, is like ANA except that C(AC) is undisturbed and register x gets the result which ANA would have placed in the logical AC.

$$\text{ANS} \quad x : \quad C(AC)_i \wedge C(x)_i \rightarrow C(x)_i \qquad i = P(S), 1\text{-}35$$

(3) The instruction, ORA x, compares each digit in the logical AC with the corresponding digit in C(x). If <u>either</u> or <u>both</u> of these digits are 1, then a 1 replaces the corresponding digit in the AC, otherwise a 0 is placed in this digit. $C(AC)_{S,Q}$ are unaffected.

$$\text{ORA} \quad x : \quad C(AC)_i \vee C(x)_i \rightarrow C(AC)_i \qquad i = P(S), 1\text{-}35$$

The table of combinations for the "or" function is given below.

		0	1	(d_x)
(d_{AC})	0	0	1	
	1	1	1	

(4) The instruction, ORS x, is like ORA except that C(AC) is undisturbed and register x gets the result which ORA would have placed in the logical AC.

$$\text{ORS} \quad x : \quad C(AC)_i \vee C(x)_i \rightarrow C(x)_i \qquad i = P(S),1\text{-}35$$

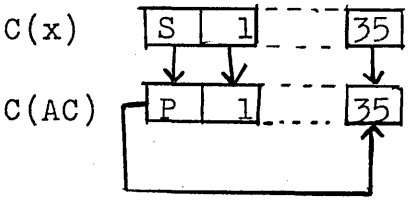
(5) The instruction, COM, cannot have an address section and affects only C(AC) as follows: every 0 in $C(AC)_{Q,P,1-35}$ is replaced by a 1 and every 1 in $C(AC)_{Q,P,1-35}$ is replaced by a 0. $C(AC)_S$ is unaffected.

$$\text{COM} \quad : \quad \overline{C(AC)_i} \rightarrow C(AC)_i \qquad i = Q,P,1\text{-}35$$

(6) The instruction, CLM, cannot have an address section. It affects only C(AC) and places zeros in $C(AC)_{Q,P,1-35}$. $C(AC)_S$ is unaffected.

$$\text{CLM} \quad : \quad 0 \rightarrow C(AC)_{Q,P,1-35}$$

(7) The instruction, ACL x, adds C(x) to the contents of the logical AC. ($C(x)_S$ is added to $C(AC)_P$ like any other numerical digit.) A carry out of the P bit does not, however, get added into the Q bit. Instead it is added into AC_{35}. $C(AC)_{S,Q}$ are unaffected.



A use for the ACL instruction will be described in the chapter on input-output equipment.

The Logical Control Instructions

The logical control instructions are the following

(1) PBT : P Bit Test
(2) LBT : Low Bit Test

(1) The instruction, PBT, causes the computer to skip the next instruction in sequence if $C(AC)_P = 1$.

$$\text{PBT} : \quad C(AC)_P = 1 \Rightarrow C(ILC) + 2 \rightarrow C(ILC)$$

The PBT instruction cannot have an address section.

(2) The instruction, LBT, causes the computer to skip the next instruction in sequence if $C(AC)_{35} = 1$.

$$\text{LBT} : C(AC)_{35} = 1 \Rightarrow C(ILC) + 2 \rightarrow C(ILC)$$

Octal Numbers

The logical word is frequently interpreted as an array of binary digits. For this reason a SAP notation has been provided by means of which a coder can write binary numbers in his program. Since the binary notation would be cumbersome (36 digits required per word) SAP provided a means for writing octal (base 8) numbers instead.

The conversion between binary and octal is trivial and is based entirely on the following table which can easily be memorized:

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

To make the conversion from binary to octal the coder breaks up the 36 digit binary word into 12 groups of 3 digits and writes down the octal digits corresponding in the table.

The notation used for octal numbers is the following

OCT N

where N denotes a 12 digit octal integer. The coder may also write octal integers with algebraic signs and drop insignificant zeros. In this case the integer can contain at most 12 digits. If it contains exactly 12 digits then the leading digit should be 0,1,2 or 3 since one binary digit is used up by the algebraic sign.

Instruction Arithmetic

A significant feature (probably the significant feature) of the digital computer is the fact that instructions are placed in the storage element and can be operated on by the computer. The importance of being able to change effective addresses with index registers has already been made clear. By using the fixed-point and logical instructions the coder may perform arbitrary arithmetic on his program. To do so however he should know exactly how instructions appear in the storage register (see page 7 of the 704 manual).

The significant point involved is that an instruction will be considered to be a positive or negative number according to a rather arbitrary array of digits which define its operation section. As an example of a possible difficulty suppose that register INST contains an instruction and register INT contains an integer, and that we wish to increase the address of C(INST) by C(INT). We may be tempted to use the sequence

```
CLA   INST
ADD   INT
STO   INST
```

This would work as long as C(INST) were an instruction with a 0 sign digit (e.g., CLA). If, however, it had a 1 in the sign digit (e.g. MPR) we would decrease (rather than increase) the address by C(INT). This is a consequence of the sign-magnitude convention used by the 704.

The above difficulty could have been avoided by using the sequence

```
CAL   INST
ADD   INT
SLW   INST
```

or the sequence

```
CLA   INC
ADM   INST
STA   INST
```

Some Examples Using the Logical Instructions

Example 1: An integer division can be performed using the following program

```
LDQ   NUM   ┐  Place 0 in AC having same
CLM         │  sign as C(NUM)
LLS   0    ─┘
DVH   DENOM    ~~C(NUM) / C(MQ)~~ c(NUM)/c(DENOM)⇒c(MQ)
```

Example 2: The "or" function formed by ORA and ORS is called an inclusive or. An equally important function is the exclusive or which produces 1 in the AC whenever $C(AC)_1 = 1$ and $C(X)_1 = 1$ but not both. This can be done by the following program

```
SLW   TEMP1
ANA   X
COM
SLW   TEMP2
CAL   TEMP1
ORA   X
ANA   TEMP2
```

Readers familiar with Boolean algebra will recognize the function, (A + X) (AX)'. An easy way to check the program however is to try it out for the four possible cases.

Example 3: The following program inverts the digits of the binary number in register BIN, i.e. i.e. , $C(BIN)_{35} \rightarrow C(BIN)_S$, $C(BIN)_{34} \rightarrow C(BIN)_1$, $C(BIN)_{33} \rightarrow C(BIN)_2$, etc.

```
             LXA   COUNT,1
             LDQ   BIN
             RQL   1
LOOP         RQL   34        ]   Loop forms inverted
             LGL   1         ]   word one digit at a time
             TIX   LOOP,1,1  ]   in the AC.
             SLW   BIN

COUNT        HPR   36
```

Example 4: The "and" instructions can be used to extract a group of digits from a logical word. This is done by "anding" together the logical word and a special constant (sometimes called a mask) which contains 1's in the digit positions corresponding to the group and 0's elsewhere. For example, if we desire to know if digits 30-35 of register BIN contain the binary number

101101

we can write the following program

```
             CAL   BIN    ]     Extract required digits
             ANA   MASK   ]     to AC
             CAS   NUM    ]
             TXL   NO     ]     Exit to YES if
             TXL   YES    ]
             TXL   NO     ]     C(BIN)30-35 = 101101
     MASK    OCT   77
     NUM     OCT   55
     BIN
```

Care must be exercised in mixing arithmetic instructions (like CAS) with logical instructions. Suppose, for example, the coder desired to ask the above question about digits S, 1-5 of register BIN. He might be tempted to repeat the above program with

```
            C(MASK) = OCT  770000000000
```

and

```
            C(NUM)  = OCT  550000000000
```

but this would not work. (Can you say why and can you correct the program?)

Example 5: The "and" and "or" instructions can be used to store a binary number in a selected group of digits of a register without disturbing the other digits of the register. For example, if we desire to store the binary number

101101

in digits 30-35 of register BIN we could write the following program

```
                   CAL  MASK
                   ANS  BIN
                   CAL  NUM
                   ORS  BIN
                   HPR
            MASK   OCT  -377777777700
            NUM    OCT  55
            BIN
```

or alternatively

```
                   CAL  MASK
                   ANA  BIN
                   ADD  NUM
                   SLW  MASK
                   HPR
```

Example 6: In an earlier chapter it was mentioned that the integer, -n, when it occurred in an address was a shorthand notation for the positive integer, 32768 - n = 2^{15}-n. It may occur in a program that given n, we must compute -n. If n is in IR1 we could do this with the following program

```
              PXD  0,1
              COM
              ADD  DECR
              PDX  0,1
              HPR
       DECR   PZE  0,0,1
```

This program works since when a 15 digit integer is complemented and then added to its original value, the result is a 15 digit number consisting entirely of 1's, i.e. the number, 2^{15} - 1. If index registers are involved the above holds modulo machine size.

Two equivalent alternative programs follow:

```
       TXI  A,1,-1            PXD  0,1
    A  PXD  0,1               COM
       COM                    PDX  0,1
       PDX  0,1               TXI  A,1,1
       HPR                  A HPR
```

The simplest program obtainable, however, is the following

```
          PXD  0,1
          SUB  A
          PDX  0,1
          HPR
       A  PON
```

CHAPTER XI

INPUT AND OUTPUT

In the following chapter we shall briefly describe the input and output devices associated with the computer. We shall not go into many of the details of coding for these devices. The reader can consult the 704 manual for these. In many cases the coder will be able to completely avoid contact with these devices by using subroutines.

The Input-Output Devices

The main frame of the computer consists of the arithmetic and control elements and the high-speed memory. All communication between the main frame and the rest of the universe comes under in-out (IO). IO therefore includes input, output, and auxiliary storage (the preservation of binary computer words outside the main frame) IO equipment is classified as on-line if it is under the direct control of the main frame. Off-line or peripheral equipment may be operated independently.

At the Computation Center there are a cathode ray tube (CRT) unit, a magnetic drum unit, a card reader, a punch, a printer, and up to ten magnetic tape units on line. A card reader, a punch, and a printer comprise the off line complement. Each of these peripheral units has a magnetic tape unit associated with it, but the tape unit for the off-line punch may be connected on-line as a tenth unit. The console lights and switches, although part of IO, are not covered here.

IO Programming

IO programming has two parts. One, obviously, consists of the actual transfer of data between the main frame and the desired unit. The other is the manipulation of the data before output or after input, since meaningful words inside the main frame are not readily intelligible by human beings or the electrical accounting machines (EAM). Even to use the CRT properly some manipulation is necessary to prepare the data for display.

The basic IO instructions are

1) RDS n : Read Select
2) WRS n : Write Select
3) CPY x : Copy

RDS selects the on-line unit whose identifying number is n in the input mode. WRS selects a unit in the output mode. A table of identifying numbers appears on page 27 of the 704 manual. The coder, however, need not remember these numbers since SAP provides mnemonic pseudo-operations for each unit. Both RDS and WRS can have tags but rarely do. All the words transferred between one select instruction and the next comprise a record. A collection of records terminated in a special way is called a file.

A CPY instruction usually transfers one word between core memory at the location effectively addressed and the unit currently selected. The word passes through the MQ, which is used as a buffer for IO. If no unit is currently selected, the 704 stops with the Read-Write Check Light on. There is no instruction which deselects a unit; a unit is automatically disconnected if a certain time has elapsed since the last CPY or if the maximum number of CPY instructions for the record have already been executed.

When reading magnetic tape or from the card reader, records and files are well-defined. If a CPY is given after the last word of a record has been read, core memory is unchanged and control is transferred to the third instruction beyond the CPY. This feature facilitates reading records whose lengths are not known or records of known length without having to count words. When an end-of-file condition has been established, the first CPY given after an RDS causes the ILC to be increased by two, skipping one instruction and transferring control to the instruction two registers beyond the CPY.

It is often desirable to compute and record check-sums with auxiliary storage records so that when the record is read a new check-sum can be formed and compared with the recorded one. This process if facilitated by the instruction

CAD x : Copy and Add

which does everything CPY does and in addition logically adds+ the word being transferred and C(AC). It is thus possible to transfer data and form a check-sum simultaneously. CAD is not described in the 704 manual.

Every unit but the card reader has some special instructions associated with it. These will be described with the unit.

Coding for the Cathode Ray Tube

The complete CRT unit has two oscilloscopes, a large scope for visual display and a 7-inch CRT optically connected to a 35 mm. camera for permanent recording on film. For most programmers the display CRT is just a check that information is being recorded on film. The CRT has a theoretical raster size of 1024 by 1024. However spot size and the limitations of the system reduce this to an effective size of 256 by 256.

The CRT is selected by the SAP instruction

WTV

Note: all of the SAP WRS pseudo-instructions begin with W and all of the RDS pseudo-instructions begin with R. The CRT is unique in that it has no timing problem. It remains selected until some other IO unit is selected no matter when the CPY instructions are given. There is a minimum time between CPY's

+ See the description of the instruction, ACL.

but if this time is not used in computing the 704 will wait until the CRT is ready to accept more information.

The SAP instruction

CFF : Change Film Frame

advances the film and exposes a new frame. This should always be done before displaying points on the CRT. After CFF is given a half-second must elapse before the next CPY to the CRT can be executed.

The point to be displayed on the CRT is determined by the word transferred by the CPY. The x-coordinate is specified by the rightmost 10 digits of the decrement. The y-coordinate is specified by the rightmost 10 digits of the address. The digits of the prefix (y_s, y_1, y_2) have the following meanings:

$y_1 = y_2 = 0$	display point at (x,y)
$y_s = 0$	display with normal intensity
$y_s = 1$	display with high intensity
$y_1 = 1$, $y_2 = 0$	display horizontal axis through (x,y)
$y_1 = 0, y_2 = 1$	display vertical axis through (x,y)

For example, the following program displays a point in the center of the CRT

```
        WTV
        CPY  COORD
        HPR
COORD   PZE  512,0,512
```

The CRT is normally used to plot curves. Alphanumeric information may be written on film (by plotting it point by point) but it is preferable to use the off-line printer for large amounts of alphanumeric information.

Coding for the Magnetic Drum

The MIT 704 has four logical drums each capable of storing 2048 words. These are purely auxiliary storage devices.

While the access time for the first word transferred averages 12 ms, the rate at which subsequent words are transferred is 96μs per word. To attain this speed CPY instructions must not be more than 36μs apart. If a loop of the form

```
Z  CPY  Y, 1
   TIX  Z, 1, 1
```

is used, then the TIX instruction uses 24 of the allowable 36 μs so that no other instructions can be done in the copy lcop.

The drum is selected for reading and writing by the SAP instructions

```
WDR   n   :   Write Drum
RDR   n   :   Read Drum
```

where n = 1,2,3 or 4 specifies the number of the logical drum being selected. The instruction

```
LDA   x, t
```

is used (following the WDR or RDR) to select the register on the drum to which the first CPY will refer. This address is specified by the right most 11 digits of the word effectively addressed by the LDA (and <u>not</u> by the address section of the LDA itself). If drum address zero is desired the LDA may be omitted. The drum remains selected indefinitely after WDR or RDR waiting for an LDA. Once the LDA is given, however, a CPY must follow within 36μs. If more than 36μs elapses without a CPY being given the drum deselects.

The CAD instruction is usually used in drum copy lcops since the drum has no internal checking devices. Thus check-sums should be formed by the coder to guard against errors. In the following example we write 100 words on a drum. The first 99 words are data words and the last word is a check sum. The reader should note that the final data word must be handled separately from the main copy loop to avoid deselecting the drum.

```
      WDR  1         Select drum 1
DA    PXD  200       Prepare to form check-sum
      LDA  DA        Select location 200
      LXA  N,1       Set for 98 cycles
CAD   CAD  DATA,1    C(DATA-98),...,C(DATA-1)
      TIX  CAD,1,1
      CAD  DATA      C(DATA)
      SLW  TEMP
      CPY  TEMP      Record check-sum on drum
        . . . .
N     PZE  98
TEMP  PZE
```

The block of data can be read back into the computer and checked by the following program

```
      RDR  1         Select drum 1
DA    PXD  200       Prepare to form check sum
      LDA  DA        Select location 200
      LXA  N,1       Set for 99 cycles
CAD   CAD  DATA+1,1 C(DATA-98),...,C(DATA)
      TIX  CAD, 1,1
      CPY  TEMP      Check-sum from drum
      SLW  TEMP1     Compare it with computed check-sum
      CLA  TEMP1
      CAS  TEMP
      HTR  ERROR
      HTR  GOOD
      HTR  ERROR
N     PZE  99
TEMP  PZE
TEMP1 PZE
```

Coding for the Card Reader

The card reader is selected by the SAP instruction

RCD

It provides a natural introduction to the card image (see fig. 24 on page 40 of the 704 manual). Like all units which process card image type data it has a plug board. The SHARE organization has adopted a standard plugboard with which the first 72 columns of a card may be read, by half-rows, into the computer. The last 8 columns of the card may not be read.

Since each half-row can be construed as a 36 digit binary number it is convenient to read binary cards with the card reader. If Hollerith (BCD) data is punched on the card, however, the characters appear in the columns of the card. If these are read by the card reader the coder is presented with a tremendous unscrambling problem. Fortunately there are subroutines for doing this and moreover it can be done during the time that the card is being read by the computer.

A record for the card reader is a single card. A new RCD is needed for each card to be read. If an RCD is followed by 25 CPY instructions the 25th CPY produces an end-of-record skip (skipping two instructions after the CPY). An end-of-file condition can also be produced in the card reader by letting the card hopper empty. If this occurs the computer stops (still selecting the card reader). Pressing the start button on the reader at this time causes the computer to read the remaining cards in the reader. After the last card has been read the next RCD sets up an end-of-file condition and the first CPY following produces an end-of-file skip.

The card reader will deselect if the coder waits too long between CPY instructions or between the last CPY instruction required for a card and the RCD required for the new card. The timing is given in the manual.

Coding for the Card Punch

The card punch is essentially the inverse of the card reader. It is selected by the SAP instruction

WPU

The following SAP instructions

SPU n : Signal Punch Hub n = 1,2

produce pulses at the plugboard but the SHARE standard board for the punch has not been wired to use these pulses.

If more than 24 CPY instructions follow a WPU a read-write check stop occurs in the 704. A WPU instruction is required for each card to be punched. If too much time elapses between CPY instructions the punch will deselect.

The following program reads a card from the card reader and punches it on the card punch.

```
       LXA   HTR, 1        0→C(IR1)
       RCD                 Read Card
CPY1   CPY  CARDIM,1
       TXI  CPY1, 1,-1
 HTR   HTR                 End-of-file skip
       WPU                 End-of-record skip
CPY2   CPY  CARDIM-24,1    Punch Card
       TXI  NEXT,1,-1
NEXT   TXH  CPY2,1,-48
       HTR
```

Coding for the Printer

The printer can be selected by either of the SAP instructions

WPR
RPR

The latter on is less often used and selects the printer with echo checking. In this mode 24 CPY's provide a card image as for WPR or WPU but there are also an additional 22 CPY's which read back into memory the echo pulses from the print wheels. A printer card image must consist of Hollerith characters unlike the reader and punch which accept binary card images. There are standard subroutines for translating BCD characters into card image form.

The following special instructions are associated with the printer

SPR n : Signal Printer Hub 1 n 10
SPT Sense Printer Test

The SPR instruction causes a pulse to appear at the specified exit hub of the printer plug board. The SPT instruction causes the computer to skip one instruction if a pulse is being applied to the sense entry hub of the printer plug board. In SHARE standard board 2 the SPR instructions have the following meaning:

SPR 1: skip to channel 1, i.e. restore the paper form to the top of a page.

SPR 2: skip to channel 2

SPR 3: extra space (after printing)

SPR 4: double space

SPR 5: suppress spacing before printing

SPR 7: send pulse to be tested by SPT for board check.

SPR 8: suppress overflow, used with SPR9 to print 120 characters per line. This sense exit must always be pulsed during the first print cycle for a line.

SPR 9: suppress spacing and set so that next image goes into columns 73-120 from 1-48 with 49-72 in place, so as to be able to print 120 characters per line.

The following program will print a 120 character line with double spacing. We assume 72 characters in card image form in registers LEFT-24,...,LEFT-1 and 48 characters in card image form in RIGHT-24,...,RIGHT-1 (thus bits 12 through 35 of RIGHT-23, RIGHT-21,...must be 0).

```
          LXA  HTR,3      24→C(IR1) and C(IR2)
          WPR             Select for left side
          SPR  8          Prevent overflow skip.
    CPY1  CPY  LEFT,1     Left side image
          TIX  CPY1,1,1
          SPR  9          Prepare to print on right
          WPR             Select for right side
    CPY2  CPY  RGHT,2     Right side image
          TIX  CPY 2,2,1
          SPR  4          Double space
     HTR  HTR  24
```

Coding for the Magnetic Tape Units

In using magnetic tape a new difficulty is presented: the MQ may not be used between CPY instructions and for a time after the last CPY. This restriction also applied to the drum but in that case the time between CPY's was so short that the MQ could hardly have been used anyway. There are at most 28 machine cycles available for computation between CPY's while writing and 24 while reading. 42 cycles after the last effective CPY the MQ becomes available. The SAP instruction

IOD : In-Out Delay

if given after the last effective CPY will delay the computer long enough to make the MQ available.

For auxiliary storage the tape is used in the binary mode. The instructions for using tape are:

RTB	n	:	Read Tape Binary
WTB	n	:	Write Tape Binary
REW	n	:	Rewind
BST	n	:	Back space Tape
RTT		:	Redundancy Tape Test
ETT		:	End of Tape Test

Here $1 \leq n \leq 10$ selects one of the ten logical tape addresses.

An ETT instruction must be given while a tape is selected. The end-of-tape indicator which it tests can be turned on only during writing operations with the tape and may be turned off by a REW or BST.

The RTT instruction tests a light which is turned on whenever a line of tape is read which has a wrong lateral redundancy bit or whenever a record from tape is read which has a wrong longitudinal redundancy bit. RTT may be given after the tape unit has been deselected. A delay must be inserted between the last effective CPY and the RTT in order to give the tape unit time to compute the longitudinal redundancy bits. The coder may provide this delay if he wishes by giving the RTT immediately after the next RTB. In this case, however, he will require two BST's to return to the erroneous block.

The tape units are also used for input and output in connection with the off-line equipment. The following instructions are provided for this mode

RTD	n	:	Read Tape Decimal
WTD	n	:	Write Tape Decimal

The off-line card reader with the standard SHARE plugboard reads all 80 columns of a card and produces 14 BCD words, the last four characters of the last word being blanks. Each card produces a record on tape. An end-of-file gap can be written at the end of the deck. A decimal tape of this type may be used to punch cards with the off-line card punch. The off-line printer can be used to print records up to 20 words long (containing 120 characters). A switch sets the printer to single or double space, or to program control. In the latter case the first character is not printed but is used to control spacing on the printer.

Subroutines for Input and Output

Coding for the input and output devices is undoubtedly the most difficult aspect of 704 coding. Many subroutines for handling input and output are described in the SHARE distribution literature. The coder is urged to use them whenever possible.

CHAPTER XII
OVERFLOW, UNDERFLOW, AND MISCELLANEOUS TOPICS

Overflow, Underflow and You

It is an important feature of the 704 computer that a program normally halts only when either of two explicit instructions (HTR or HPR) is executed. There are three exceptions, however.

The first exception is the "dynamic stop." (An extreme example of the dynamic stop is an unconditional transfer instruction which has its own location as an effective address.)

The second exception is that the computer may stop on a read-write check. This occurs when a program instructs input or output equipment to read or write at improper times.

The third exception is the fixed-and floating-point divide-or-halt instructions (DVH and FDH) which stop the computer whenever the dividend and divisor do not satisfy certain conditions (e.g. the divisor must not be zero).

The divide-or-halt instructions adequately protect the programmer from making an incorrect or meaningless division.

Instructions, like DVH and FDH, which contain a protective (alarm) stop provide examples of what arithmetic instructions in most older computers were like. Thus in many computers a fixed-point addition yielding too large a result caused a computer stop (called an overflow alarm). Clearly this type of instruction might be desirable if absolutely no overflow should ever occur in a problem.

Alternatively there are programs in which overflows can be ignored or in which, when overflows occur, an alternative procedure can be provided. In this case an alarm stop is just what the programmer does not want. The 704 allows the coder this more general facility through the mechanism of the overflow lights and the transfer-on-overflow instructions. In doing so, however, it shifts a heavy burden of responsibility from itself to the coder. The computer can compute wrong answers at a very rapid rate.

If all coders were to follow every 704 arithmetic instruction by an alarm test instruction then clearly no undetected erroneous arithmetic could be done by the 704. But to begin with, this is inefficient and besides, experience has shown that coders dislike to use two instructions when they suspect that one will do. Moreover, the naive coder will argue that no tests are needed for unexpected alarms since any oversights will turn up in checking-out the program. Unfortunately nothing could be further from the truth, for unlike the usual coding mistake which produces an easily noticeable discrepancy in nearly all of the results, an undetected overflow condition may only occur sporadically and perhaps not at all for the chosen test cases. Thus if the coder neglects to query the computer about alarm conditions his later results can be completely wrong. In fact one would wonder why he went to so much trouble, took so much time, and used such an expensive machine only to get some meaningless numbers. Needless to say the situation is aggravated if the numerical results are used as the basis of an article in a research journal.

As it may have been suspected, the purpose of the preceding paragraph has been to thoroughly jolt the reader into realizing that the non-stop instructions of the 704 create serious coding complications. However by clear thinking and orderly procedures, which will be described in the remainder of this section, it will be seen that these problems can be minimized.

The fundamental philosophy which every good coder must follow is not to gamble needlessly with the computer. This is not to say that using a computer does not involve many risks but the goal should be to keep all risks calculated and limited. In particular, the calculated risks are: the reliability of the computer instructions performing as specified; the reliability of card readers, tape units, drum units, punches and printers; the possiblity of a mistake in programming (i.e. method); the possibility of a mistake in coding. The important thing to note is that in all of these risks, the probabilities can be estimated and can always be kept within desired bounds by a multitude of strategems. (Normally these probabilities are not explicitly discussed by coders but instead are replaced by descriptions of the various double-checks and tests which the careful worker uses to convince himself and his audience that he is doing a correct calculation.) Thus the role of the good coder bears an analogy to that of the good laboratory experimenter who must similarly suspect his own equipment until he convinces himself of its worth.

Having spent so much time exposing the problems of undetected arithmetic mistakes, it is now pertinent to discuss the techniques of dealing with these problems on the 704. Treating first the fixed-point instructions, there are only two possible arithmetic mistakes, both of which have already been casually mentioned. These are the overflow and the divide-error conditions which are associated with the AC overflow light and the divide-check light, respectively. These indicators may be queried and turned off by means of the test instructions TOV, TNO and DCT.

Every fixed point calculation can be decomposed into sequences of instructions which correspond to three cases:

1) Alarm conditions are not meaningful and should be ignored.
2) Alarm conditions may occur and require special treatment.
3) Alarm conditions should never occur but if they do, the program should be stopped.

The first case is trivial in that nothing need be done. In the second and third cases the pertinent indicator lights should be turned off before the sequence of instructions in question and after the sequence the indicators should be queried so that appropriate action is taken if alarm conditions occured. Thus the treatment of alarm conditions with fixed-point instructions is straightforward. It is worthy of note that there is only one fixed-point divide instruction which turns on the divide-check indicator (and does not stop the computer), DVP, and there are only 8 fixed-point instructions which can turn on the AC overflow indicator:

ADD, ADM, SUB, SBM, RND, ALS, LLS, LGL.

We finally consider the alarm conditions associated with the floating-point instructions. These are more difficult to deal with. The electronics of the 704 are so arranged that it is not possible for the fractional part of a floating-point result to overflow since the exponent is always adjusted to prevent this; however the characteristic (i.e. the exponent of 2 plus 128) has only 8 bits allotted to it and thus has a limited range. In particular, a characteristic which is too large (>255) is called an overflow and one which is too small (<0) is called an underflow. Inasmuch as all of the 8 floating-point instructions produce a double-register result, it is possible to get a wide variety of possibilities. The indications of these possibilities are left in an extraordinarily clumsy form in the arithmetic element and the situation represents a major weakness in the 704 design. (The situation is usually referred to as "the underflow-overflow problem" and has been a topic of protracted discussion among 704 users.)

The alarm indicators which are turned on by floating-point instructions are three: the divide-check light, the AC overflow light and the MQ overflow light. The divide-check light works as it did for the fixed-point instructions.

The AC and MQ overflow lights indicate that the AC or MQ register, respectively contain an incorrect result. The exact conditions are given by the following two charts. It is assumed that all indicators were off before each instruction and that on and off are signified by one and zero, respectively.

Instructions FAD, UFA, FSB, UFS, FMP, UFM

AC Indicator	MQ Indicator	Q bit in AC	Condition of: AC	 MQ
0	0	0	ok	ok
0	1	0	ok	underflow
1	0	0	overflow	ok
1	1	0	overflow	overflow
1	1	1	underflow	underflow

Instructions: FDH, FDP

AC Indicator	MQ Indicator	Range of Characteristic in MQ	Condition of: AC	 MQ
0	0	0 - 255	ok	ok
1	0	0 - 154	underflow	ok
1	1	129 - 255	underflow	underflow
0	1	129 - 255	ok	underflow
0	1	0 - 128	ok	overflow

The remainder of this section will deal with three procedures for utilizing the information in the above tables. The first procedure and by far the simplest is to consider all floating-point overflows and underflows to be cause for stopping the program. In this case, the program need only be analyzed into sections where the sequence of floating-point instructions is uninterrupted by any of the 8 fixed-point instructions which might turn on the AC indicator. Then for each sequence, it is only necessary to be sure thatboth the

AC and MQ indicators are off before the sequence and that the program is stopped if either indicator is on after the sequence. This is done with the instructions TOV, TNO and TQO.

It is to be emphasized that the above procedure is the minimum that should be done for any floating-point program. Unfortunately the procedure is often inadequate for the following reason. Most problems are described in a way which biases the exponents of the intermediate results. Thus a power series which converges well for all x less than a fixed value, will for small x contain terms which are extemely small. For example, in the polynomial

$$f(x) = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!}$$

for $x = 10^{-11}$, the last term is approximately 10^{-46}. In computing the last term an underflow would be obtained, so that it is clear that the appropriate corrective procedure (if the calculation were done this way) is to set the last term to zero. (It is tempting to ignore the meaningless underflowed term but because of the way the 704 was designed, an underflowed register when stored becomes a very large number which yields a gross error.) With this bias in mind, the most frequently desired case is single-register precision (i.e. the result is in the MQ after FDH or FDP and in the AC for all other floating-point instructions) where if overflow occurs, the program is stopped, and if underflow occurs, the result is replaced by zero. This prescription forms the basis of the next corrective procedure.

Again one analyses the program into sequences of instructions that contain no fixed-point instructions which might turn on the AC overflow light. At the beginning of each sequence both the AC and MQ overflow lights are turned off, for example, by the instruction TSX RESET, 4, where the subroutine RESET is:

```
RESET  TQO  * +1
       TOV  * +1
       TRA  1,4
```

(Here the asterisk in an instruction address designates the location of the instruction itself.) After every floating-point division instruction, (i.e. FDH, FDP) one uses the instruction, TSX DVTST, 4; similarly after every other floating-point instruction, (i.e. FAD, UFA, FSB, UFS, FMP, UFM), one uses the instruction, TSX AMTST,4. Two subroutines suitable for single-register precision arithmetic are the following:

```
AMTST   TQO   *+1
        TNO   1,4       Normal return
        ARS   1
        PBT
        TRA   ALARM     Overflow
        PXD   0,0
        TRA   1,4       Underflow return
DVTST   TQO   DV1
        TNO   1,4       Normal return
        TRA   1,4       MQ ok return
  DV1   TOV   DV2
        PXD   0,0
        LLS   8
        SSP
        SUB   DV3
        TMI   ALARM     Overflow
  DV2   LDQ   DV4
        TRA   1,4       Underflow return
  DV3   PZE   129       Integer 129
  DV4   PZE             Zero

 ALARM  HTR   ALARM     Overflow stop
```

There are several features to be noted about the above procedure. First, the two indicators are automatically turned off after each floating-point operation in anticipation of the next instruction. Second, the normal case of neither underflow nor overflow requires three operating instructions which raises the minimum floating-point operation times from 7, 17, and 18 to 13, 23, and 24 machine cycles. Third, the above procedure uses a valuable index register which may require further slowing down of the program.

The third and final overflow-underflow procedure accomplishes the same corrective action as the second procedure just described. However by a clever use of the trapping mode, the use of an index register is eliminated and the normal overflow-underflow test time is reduced from 6 to 2 machine cycles.

The 704 trapping mode is a special state of the computer (entered and left by the instructions ETM and LTM, respectively) in which all instructions perform as usual except for the transfer part of all transfer instructions. In the latter case, the location of the transfer instruction is set into the address section of location 0 and the computer control is transferred to location 1. (One transfer instruction, TTR, is immune to the trapping mode.) The usual use of the trapping mode is to trace the flow of control by means of diagnostic programs for trouble-shooting incorrect programs. In the present application, however, the trapping mode serves as a device to save the return point when control is transferred due to the AC or MQ overflow lights being on after a floating-point instruction.

The procedure again consists of first analysing the instructions into sequences where the indicators are not turned on except by floating-point instructions. Somewhere before the first sequence, a brief initializing program must be given.

This is

```
            CLA   WORD
            STO   1          Initialize analysis transfer
```

where

```
      WORD  TTR   TEST
```

Now it is not necessary to turn off both the indicators before entry into each sequence. This follows from the fact that for single-register precision the condition of the MQ indicator and the characteristic of the MQ are sufficient for testing the result of division, whereas the AC indicator and the Q bit are sufficient for testing the non-division instructions. Thus all that is required is that before every run of non-division floating-point instructions (i.e. no intervening divisions) the AC indicator is turned off and similarly before a run of division instructions the MQ indicator is turned off. This is done, of course, by either TOV*+1 or TQO*+1. Finally after every floating point division one gives the instruction, TQO TSTDV, and correspondingly after the non-division instructions, TOV TSTAM, where both instructions should be executed <u>in</u> the trapping mode. Inasmuch as these instructions will not trap unless there was an underflow or overflow, the increase in time of normal floating point operations is only 2 machine cycles. In addition one places somewhere in the program the following analysis subroutine:

```
      TEST  STO   T5         Save AC
            ARS   1
            SLW   T6         Save Q bit
            CAL   0
            STA   T1         Set pick-up
            ACL   T9
            STA   T4         Set return
        T1  CAL              Pick-up transfer instruction
            STA   T2
            CLA   T5         Restore AC less P,Q bits
        T2  TTR              Execute analysis or transfer
```

The subroutines for alarm testing are the following:

```
TSTAM  LTM
       CLA  T6     Q bit in S bit position
       TPL  ALARM  Overflow
       PXD  0,0    Clear AC
       TRA  T3     To return
TSTDV  LTM
       PXD  0,0
       LLS  8
       SSP
       SUB  T7
       TM1  ALARM  Overflow
       LDQ  T8
   T3  ETM
   T4  TTR--       Underflow return
   T5  PZE         AC Store
   T6  PZE  1      Q bit store
   T7  PZE  129    Integer 129
   T8  PZE         Zero
   T9  PZE  1      Integer 1
ALARM  HTR  ALARM  Overflow stop
```

Inspection of the above analysis routine reveals that any transfer instruction encountered while in the trapping mode will be interpreted correctly (except an indexed transfer, e.g. TRA1,4) although its execution time will be multiplied by a factor of 13. (The P and Q bits will also be cleared but this usually does not affect anything.) Consequently any conditional transfer instruction which is unlikely to meet the conditions of transfer may, if it is convenient, be executed in the trapping mode without any great loss in program operating speed; otherwise, it is desirable to leave the trapping mode before a probable transfer and reenter the trapping mode again after the transfer.

Finally the following sample program is given to illustrate the technique of the last procedure. Here it is assumed that DATA1 through DATA7 are the initial addresses of data blocks which along with the analysis subroutines just described are available elsewhere in the program. The sequence of control transfers will be left as an exercise for the reader

```
START  CLA WORD               Initialize in location 1
       STO 1                   trap transfer to analysis
                               routine
       TRA WORD+1
 WORD  TTR TEST
       LXA COUNT,1
 LOOP  CLA DATA1+100,1
       TOV *+1                Turn-off probably-on AC
                              indicator
       FAD DATA2+100,1
       ETM                    Enter trapping mode for test
       TOV TSTAM              Trap if underflow or overflow
       FSB DATA3+100,1
       TOV TSTAM              Trap if underflow or overflow
       LTM                    Leave trapping mode for
                               probable transfer
       TMI SKIP
       ETM                    Enter trapping mode for test
       TQO *+1                Turn-off probably-off MQ
                              indicator
       FDH DATA4+100,1
       TQO TSTDV              Trap if underflow or overflow
       TOV  *+1               Turn-off probably-off AC
                              indicator
 SKIP  ETM                    Enter trapping mode in case came
                              from TM1
       FMP DATA5+100,1
       TOV TSTAM              Trap if underflow or overflow
       TQO *+1                Turn-off probably-off MQ
                              indicator
       FDH DATA6+100,1
       TQO TSTDV              Trap if underflow or overflow
       LTM                    Leave trapping mode for
                              probable transfer
       STQ DATA7+100,1
       TIX LOOP,1,1
  END  HTR END
COUNT  PZE 100
```

A subroutine for dealing with floating-point overflows and underflows (CLOUD1) has been distributed by SHARE (distribution No. 248).

The Sense Switches

There are six switches on the operator's console (numbered from left to right by the digits 1 to 6) whose positions (either up or down) can be sensed by using the instructions

SWT n: Sense Switch Test n= 1,...,6

Sense switches can be used by the computer operator to modify the effect of a program while it is running. Both the SHARE Assembly Program and the MIT Post-Mortem Program use sense switches. For example, SAP will read-in the symbolic deck from the card reader if sense switch 1 is down but will read it in from a magnetic tape unit if sense switch 1 is up.

The Sense Lights

There are four lights (numbered from left to right by the digits 1 to 4) on the operator's console which can be turned on and off, or tested, by means of the following instructions:

1) SLN n: Sense Light On n = 1,2,3,4
2) SLF : Sense Lights Off
3) SLT n: Sense Light Test n = 1,2,3,4

1) The instructions, SLN n, turns on the sense light numbered n.

2) The instruction, SLF, turns off <u>all</u> of the sense lights.

3) The instruction, SLT n, turns off the sense light numbered n and skips one instruction if it was on.

SLT n: Sense light off and
Sense light n on $\Rightarrow$ C(ILC) +2 $\rightarrow$ C(ILC).

The sense lights can be used as a visible means to convey information to the operator about the state of the program. The MIT Post-Mortem Program uses a sense light as follows: By using certain sense switches an operator can stop the post-mortem program and insert manual post-mortem requests into the MQ register which the program will then execute. If, however, the operator inserts an illegal request the program will detect this illegality and return to the original stopping point with a sense light on.

Example: The following example considers the four sense lights to be a four digit binary counter (with the convention that a sense light being on denotes a 1). The example is a subroutine which increases the contents of this counter by 1 each time it is entered.

```
COUNTR  SLT  4
        TRA  FOUR
        SLT  3
        TRA  THREE
        SLT  2
        TRA  TWO
        SLT  1
        TRA  ONE
        TRA  1,4
   ONE  SLN  1
        TRA  1,4
   TWO  SLN  2
        TRA  1,4
 THREE  SLN  3
        TRA  1,4
  FOUR  SLN  4
        TRA  1,4
```

<u>A Quick Look at the Console</u>

The operator's console on the 704 contains an assortment of switches, buttons and lights most of which rarely concern the coder. A detailed description of the console may be found in the 704 manual on pp. 13-15. A few of the buttons are important to the coder, however and are briefly described below.

The Clear Button: When a binary deck is loaded into the 704 words are placed in registers specified by the coder. If the coder does not specify the contents of a register it will not be changed (and will contain whatever the previous user left in it). It is thus important that a coder fill all registers whose initial values affect his program.

The clear button (if pressed by the operator before loading begins) enables the coder to start with memory in a known state (namely all zeros). This button also resets all of the registers and alarm lights in the arithmetic element.+ It is recommended that the coder clear memory whenever possible since this will make simpler the interpretation of post-mortem results.

The Start Button: The start button can be pushed by the operator after the computer has stopped on one of the following instructions

HPR, HTR, DVH, FDH

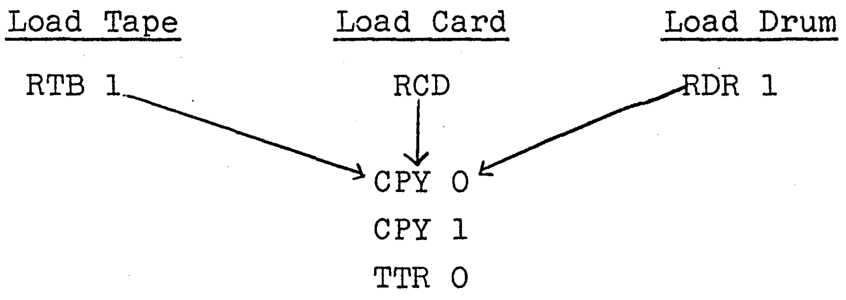
With HPR, DVH, and FDH its effect to start the computer on the next instruction in sequence. An exception is provided by the instruction

HTR x : Halt and Transfer

This instruction stops the computer in such a way that if the start button is subsequently pressed then the computer begins with the instruction in register x.

+ Another button, the reset button, clears the arithmetic element but does not change memory.

<u>The Load Buttons</u>: There are three buttons; called the load tape button, the load card button and the load drum button; which are used to initially bring information into core memory (e.g. after core memory has been cleared). These (when pushed by the operator) start the computer and cause it to initially execute one of the following sequences of instructions:



```
Load Tape        Load Card        Load Drum

 RTB 1             RCD              RDR 1
                  CPY 0
                  CPY 1
                  TTR 0
```

The sequence thus places words (from the outside world) into registers 0 and 1 and transfers control to the first of these. This simple sequence suffices to bring in more complicated loaders which can read binary cards into memory. In the next section we describe one such loader.

<u>A Binary Loader</u>

The binary cards produced by SAP during asembly can be described as follows

1. <u>Data Cards</u>
   - a) The decrement of the 9 left row contains the number of words on the card (call this n).
   - b) The address of the 9 left row contains the the location of the first word (call this x)
   - c) The 9 right row contains the check sum (which is the logical sum of all other words on the card).
   - d) Rows 8 left, 8 right, ...contain the n words to be stored in locations x,x+1,...x+n.

2. Transfer Cards
   a) Transfer cards are characterized by the fact that the decrement of the 9 left row is zero.
   b) The address of the 9 left row contains the starting address of the program.

Many more or less complicated loaders have been written by SHARE members and are described in the SHARE distribution material. The one listed below (NYBL1) is a relatively simple one which consists of a single card.

```
     ORG  0
     LXA  0,4     ]  These words are copied to 0 and 1 by
A    CPY  2,4     ]  the load card sequence
9R   TXI  A,4,-1  ]  This copy loop brings the rest of the
      |              card to core memory and terminates on
      ↓              the end-of-record skip
9L   PZE          ]  Used to store 9 left row.
     LTM          ]  We leave the trapping mode (just in case)
      |              and enter the loader proper.  The TXI in
      ↓              register 9R is right in any case.
RCD  RCD          ]  Read binary card
     CPY  9L      ]  x and n to 9L and MQ
     LLS  17      ]  n from MQ to AC
     PAX  0,4     ]  n to IR4
     ADD  9L      ]
     STA  TRA     ]  x + n to TRA, CPY and ACL
     STA  CPY     ]
     STA  ACL     ]
TRA  TXL  0,4     ]  exit to x if transfer card
     CPY  9R      ]  Check sum to 9R
     CAL  9L      ]
CPY  CPY  0,4     ]  Words to x, x+1,...,x+n-1
ACL  ACL  0,4     ]  Form a new check sum
     TIX  CPY,4,1 ]
```

```
SLW  9L       New check sum to 9L
CLA  9L
SUB  9R
TZE  RCD      Stop if checks sums disagree
HTR  RCD
END
```

M.I.T. Computation Center
Massachusetts Institute of Technology
Cambridge 39, Massachusetts

Subject: DESCRIPTION OF THE SHARE ASSEMBLY PROGRAM FOR THE I.B.M. 704 COMPUTER

To: Prof. Philip M. Morse, Director

From: Dr. Fernando J. Corbató

Date: April 15, 1957

PREFACE

The following memorandum is a description of the standard coding language and corresponding translation (i.e., assembly) program agreed upon by all members of the SHARE organization, an organization consisting of most of the users of the I.B.M. 704 computer. The memorandum is basically a copy of one written by Roy Nutt of the United Aircraft Corporation, dated March 22, 1956, which may be found in the appendix section of the SHARE Reference Manual. However, in the present version several minor clarifications and corrections have been added as well as a major revision made of the section on *arithmetic expressions*. In addition, an introduction has been added for those not familiar with the purpose of an assembly program. For convenience, a list of the acceptable instruction codes is included since some of the allowable input-output instruction abbreviations (as well as the CAC and STZ instructions) are not yet given in the current I.B.M. 704 manuals. Finally, a description is included of the format used for the absolute and relocatable binary cards which are produced as output by the assembly program.

Fernando J. Corbató

Fernando J. Corbató

INTRODUCTION

The SHARE Assembly Program (SAP) is a program which enables one to use the I.B.M. 704 computer as a special-purpose translation machine. This translation process consists of transforming ordinary 704 computer programs from a convenient coding language to the explicit binary number language that the 704 computer truly uses. It is the exact form and rules of this coding language that will be described in the following sections. The binary number conventions and nomenclature that are to be used are those described in the I.B.M. 704 Manual of Operation.

The manner in which SAP is used is the following: The programmer first prepares his program in symbolic cards using an I.B.M. key punch (with the special characters required by the 704 system), and the conventions described in this memo. This symbolic program is then processed on a 704 computer using SAP. The resultant output of this processing consists of binary cards which (at the discretion of the programmer) essentially may be of two possible forms: absolute or relocatable. The binary cards may then be read into a 704 computer very simply by means of any one of several binary card loader programs.

An absolute binary card loader program can normally be assumed to be available at the computer so that the loader program need not concern the programmer except in that it will occupy in the order of the first 100 cells of core memory storage. Thus, as a normal practice a programmer should never attempt to place a program in the first few hundred cells of core memory storage.

Relocatable binary cards, which are similar to absolute binary cards, are used whenever there are several applications for a program (or a portion of a program), and it is desired to locate the program (or portion) at different storage locations in the different applications without translating more than once. The most frequent case is that of subroutines which may be used in many different programs. It should be pointed out that it is not necessary ever to use relocatable binary cards as long as one always translates an entire program from symbolic cards to absolute binary cards. In other words, relocatable binary cards merely offer a short cut, the efficiency of which depends on the manner in which a given 704 installation is normally operated.

The coding language which SAP accepts consists of three-letter instruction abbreviations with references (i.e., addresses, tags and decrements), which are arithmetic expressions composed of symbols and/or decimal integers. The exact instruction abbreviations are those agreed upon by SHARE (usually the same as those of the I.B.M. 704 Manual) and are given in the appendix.

The symbols, which are used in making *references*, are integer quantities, essentially arbitrarily labelled by the programmer by means of combinations of letters and numbers which must be uniquely defined at some place in a given program. Symbols are usually used for two purposes: 1) to designate the *location* of a particular instruction in a program to which reference is to be made; and 2) to flexibly designate a frequently-referred-to *parameter* which is unchanged during the operation of a program but which may change with different applications of a program (e.g., the order of a matrix in a matrix multiplication program, or the number of an index register in the tag section of an instruction). In the parameter usage of symbols, of course, a new translation (i.e., assembly) has to be made in order to change the values of the parameters involved.

The SHARE Assembler

Consisting of Programs: UA SAP 1 and UA SAP 2

by

Roy Nutt
United Aircraft Corporation

with minor revisions by

F.J. Corbató
M.I.T. Computation Center

704 instructions to be assembled by this program are written with references expressed as arithmetic combinations of symbols and/or decimal integers. A variable field format is used in which the parts of the instruction are given in the order: address, tag and decrement. In addition to instructions, data in decimal, octal or Hollerith (BCD) form may be assembled, and library routines written in the same symbolic form may be conveniently incorporated into the program being assembled.

A program to compute

$$P_N\ (x,y) = \sum_{j=0}^{N} \left[\sum_{i=0}^{i+j=N} a_{ij}x^i \right] y^j$$

is used as an example of the use of this assembler and of the printed program listing which is an output of the assembler. (See page 18.)

In order to describe the use of this assembly program, let us consider first a simplified explanation of symbolic assembly operation.

The assembly procedure is divided into two parts; in the first, the UA SAP 1 program examines the particular program to be assembled in order to define each symbol used in writing the program. In the second part, the UA SAP 2 program prepares the actual machine language program, punches it in binary form on cards and produces a printed copy of the program in symbolic form together with the corresponding printed octal machine language program.

During the first part, a counter is used to specify the absolute location of each word in the program. Call this location counter L. L is set initially to an integer supplied to the assembly program by the program being assembled; henceforth L is increased by one for each word to be used by the program.

Simultaneously with this counting procedure a table is constructed. Each entry in this table defines a symbol used in the program as being equivalent to some integer. Entries to the table are made in two ways:

1. A symbol appears as the "symbolic location" of a word in the program being assembled and is assigned the value of L.

2. A symbol is defined by a pseudo operation. (All symbol values are taken modulo 2^{15}. If a symbol is defined as a negative value, the 2's complement, modulo 2^{15} will result, e.g., if N is defined as -6, a value of $N=2^{15}-6$ will be used in assembly.)

It is important to note that the order of the absolute instructions produced by symbolic assembly is determined solely by the order in which the symbolic instructions are read by the assembly program (i.e., by the physical order of the symbolic cards).

During the second part of the assembly process, L is computed in exactly the same manner as it was during the first part. In addition, all symbols in the symbolic program are replaced by the integer equivalences given in the table formed during the first part, thus producing an absolute program.

Note that this operation requires that each symbol be uniquely defined.

For use in the assembly program, the following definitions are made:

Symbol: Any combination of not more than 6 (but usually not more than 5) Hollerith characters, none of which is + -*/,$ and at least one of which is non-numeric. (Note: For this purpose, any character without a zone punch is numeric. Thus the = (8-3 punch) and the - (8-4 punch) are numeric.)

Integer: (with respect to instructions)
Any decimal integer less than 1,000,000.

The operation part of each instruction is specified by the standard "SHARE" abbreviation of 3 alphabetic characters.

Ordinarily a storage cell location should be identified by a symbol ("symbolic location") if and only if it is necessary to refer to this location in the program.

The address, tag and decrement parts of symbolic instructions are given in that order. In some cases the decrement, tag or address parts are not necessary; therefore the following combinations where OP represents the instruction abbreviation are permissible:

```
OP
OP    Address
OP    Address, Tag
OP    Address, Tag, Decrement
```

Examples of the last three types occur in the illustrative problem at P3, P3+2, and P3+3, respectively.

Note that the tag, if present, must be separated from the address by a comma and, similarly, the decrement, if present, must be separated from the tag by a comma. For the few instructions which require a tag but no address, the address zero should be used; for example:

```
PDX 0,4
```

Similarly, where a decrement is required with no tag, a zero tag should be used, as in

```
TXL A,0,B
```

The following card format is used by the assembly program:

Columns	Contents
1-6	Symbol or blank
7	Blank
8-10	Abbreviated operation or blank
11	Blank
12-72	Variable field
73-80	Not used

Expressions defining the address, tag and decrement are punched without blanks from column 12 on. The first blank to the right of column 12 defines the end of the instruction. All punching to the right of such a blank is considered to be a remark made by the programmer for his personal convenience and has no effect on the assembly process.

If an instruction requires a symbolic location, the symbol used is punched in column 1-6.

Arithmetic expressions

As stated before, the references which may be used in instructions can consist of arithmetic expressions of symbols and/or decimal integers. The arithmetic operations allowed in these expressions are addition, subtraction, multiplication, and division designated by + (12 punch), - (11 punch), *, and /, respectively. However, no parentheses are allowed, so that it is necessary to specify how the evaluation of an expression is done.

All of the arithmetic operations are done with 35 binary place integral arithmetic (i.e., modulo 2^{35}). In the case of division, only the integral quotient is retained, and the non-integral remainder (of the same sign as the quotient) is discarded. The evaluation of an arithmetic expression then proceeds as follows: Each segment of the expression where a segment is that portion of the expression from a + or - sign (or the beginning of the expression) to the next + or - sign (or the end of the expression) is separately evaluated from left to right with the consecutive multiplications and divisions being performed as specified; as each segment of the expression is evaluated,they are combined from left-to-right as indicated by the connective + and - signs.

As an example of the above procedure, the expression

$$A+200/15/6*15-B/C*D$$

is taken to have a meaning of

$$A + \left[\frac{\left[\frac{200}{15}\right]}{6}\right] \times 15 - \left[\frac{B}{C}\right] \times D$$

where the brackets denote the integer division described above.

Finally, if the result of an expression is to be expressed in n binary places, its magnitude is computed modulo 2^n. This quantity is taken to be the result unless the expression is negative, in which case the 2's complement is taken as the result.

Hence, if v is the value of an expression, r is the result used and

$$m = |v| \bmod 2^n$$

then

$$r = \begin{cases} m & v \geqslant 0 \\ 2^n - m & v < 0 \end{cases}$$

For example, the instruction at location P3+3 in the illustration has a decrement part of -1. Here m=1, v=-1, n=15, so that

$$r=2^{15}-1.$$

Consider also the tag part of instruction P4-1 where

$$v=J+K=1+4=5$$

$$m=5,\ n=3$$

so that $r=5$

The decimal integers which are allowed in constructing expressions are limited to values less than 1,000,000. The symbol values which are allowed are those integers less than 2^{15}, all larger values being taken modulo 2^{15}.

If symbol is given a negative value, it is "negative" in the sense that the 2's complement (modulo 2^{15}) is taken of the magnitude of the value. For example, if one states that N= -6, then the value that SAP will use for the symbol N will be $2^{15}-6$.

PSEUDO-OPERATIONS

In the following descriptions, a pre-defined expression is an arithmetic expression in which all the symbols used must have been previously defined, i.e., appeared in the symbol field, columns 1-6, of some preceding instruction or pseudo-instruction card.

Origin specification: ORG

The location counter L is set to the value of the pre-defined expression appearing in the variable field.

If no origin specification is given for a program, the initial value of L will be zero.

Origin specification instructions may be used at will.

Equals: EQU

The symbol appearing in 1-6 is assigned the integer value given by the pre-defined expression appearing in the variable field.

Note that the pseudo operation EQU is to be used only in those cases where the symbol appearing in columns 1-6 specifies a preset parameter such as the order of a matrix, the degree of a polynomial, the number of items in a group, or any other quantity which is invariant with respect to the location of the program in storage. If the symbol specifies the location of a piece of data or an instruction, the pseudo operation SYN should be used. The reason for this distinction is that symbols must be distinguishable as non-relocatable and relocatable whenever relocatable binary cards are produced as an output by the assembler.

Synonym: SYN

The symbol appearing in columns 1-6 is assigned the integer value given by the pre-defined expression appearing in the variable field.

Note that the pseudo operation SYN is to be used only in those cases where the symbol appearing in columns 1-6 specifies the location of a piece of data, the location of an instruction, or any other quantity whose value depends upon the location of the program in storage. If the symbol specifies a preset parameter, the pseudo operation EQU should be used.

Decimal data: DEC

The decimal data beginning in column 12 is converted to binary and assigned to consecutive locations L, 1+1, ...

Successive words of data on a card are separated by commas, and the first blank to the right of column 12 indicates that all punching to the right of this blank is a remark.

Signs are indicated by + or - (12 or 11 punch) preceding the number, the exponent or the binary scale factor. However, it is not necessary to use the + sign.

The symbol or absolute location appearing in columns 1-6 specifies the location of the first decimal data word on the card; the remaining data words are located consecutively after the first data word. If no symbol or absolute location appears in columns 1-6, the data words are located consecutively after the previous word of the program. (Note: the current version of SAP makes a mistake if an absolute location is used in columns 1-6 on a multiple word card. This mistake will be rectified.)

Briefly, binary integers (i.e., binary point just to the right of the 35th bit) may be denoted by just the integer values. For example: +7, -3, 7. Binary fractions with the binary point between the sign and the 1st bit are given in the form: +.23B, -.34B, .45B. Floating point numbers may be denoted by just writing the number in the ordinary way but with a decimal point as in the examples: +8., -9., 10. Since the exact rules for decimal numbers used by the assembler allow greater generality, they are now given.

If none of the characters . E or B appear in a decimal data word, the word is converted as a binary integer with the binary point at the right-hand end of the word.

If either of the characters E or . or both appear in a decimal data word and the character B does not appear, the word is converted to a 704 type floating binary quantity. The decimal exponent used in this conversion is the number which follows immediately after the character E. If the character E does not appear, the exponent is assumed to be zero. If the decimal point does not appear, it is assumed to be at the right-hand end. For example, 12.345, +12.345, 1.2345E1, 1234.5E-2, and 12345E-3 are all equivalent representations of the same floating point word.

If the character B appears in a decimal data word, the word is converted as a fixed point binary quantity. The binary scale factor used in this conversion is the number which follows immediately after the character B; this number being the number of binary places between the left-hand end of the storage cell and the binary point of the fixed point binary result. If the decimal point does not appear in the decimal data word, it is assumed to be at the right-hand end. The decimal exponent used in this conversion is the number which follows immediately after the character E. The order of B and E is not significant. For example, 12.345B4, +1.2345E1B4, and 12345B4E-3 are all equivalent representations of the same fixed point quantity.

It should be noted that in all cases decimal input which is too small and out-of-range (e.g., 1E-50) is replaced by zero. However, there is currently a mistake in SAP for exponents of ten from E-42 to E-49. This mistake will be fixed.

Octal data: OCT

The octal data beginning in column 12 is taken in binary integer form, the binary point considered to be on the right-hand end of a 704 word, and assigned to consecutive storage locations L, L+1, ...

Successive words are separated by commas and the first blank to the right of column 12 indicates that all punching to the right is to be considered a remark.

The symbol or absolute location appearing in columns 1-6 specifies the location of the first octal data word on the card; the remaining data words are located consecutively after the first data word. If no symbol or absolute location appears in column 1-6, the data words are located consecutively after the previous word of the program. (Note: the current version of SAP makes a mistake if an absolute location is used in columns 1-6 on a multiple word card. This mistake will be rectified.)

In the case of 12 digit octal numbers, the following equivalences exist with respect to the high order digit:

$$-0 \equiv 4 \qquad -1 \equiv 5 \qquad -2 \equiv 6 \qquad -3 \equiv 7$$

Either form may be used in coding for the assembly.

Hollerith data: BCD

Normally the 10 six-character words of Hollerith information from columns 13-72 are read and assigned to locations L, L+1, ..., L+9. If however, less than 10 BCD words are desired, a word count v ($0 \leq v \leq 9$) is punched in column 12, in which case v words are read and assigned to locations L, L+1, ..., L+v-1.

The symbol or absolute location appearing in columns 1-6 specifies the location of the first Hollerith word on the card; the remaining words are located consecutively after the first word. If no symbol or absolute location appears in columns 1-6, the words are located consecutively after the previous word of the program. (Note: the current version of SAP makes a mistake if an absolute location is used in columns 1-6 on a multiple word card. This mistake will be rectified. In addition, a BCD card of zero words currently creates a mistake during assembly. These SAP mistakes will be eliminated.)

Block started by symbol: BSS

The block of storage extending from L to L+N-1, where N is the value of the pre-defined expression beginning in column 12, is reserved by this pseudo operation.

If a symbol is punched in columns 1-6, it is assigned the value L, corresponding to the first word of the block reserved.

Finally, L is replaced by L+N.

Block ended by symbol: BES

This pseudo operation is exactly the same as BSS, except that the value assigned to any symbol appearing in columns 1-6 is L+N, corresponding to the location of the first word following the block reserved.

Repeat: REP

Two pre-defined expressions, the first beginning in column 12 and separated from the second by a comma, define two integers M and N. The block of instructions and/or data preceding the REP operation in locations L, L+1,...,L+M-1 is repeated N times, the repeated information being assigned to locations L+M, L+M+1,...,L+M* N+M-1. For example, if N is 1, then one obtains two identical blocks, the original block and the repeated block. Only one word of information may appear on each card which is part of a repeated block.

Library search: LIB

The library routine identified by the symbol in columns 1-6 is obtained from a library tape and inserted in the program being assembled. If the library routine required k words of storage, it will occupy locations L, L+1,..., L+k-1. The identification symbol is not entered in the table of symbols, but any symbols appearing in the library routine are entered and properly defined.

The first set of information on the library tape is an ordered list of the subroutines which are on the tape. The assembly program always keeps track of the position of the library tape and makes use of the information in the ordered list of subroutines in order to directly pick up the subroutines in the sequence that they are requested by a program. (The library tape is not rewound between library requests.)

Tape searching time may be minimized both by recording the most frequently used subroutines at the beginning of the tape and by specifying that the subroutines to be incorporated into any particular program are called for in the order in which they appear on the tape.

Heading: HED

It is often convenient to combine several programs into one program. Two difficulties immediately arise. First, the symbolic references to data common to the several programs may differ in the individual programs. This can be easily corrected by the use of synonyms which equate the proper symbols.

Second, it may be that two or more of the individual programs use the same symbols for references which should be unique. In order to restore uniqueness, it is necessary to change the symbols in each program in some way. The heading pseudo operation accomplishes this result in the following manner.

The heading card supplies to the assembly program a single character (punched in column 1 of the HED card). Any Hollerith character is permissible except zero, comma, plus, minus, asterisk, slash, or dollar sign. Each symbol in the program following the HED pseudo operation is prefixed by this character except when a special indication to cancel the prefixing operation is given. A new heading pseudo operation card will replace the prefix character. Thus several programs having non-unique symbols may be combined by giving the heading pseudo operation with a unique character before each program. If a numerical heading is used, then some non-numeric character must be punched in 2-6 of the heading card. (Note: Currently SAP does not function properly if the heading character is a numerical digit. This mistake will be fixed.)

It is, however, sometimes necessary to make cross-references between the individual programs. To accomplish this, such references must be written in the following way. Let H be a heading character and K be the symbol in the block headed by H to which reference is to be made. To refer to K (i.e., to use the value represented by K in an address, tag or decrement) in a part of the program not headed by H but by, say, J, write

H $ K .

The special character $ indicates to the assembly program that K is to be prefixed by H instead of by the prefix J given on the last heading card.

It is important to note that if use is to be made of the heading feature, all symbols used throughout the program will usually be restricted to five or fewer characters. If any six-character symbols (such as the erasable storage designation COMMON) are used, these symbols will not be headed.

Some additional remarks are that:

1) A $ B is not the same as AB.

2) A $ BCDEF is the same as ABCDEF.

3) OOOA, where O is zero, is the same as OA is the same as A.

4) A symbol in an unheaded portion of a program cannot be referred to from a headed portion of the program by the $ notation. Hence, in general the rule: head everything or nothing.

Define: DEF

If there exist in the program symbols not defined in accordance with the normal rules, such symbols may be defined in a different manner by use of the pseudo operation DEF. This pseudo operation causes the first such symbol encountered in an address, tag or decrement to be assigned the value given by the expression (which need not be pre-defined) beginning in column 12 of the DEF card. Successive undefined symbols are then given successive values until either a new DEF is given (in which case a new assignment is begun) or until the capacity of the symbol table is exceeded. A common application of this pseudo instruction is to reserve temporary storage locations in an automatic manner without having to explicitly assign each location. However, when this pseudo operation is used, the symbols so defined will still be included in the list of "undefined symbols" given by the assembler at the end of the program listing output.

Note that the pseudo operation DEF cannot be used to define an otherwise undefined symbol if this symbol occurs in the address, tag or decrement of an instruction which precedes the DEF card. The pseudo operation DEF defines only those otherwise undefined symbols which are first encountered after the DEF card itself has been encountered.

Similarly, if two DEF cards are used, and if an otherwise undefined symbol occurs both in instructions which appear between the two DEF cards as well as in instructions which follow the second DEF card, then the definition which will be used throughout is the one established by the first DEF card. The second DEF card has in such a case no effect on the already-established definition.

Remarks: REM

Any Hollerith punching in columns 12-72 will be reproduced in the printed listing of the assembly without otherwise affecting the assembly in any way. This is a useful feature for labeling and describing blocks of program since as many REM cards can be used as desired.

End of program: END

This pseudo operation must be the last read by the assembly program. The value of the expression beginning in column 12 is punched as the transfer address in a 704 binary correction transfer card.

Operation Code

Among the standard 3-letter operation codes adopted by SHARE, this assembly program recognizes the following codes which may be used to assign arbitrary values to the prefix and sign of calling sequence words:

Alphabetic Code	Name	Octal Code
MZE	Minus zero	-0000
MON	Minus one	-1000
MTW	Minus two	-2000
MTH	Minus three	-3000
PZE	Plus zero	+0000
PON	Plus one	+1000
PTW	Plus two	+2000
PTH	Plus three	+3000
FOR	Four	-0000
FVE	Five	-1000
SIX	Six	-2000
SVN	Seven	-3000

In coding symbolic instructions which have CFF, CHS, CLM, COM, DCT, ETM, IOD, LTM, LBT, PBT, RCD, RPR, RTT, RND, SLF, SPT, SSM, SSP, WTV, WPR, or WPU as their operation part, the address part should be blank or zero, since the assembly program automatically introduces the correct address.

In coding symbolic instructions which have BST, RDR, RTB, RTD, REW, SLN, SLT, SPR, SPU, SWT, WDR, WEF, WTB, WTD, or WTS as their operation part, the address part should be the unit number (in decimal). For instance, BST 2 implies Back Space Tape No. 2, SPR 9 implies Sense Printer Exit No. 9, WDR 3 implies Write Drum No. 3, and so on. The assembly program automatically computes the correct octal address (222, 371, and 303, respectively, in the foregoing examples).

Location counter

If an absolute decimal location (i.e.,: one containing no non-numeric characters) is punched in columns 1-6 of any card in the assembly, the location counter L will be set to that value. The effect of absolute decimal punching in these columns is therefore identically the same as if the card in question were to be placed immediately behind an ORG card having the exact same absolute decimal location punched in its variable field.

Operational features

As an aid to the programmer this assembly program gives some indications of erroneously prepared programs.

If a symbol used in the program is not defined, address, tag or decrement parts containing this symbol are left blank in the printed assembly, and zero is used for the corresponding address, tag or decrement parts in the binary instruction deck. In the case of pseudo operations involving undefined symbols, any expressions containing such symbols are evaluated using zero as the value of the undefined symbol. A list of all undefined symbols will be printed at the end of the assembly. (Included in this list will also be any symbols which have been defined by means of the pseudo operation DEF.)

If a non-existent operation code is used, the prefix part of the corresponding instruction is left blank in the printed assembly, and zero is used as the operation code in the binary deck.

A list of duplicated symbols is printed prior to the printing of the program. This list gives the symbol duplicated and the integer values assigned to it.

Other convenient features are:

Printing of the entire program listing may be suppressed or printing of the subroutines copied from the library may be suppressed.

Single or double spacing of the program listing is optional.

Assembly may be made from either a BCD tape (i.e., off-line operation) or from cards (i.e., on-line operation).

Binary punching is available in either absolute or relocatable format. (Currently there are several mistakes in SAP concerning the creation of relocatable binary cards. These mistakes, which are included in the appendix and were described in a United Aircraft Corporation memo of December 6, 1956, distributed at the December 1956 SHARE meeting, will be fixed.)

Capacity of the symbol table

Sufficient space has been set aside in a 4096-word core storage in order to permit the assembler to construct a symbol table containing 1097 entries. In cases where the program to be assembled makes use of the library tape, however, the maximum number of symbols which the assembler can handle is somewhat reduced. This follows from the fact that the entire ordered list of subroutines which forms the first set of information on the library tape is copied into the upper end of the symbol table area at the time that the first LIB card is encountered. Hence, if the library tape is used during an assembly, the effective symbol table size becomes 1097 minus the number of library subroutines on the tape.

In connection with the capacity of the symbol table, it should also be noted that any unassigned symbols are also recorded in this symbol table area preparatory to printing the list of unassigned symbols at the end of the assembly. Hence, if a case arises where the number of assigned symbols plus the number of subroutines in the tape library (if used) plus the number of unassigned symbols should total more than 1097, then the list of unassigned symbols printed at the end of the assembly will include only enough symbols to make up the 1097 total. The rest of the unassigned symbols can only be detected by noting blank addresses, tags or decrements in the printed output.

Reassembly features

Additions to a program which has been assembled are easily accomplished if the table of symbols which was punched during the initial assembly process has been saved. It is then necessary only to reload this table and assemble the new parts of the program. The original program need not be reloaded.

Furthermore, any change to the original program which does not involve relocation of any part of the program, or any reassignment of symbols, may be made by assembly of only those parts of the program which are to be changed.

Enlarged core storage

The assembler has been so written as to permit it to be used, without change, in 704's with enlarged core storage.

For each additional two words of core storage beyond the minimum of 4096, the assembler automatically provides for one additional symbol in the symbol table. (Note: Currently when SAP is used in a 4096-word machine, it produces a symbol table which cannot be used for correct reassembly in a 8192-word machine. This incompatability will be removed.)

Example of a Program Listing Obtained from SAP

```
                        04000       ORG 2048
04000 -0 53400 5 04011              LXD P1,J+K       INITIALIZE INDEX
                                                       REGISTERS
04001 -0 63400 4 04020  P4          SXD P2,K         STORE K
04002  0 50000 1 04022              CLA A+1,J        OBTAIN FIRST ELEMENT
04003  1 77777 1 04004              TXI P6,J,-1      X
04004 -2 00001 4 04017  P6          TNX P5,K,1       X
04005  0 76500 0 00043  P3          LRS 35           FORM POLYNOMIAL
04006  0 26000 0 04046              FMP X            IN X
04007  0 30000 1 04022              FAD A+1,J        X
04010  1 77777 1 04011              TXI P1,J,-1      STEP COEFFICIENT
04011  2 00001 4 04005  P1          TIX P3,K,1       TEST REDUCED K
04012  0 60100 0 04051              STO S            STORE PARTIAL SUM
04013  0 56000 0 04050              LDQ Z            FORM POLYNOMIAL
04014  0 26000 0 04047              FMP Y            IN Y
04015  0 30000 0 04051              FAD S            X
04016 -3 77754 1                    TXL OUT, J,      X
                                      -R/2+1
04017  0 60100 0 04050  P5          STO Z            X
04020  1 00000 4 04001  P2          TXI P4,K         X
                        00005  N    EQU 5
                        00052  R    EQU N*N+3*N+2
                        04021  A    BSS R/2
04046  0 00000 0 00000  X
04047  0 00000 0 00000  Y
04050  0 00000 0 00000  Z
04051  0 00000 0 00000  S
                        00001  J    EQU 1
                        00004  K    EQU 4
                        04000       END P4-1
                        00000   OUT
```

Note that the a_{ij}'s are stored in the order a_{05}, a_{14}, a_{04}, a_{23}, a_{13}, a_{03}, ..., a_{00} from location A on.

APPENDIX

The following is essentially an abstract from the SHARE Reference Manual, Section 03.1, Programming Standards.

B. Card formats

1. All cards (on-line or off-line) which contain 72 columns of information and 8 columns of identification are to be punched with the information in columns 1-72 and the identification in columns 73-80.

2. Relocatable binary information format

   For convenience, we shall use an abbreviated designation for various parts of the card. For example, the 13th bit position of the word in the left half of the 6th row would be denoted by 6L13. The decrement field of the same word would be 6LD. P, T and A stand, respectively, for prefix, tag and address. The sign bit is denoted by S.

   The 9L word is always the control word and the 9R word is always the 36 bit ACL check sum (denoted by CKS). The following list contains the various types of binary cards used.

   a) Absolute Data
   b) Relocatable Data
   c) Correction and/or Transfer
   d) Origin Table

   Detailed descriptions of these card types follow:

   a) Absolute Data

   Bits 9L13 to 9L17 contain the word count V. 9L21 to 9L35 contain the initial location R. All other positions in 9L are ordinarily blank. 8L, 8R, 7L, 7R,... contain the absolute data. The maximum word count is 22. If 9L2 is punched, the CKS is meant to be ignored, and no check is to be made against it. This applies also to a completely blank CKS.

   b) Relocatable Data

   9L1 is punched. 9L13 to 9L17 contain the word count V. 9L21 to 9L35 contain the nominal initial location R. All other positions in 9L are ordinarily blank. If 9L2 is punched, the CKS is to be ignored,as in the case of a completely blank CKS. The indicator bits are in the 8th row, starting from the left. The following one- and two-bit codes are used to indicate the type of field:

0	absolute field
10	relocatable direct field
11	relocatable complemented field

"Direct" here means uncomplemented. The string of these codes starts at 8LS and proceeds continuously to the right until it terminates. 7L, 7R, 6L, 6R,... contain the relocatable data words.

Let us, for illustration, suppose that 7LD is absolute, 7LA is relocatable direct, 7RD is absolute, 7RA is relocatable and complemented, 6LD is relocatable direct, 6LA is absolute, 6RD is absolute, and 6RA is relocatable complemented. Then the indicator bit pattern would be:

```
0    10    0    11    10    0    0    11
```

This may be condensed into:

```
010011100011
```

and this pattern is to be punched into the 8th row beginning with 8LS.

c) Correction and/or Transfer

(1) Correction

Rows 8 through 12 contain corrections which are entered in the following manner: The nominal location is punched in the LA field and the correction word itself in the right-hand word of the same row. If the location is to be adjusted by an increment (i.e., the correction word is to be relocated), then the L1 bit is punched. (Note that the L1 bit always indicates relocation). If a row is completely blank, it is ignored. The indicator bits for the decrement and address fields of the correction word are punched in the L3 to L5 bit positions, using the indicator scheme outlined in section (b). The sequence of correction entries is assumed to be from the 8th row upwards. If the L20 bit (LT3) is punched, then the nominal location is assumed to be 1 more than the preceding one. Hence, it is not necessary to punch every nominal location in a consecutive block, If this punch (L20) appears in the 8th row, however, it means that the nominal location is the one actually punched in 8LA. Hence, it is possible to load absolute zero at location zero.

If 9L2 is punched, the CKS is ignored. No punches at all need appear in the 9L word, or in the 9R word if there is to be no CKS comparison.

(2) Transfer

The contents of the 9LA field are taken to be the location to which control is to be transferred after all corrections have been loaded. If 9L1 is punched, then this nominal location is to be relocated in the usual manner.

d) Origin Table

Bit 9L12 is punched. If 9L2 is punched, the CKS is ignored as usual. Starting with the 8th row, the card contains a table of origins in the following format:

In each row, the nominal location which begins a region is punched in the LA field. The operating location (i.e., the final location of an instruction when it is actually to be executed) is punched in the RA field. If there is a loading location distinct from the operating location, this is punched in the RD field. If there is no loading location, then the operating location is used in place of it. The entries need not be punched in order of ascending nominal locations. If a row is completely blank, or if the L2 bit is punched in a row, then that row will be ignored. If L20 is punched in an otherwise blank row, then nominal zero will be set to absolute zero.

A general binary loader which fulfills these specifications if PKCSB4.

3. Symbolic instruction card - See main body of this memorandum or SAP description in appendix of SHARE Reference Manual (Section 10.03).

C. SHARE mnemonic operation codes

	ACL	Add and Carry Logical Word	0631
	ADD	Add	0400
	ADM	Add Magnitude	0401
	ALS	Accumulator Left Shift	0767
	ANA	And to Accumulator	-0320
	ANS	And to Storage	0320
	ARS	Accumulator Right Shift	0771
	BST	Backspace Tape	0764
*	CAC	Copy Add and Carry	-0700
	CAD	" " " "	-0700
	CAL	Clear and Add Logical Word	-0500
	CAS	Compare Accumulator with Storage	0340
	CHS	Change Sign	0760,002
	CLA	Clear and Add	0500
	CLM	Clear Magnitude	0760,000
	CLS	Clear and Subtract	0502
	COM	Complement Magnitude	0760,006
	CPY	Copy or Skip	0700
	DCT	Divide Check Test	0760,012
	DVH	Divide or Halt	0220
	DVP	Divide or Proceed	0221
	ETM	Enter Trapping Mode	0760,007
*	ETT	End of Tape Test	-0760,011
	FAD	Floating Add	0300
	FDH	Floating Divide or Halt	0240
	FDP	Floating Divide or Proceed	0241
	FMP	Floating Multiply	0260
	FSB	Floating Subtract	0302
	HPR	Halt and Proceed	0420
	HTR	Halt and Transfer	0000
	LBT	Low Order Bit Test	0760,001
	LDA	Locate Drum Address	0460
	LDQ	Load MQ	0560
	LGL	Logical Left	-0763
	LLS	Long Left Shift	0763
	LRS	Long Right Shift	0765
	LTM	Leave Trapping Mode	-0760,007
	LXA	Load Index from Address	0534
	LXD	Load Index from Decrement	-0534
	MPR	Multiply and Round	-0200
	MPY	Multiply	0200
	MSE	Minus Sense	-0760
	NOP	No Operation	0761
	ORA	Or to Accumulator	-0501
	ORS	Or to Storage	-0602
	PAX	Place Address in Index	0734
	PBT	P Bit Test	-0760,001
	PDX	Place Decrement in Index	-0734

* Not accepted by current version of SAP; SAP will be corrected to do so.

PSE	Plus Sense	0760
PXD	Place Index in Decrement	-0754
RDS	Read Select	0762
REW	Rewind	0772
RND	Round	0760,010
RQL	Rotate MQ Left	-0773
RTT	Redundancy Tape Test	-0760,012
SBM	Subtract Magnitude	-0400
SLQ	Store Left-Half MQ	-0620
SLW	Store Logical Word	0602
SSM	Set Sign Minus	-0760,003
SSP	Set Sign Plus	0760,003
STA	Store Address	0621
STD	Store Decrement	0622
STO	Store	0601
STP	Store Prefix	0630
STQ	Store MQ	-0600
* STZ	Store Zero	0600
SUB	Subtract	0402
SXD	Store Index in Decrement	-0634
TIX	Transfer on Index	2000
TLQ	Transfer on Low MQ	0040
TMI	Transfer on Minus	-0120
TNO	Transfer on No Overflow	-0140
TNX	Transfer on No Index	-2000
TNZ	Transfer on No Zero	-0100
TOV	Transfer on Overflow	0140
TPL	Transfer on Plus	0120
TQO	Transfer on MQ Overflow	0161
TQP	Transfer on MQ Plus	0162
TRA	Transfer	0020
TSX	Transfer and Set Index	0074
TTR	Trap Transfer	0021
TXH	Transfer on Index High	3000
TXI	Transfer with Index Incremented	1000
TXL	Transfer on Index Low or Equal	-3000
TZE	Transfer on Zero	0100
UFA	Unnormalized Floating Add	-0300
UFM	Unnormalized Floating Multiply	-0260
UFS	Unnormalized Floating Subtract	-0302
WEF	Write End of File	0770
WRS	Write Select	0766

* Not accepted by current version of SAP; SAP will be corrected to do so.

Extended Operations List

READ

RCD	Read Card Reader	0762,321
RDR	Read Drum	0762,301-310
RPR	Read Printer	0762,361
RTB	Read Tape - Binary	0762,221-232
RTD	Read Tape - Decimal	0762,201-212

WRITE

WDR	Write Drum	0766,301-310
WPR	Write Printer	0766,361
WPU	Write Punch	0766,341
WTB	Write Tape - Binary	0766,221-232
WTD	Write Tape - Decimal	0766,201-212
WTS	Write Tapes - Simultaneously	0766,321-325
WTV	Write CRT	0766,030

SENSE

SLF	Sense Lights Off	0760,140
SLN	Sense Light On	0760,141-144
SLT	Sense Light Test	-0760,141-144
SPR	Sense Printer	0760,361-372
SPT	Sense Printer Test	0760,360
SPU	Sense Punch	0760,341-342
SWT	Sense Switch Test	0760,161-166

OTHER

CFF	Change Film Frame	0760,030
IOD	Input-Output Delay	0766,333

The following paragraphs concerning mistakes made by SAP in producing relocatable binary cards are abstracted from the United Aircraft Corporation memorandum dated December 6, 1956, which was distributed at the December 1956 SHARE meeting:

> The transfer card produced at the end of a relocatable binary deck lacks the necessary relocatable control punch. This will be fixed.
>
> In computing the value of a compound address such as Y-3, where Y is relocatable and its equivalence happens to be 2, the Assembler indicates that the result (the complement of one) is to be relocated in complement fashion. Correspondingly, 3-Y comes out as one, relocatable direct. This is, of course, not correct. This error arises because the Assember uses the sign of the computed value of the compound address to determine whether relocatability is direct or complement. And, in general, this usually leads to a correct determination. There is already available to the Assembler, however, a quantity whose sign *always* correctly indicates whether relocatability should be direct or complement. This is the test quantity (RBITS) which is computed in order to determine whether the compound address in question is of such form as to permit of relocation at all. If this test quantity turns out to be +1, the address is relocatable direct; if -1, it is relocatable complement; if any other value, relocation is impossible. The Assembler will be changed to take advantage of this fact. As a result, in the above example, Y-3 will come out as the complement of one, relocatable direct, and 3-Y will come out as one, relocatable complement.
>
> In assembling onto relocatable cards, instructions referring to sense switches are not correctly tested for relocatability. Instead, they are in each case given the same relocatable bits as the instruction which they follow. This will be fixed.
>
> At present it is not possible to obtain a correct relocatable binary deck for a program whose origin is at location zero, unless an ORG 0 card is used. This will be changed in such manner that the ORG 0 card will not be necessary, although it will of course cause no trouble if it is included.

COMPUTATION CENTER
Massachusetts Institute of Technology
Cambridge 39, Massachusetts

TO: 704 Users
FROM: F. C. Helwig
DATE: July 24, 1957
SUBJECT: A USERS' ABSTRACT OF THE POST-MORTEM PROGRAM

Introduction

The MIT post-mortem program is a selective memory print-out or punch-out routine. The core memory ranges to be recorded and the word-forms to be produced are specified using either symbolic request cards or the computer console. This memo describes only request cards.[+]

The post-mortem program is recorded on the systems magnetic tape unit (MT1) and is entered by the load tape button or by a programmed load tape sequence:

```
(REW 1)
 RTB 1
 CPY 0
 CPY 1
 TTR 0
```

This initiates a self-loading sequence which brings the post-mortem program to memory and which saves (on MT5) the previous contents of the memory registers required for the post-mortem program. C(MQ), C(ILC) and the contents of memory registers 0-4 are destroyed by this process.

The deck of request cards to be processed must have been placed in the on-line card reader is read in by the post-mortem program.

[+] Control of the program from the computer console is described in CC-23.

(1) MT2 is required if results for off-line printing are produced.

(2) MT3 is required if results for off-line punching are produced.

The post-mortem program does not rewind MT2 or MT3 so that post-mortem results can be ganged with other output from a users' program.

Request Cards

The SHARE card format is used for request cards which are identified by the letters, PMR, in their operation field. The variable field of the card must contain four expressions separated by commas and terminated by a blank column.

The first two expressions give the initial and final addresses of the range in memory to be printed or punched. These can consist of any legal SAP expressions and may include symbols.[+] The user may also specify octal integers in such expressions by immediately preceding the integer with a division sign, e.g.,

$$/1000 = 1000_8 = 512_{10}$$

The user should note that this facility does not exist in SAP language.

The third expression designates the mode in which words are to be recorded and must be one of the following abbreviations

FLO	Floating-point numbers.
FIX	Fixed-point numbers.
INT	Integers (decimal).
SYM	Instructions with symbolic addresses.
ABS	Instructions with absolute addresses.

[+] The symbol, * , which in the new SAP may stand for the current location may not be used (in that sense).

OCT	Octal numbers
BCD	Binary-coded-decimal.
BIN	Absolute binary cards.

The output format implied by these various modes is identical to the input language used by the SHARE Assembly Program.

The fourth expression specifies the output device to be used in recording the words and must be one of the following abbreviations

NPR	On-line printer
NPU	On-line punch
FPR	Off-line printer
FPU	Off-line punch

The output produced on these devices will be identical in the sense that punched output will, if printed on an accounting machine, be identical to printed output.

If binary cards (BIN) are requested, they may, of course, be produced only by the on-line punch (NPU).

Scaling Fixed-Point Numbers

If fixed-point numbers (FIX) are requested the user may specify (if he wishes) a decimal scale-factor, x, and a binary scale-factor, y, where $0 \leq y \leq 35$, by writing

FIXExBy

to designate the mode. In this case the post-mortem program multiplies each number by

$$10^{-x} \cdot 2^{y}$$

before recording it and appends a suitable correction factor which in SHARE notation would be

ExBy

Thus the fraction, F, appears in the form

$$(F \cdot 10^{-x} \cdot 2^{y}) \text{ Ex By}$$

The special case

$$x = 0 \text{ and } y = 35$$

is detected by the program and such numbers appear as SAP integers.

Remarks Cards

Any information following the terminating blank column in the variable field of a PMR card is considered to be a remark and will appear as such immediately preceding the first line of output resulting from the request.

The user may also insert remarks cards (specified by the letters, REM, in the operation field) before PMR cards in the request deck. Each REM card is recorded just preceding the first line of output from the next PMR card in the request deck.

Remarks cards can be used to label results.

Termination Cards

Request decks must be terminated by a termination card consisting of one of the SAP instructions

TRA	x	$x \geq 5$
TTR	x	$x \geq 5$
HTR	x	

where x denotes any legal SAP expression.+

These cards cause the post-mortem program to restore core memory and the machine registers to their original contents (except for the MQ register, the ILC, and registers 0-4 of core memory) and to execute the designated transfer instruction.

+ See previous footnote.

Symbol Tables

If symbolic request cards are used or if instructions with symbolic addresses (SYM) are requested, a symbol table must be made available to the post-mortem program. This is done by preceding the request deck with the binary symbol table produced by SAP. Since these cards are read in by the post-mortem program they should not be preceded by a loader.

The Machine Conditions

The contents of the registers and indicators of the arithmetic element (except for the MQ and ILC) are recorded (as remarks cards) immediately preceding the first line of output associated with the first PMR request executed. If no PMR request is executed they are recorded on the on-line printer.

Error Detection

If an error occurs in a request card the post-mortem program produces a remark describing the nature of the error. In certain cases this causes the program to stop reading request cards and to execute only those requests already translated.

THE IBM-704 INSTRUCTIONS

Notation:

AC = Accumulator — A = Initial contents of AC; A' = final contents — $0 \leq \alpha \leq 2^{15}-1 = 32767$
MQ = Multiplier-Quotient Register — M = Initial contents of MQ; M' = final contents — $0 \leq \beta \leq 2^{3}-1 = 7$
ILC = Instruction Location Counter — L = Initial contents of ILC; L' = final contents — $0 \leq \gamma \leq 2^{15}-1 = 32767$
IRβ = Index Register β — I = Initial contents of IRβ; I' = final contents
R = Register α-I or Register α — ω = Initial contents of R; ω' = final contents
P = P-bit of AC = A_o — Y_i = ith bit of Word Y, $0 \leq i \leq 35$; for $Y \neq A$, Y_o = sign bit; $A_o = P$, $A_{-1} = Q$
Q = Q-bit of AC = A^{o}_{-1} — Y_{i-j} = Bits i through j inclusive of Word Y — $\overline{Y}_i = 0$ if $Y_i = 1$; $\overline{Y}_i = 1$ if $Y_i = 0$ (Complement)

For Floating Point Numbers: $Y_c = Y_{1-8}$ = Characteristic $Y_F = Y_{9-35}$ = Fraction Y_s = sign

For Fixed Point Numbers: $Y_m = Y_{1-35}$ = Magnitude Y_s = Sign

For Instructions: $Y_p = Y_{0-2}$ = Prefix $Y_D = Y_{3-17}$ = Decrement $Y_T = Y_{18-20}$ = Tag $Y_A = Y_{21-35}$ = Address

In General: $L' = \alpha + 1$ R = Register α-I Execution time is 2 cycles = 24 μsec. If otherwise, such will be stated.

If β=0, I=0. If β=1, 2 or 4 a single Index Register is selected. If β= 3, 5, 6, or 7 more than one Index Register is selected and I is formed by a Boolean "OR" of the contents of these registers. If the operation loads the Index, then the same number is placed in each index selected.

The first 5 instructions have decrements. All other instructions do not.

Instruction	Mnemonic Code	Octal Value	AC	MQ	IRβ	R	Comments
Transfer with Index Incremented	TXIα,β,γ	+1000	A'=A	M'=M	$I'=I+\gamma$	$\omega'=\omega$	$*L' = \alpha$, $I + \gamma$ taken mod 2^{15}
Transfer on Index	TIXα,β,γ	+2000	A'=A	M'=M	$I'=I-\gamma$ $I'=I$	$\omega'=\omega$	$*L' = \alpha$ $I - \gamma > 0$ $L' = L + 1$ $I - \gamma \leq 0$
Transfer on No Index	TNXα,β,γ	-2000	A'=A	M'=M	$I'=I$ $I'=I-\gamma$	$\omega'=\omega$	$*L' = \alpha$ $I - \gamma \leq 0$ $L' = L + 1$ $I - \gamma > 0$
Transfer on Index High	TXHα,β,γ	+3000	A'=A	M'=M	I'=I	$\omega'=\omega$	$*L' = \alpha$ if $I > \gamma$
Transfer on Index Low	TXLα,β,γ	-3000	A'=A	M'=M	I'=I	$\omega'=\omega$	$*L' = \alpha$ if $I \leq \gamma$
Halt and Transfer	HTRα,β,	+0000	A'=A	M'=M	I'=I	$\omega'=\omega$	Computer Stops $*L' = \alpha - I$
Transfer	TRAα,β	+0020	A'=A	M'=M	I'=I	$\omega'=\omega$	$*L' = \alpha - I$
Trap Transfer	TTRα,β	+0021	A'=A	M'=M	I'=I	$\omega'=\omega$	$L' = \alpha - I$
Transfer on Low MQ	TLQα,β	+0040	A'=A	M'=M	I'=I	$\omega'=\omega$	$*L' = \alpha - I$ if $A > M$
Transfer and Set Index	TSXα,β	+0074	A'=A	M'=M	$I'=2^{15}-L$	$\omega'=\omega$	$*L' = \alpha$
Transfer on Zero AC	TZEα,β	+0100	A'=A	M'=M	I'=I	$\omega'=\omega$	$*L' = \alpha - I$ if $A_m = 0$
Transfer on Non-zero AC	TNZα,β	-0100	A'=A	M'=M	I'=I	$\omega'=\omega$	$*L' = \alpha - I$ if $A_m \neq 0$
Transfer on AC Plus	TPLα,β	+0120	A'=A	M'=M	I'=I	$\omega'=\omega$	$*L' = \alpha - I$ if $A_s = 0$ (+)
Transfer on AC Minus	TMIα,β	-0120	A'=A	M'=M	I'=I	$\omega'=\omega$	$*L' = \alpha - I$ if $A_s = 1$ (-)
Transfer on AC Overflow	TOVα,β	+0140	A'=A	M'=M	I'=I	$\omega'=\omega$	$*L' = \alpha - I$ if AC overflow light on **
Transfer on No AC Overflow	TNOα,β	-0140	A'=A	M'=M	I'=I	$\omega'=\omega$	$*L' = \alpha - I$ if AC overflow light off **
Transfer on MQ Overflow	TQOα,β	+0161	A'=A	M'=M	I'=I	$\omega'=\omega$	$*L' = \alpha - I$ if MQ overflow light on **
Transfer on MQ Plus	TQPα,β	+0162	A'=A	M'=M	I'=I	$\omega'=\omega$	$*L' = \alpha - I$ if $M_s = 0$
Compare AC with Storage	CASα,β	+0340	A'=A	M'=M	I'=I	$\omega'=\omega$	$L' = L + 2$ if $A = \omega^{s}$; $L' = L + 3$ if $A < \omega$ 3 cycles

*If in trapping mode L'= 1 and L replaces the address part of register O. **Light is turned off if it was on before test.

THE IBM-704 INSTRUCTIONS (Continued)

Instruction	Mnemonic Code	Octal Value	AC	MQ	IRβ	R	Comments
Clear and Add	CLAα,β	+0500	$A'=\omega, Q'=P'=0$	$M'=M$	$I'=I$	$\omega'=\omega$	$A' = \omega$ means $A'_s = \omega_s$, $A'_{1-35} = \omega_{1-35}$
Clear and Add Logical Word	CALα,β	-0500	$A'_{0-35}=\omega_{0-35}$, $A'_S=Q'=0$	$M'=M$	$I'=I$	$\omega'=\omega$	
Clear and Subtract	CLSα,β	+0502	$A'=-\omega$, $Q'=P'=0$	$M'=M$	$I'=I$	$\omega'=\omega$	$A' = -\omega$ means $A'_{1-35} = \omega_{1-35}$ $A'_s = \overline{\omega}_s$
Load MQ	LDQα,β	+0560	$A'=A$	$M'=\omega$	$I'=I$	$\omega'=\omega$	
Load Index from Address	LXAα,β	+0534	$A'=A$	$M'=M$	$I'=\omega_A$	$\omega'=\omega$	R is Register α
Load Index from Decrement	LXDα,β	-0534	$A'=A$	$M'=M$	$I'=\omega_D$	$\omega'=\omega$	R is Register α
Store MQ	STQα,β	-0600	$A'=A$	$M'=M$	$I'=I$	$\omega'=M$	
Store AC	STOα,β	+0601	$A'=A$	$M'=M$	$I'=I$	$\omega'=A$	$\omega' = A$ means $\omega'_s = A_s$ $\omega'_{1-35} = A_{1-35}$
Store Logical Word	SLWα,β	+0602	$A'=A$	$M'=M$	$I'=I$	$\omega'_{o-35}=A_{o-35}$	
Store Left-Half MQ	SLQα,β	-0620	$A'=A$	$M'=M$	$I'=I$	$\omega'_{o-17}=M_{o-17}$	The rest of ω is $\omega'_{18-35} = \omega_{18-35}$
Store Address	STAα,β	+0621	$A'=A$	$M'=M$	$I'=I$	$\omega'_A=A_A$	unchanged in $\omega'_{0-20} = \omega_{0-20}$
Store Decrement	STDα,β	+0622	$A'=A$	$M'=M$	$I'=I$	$\omega'_D=A_D$	each of these $\omega'_p = \omega_p$ $\omega'_{18-35} = \omega_{18-35}$
Store Prefix	STPα,β	+0630	$A'=A$	$M'=M$	$I'=I$	$\omega'_P=A_P$	instructions $\omega'_{3-35} = \omega_{3-35}$
Store Index in Decrement	SXDα,β	-0634	$A'=A$	$M'=M$	$I'=I$	$\omega'_D=I$	$\omega'_p = \omega_p$ $\omega'_{18-35} = \omega_{18-35}$ R is register α 3 cycles
Place Address in Index	PAXα,β	+0734	$A'=A$	$M'=M$	$I'=A_A$	$\omega'=\omega$	
Place Decrement in Index	PDXα,β	-0734	$A'=A$	$M'=M$	$I'=A_D$	$\omega'=\omega$	
Place Index in Decrement	PXDα,β	-0754	$A' = 0$ but $A_D = I$	$M'=M$	$I'=I$	$\omega'=\omega$	PXD with $\beta = 0$ clears AC
Halt and Proceed	HPRα,β	+0420	$A'=A$	$M'=M$	$I'=I$	$\omega'=\omega$	Computer Stops
OR to AC	ORAα,β	-0501	$A'_i=A_i\,(+)\,\omega_i$	$M'=M$	$I'=I$	$\omega'=\omega$	A_s and Q unchanged $0 = i \leq 35$
OR to Storage	ORSα,β	-0602	$A'=A$	$M'=M$	$I'=I$	$\omega'_i=A_i\,(+)\,\omega_i$	$A'_i\,(+)\,\omega_i = 1$ unless $A_i = \omega_i = 0$ when $A_i\,(+)\,\omega_i = 0$
AND to AC	ANAα,β	-0320	$A'_i=A_i\,(x)\,\omega_i$	$M'=M$	$I'=I$	$\omega = \omega$	$A'_S = Q' = 0$ 3 cycles $0 \leq i \leq 35$ $A_i\,(x)\,\omega_i = 0$
AND to Storage	ANSα,β	+0320	$A'=A$	$M'=M$	$I'=I$	$\omega'_i=A_i\,(x)\,\omega_i$	$A'_S = A_s$ $Q' = Q$ 4 cycles unless $A_i = \omega_i = 1$ when $A_i\,(x)\,\omega_i = 1$
Add	ADDα,β	+0400	$A'=A+\omega$	$M'=M$	$I'=I$	$\omega'=\omega$	Overflow possible if there is a carry into P.
Subtract	SUBα,β	+0402	$A'=A-\omega$	$M'=M$	$I'=I$	$\omega'=\omega$	ω_{1-35} added to A_{1-35}. A zero result
Add Magnitude	ADMα,β	+0401	$A'=A+\lvert\omega\rvert$	$M'=M$	$I'=I$	$\omega'=\omega$	has the sign of A
Subtract Magnitude	SBMα,β	-0400	$A'=A-\lvert\omega\rvert$	$M'=M$	$I'=I$	$\omega'=\omega$	Carry out of Q is lost. Fixed Point operation.
Add and Carry Logical	ACLα,β	+0361	$A'=A+\omega_{\lvert 0-35}$ $Q' = 0$	$M'=M$	$I'=I$	$\omega'=\omega$	No overflow light possible. ω_{0-35} added to A_{0-35}. A carry out of P added to A_{35}. $^{0-35}$Carry from Q lost.
Multiply	MPYα,β	+0200	$A'+2^{-35}\cdot M'=\omega\cdot M$		$I'=I$	$\omega'=\omega$	$A'_S = M'_S$ Fixed point operation
Multiply and Round	MPRα,β	-0200	$A'=A''+M'_1$	$M'=M''$	$I'=I$	$\omega'=\omega$	$A'' + 2^{-35}M'' = \omega\cdot M$ 20 cycles
Divide or Halt	DVHα,β	+0220	$M'\cdot\omega+A'=A+2^{-35}\cdot$	$\lvert M\rvert$	$I'=I$	$\omega'=\omega$	Stops Computer on Divide Check M_S ignored. $A'_S = A_S$ 20 cycles
Divide or Proceed	DVPα,β	+0221	$M'\cdot\omega+A'=A+2^{-35}\cdot$	$\lvert M\rvert$	$I'=I$	$\omega'=\omega$	Does Nothing on Divide Check $A' = 2^{-35}$. A in order of magnitude Divide Check if $A_M \leq \omega_M$, $A' = A$, $M' = M$ on Divide Check
Floating Add	FADα,β	+0300	$A'+M'=A+\omega$		$I'=I$	$\omega'=\omega$	If $A'_F \neq 0$, $A'_C - 27 = M'_C$, $1 > A'_F \geq 1/2$; if $A'_F = 0$,
Floating Subtract	FSBα,β	+0302	$A'+M'=A-\omega$		$I'=I$	$\omega'=\omega$	$A'_C = M'_C = 0$ and $A'_S = A_S$ if $A_C \leq \omega_C$, otherwise $A'_S = \pm\omega_S$; AC overflow light only means overflow, MQ light means underflow 7 cycles min, 35 cycles max
Unnormalized Floating Add	UFAα,β	-0300	$A'+M'=A+\omega$		$I'=I$	$\omega'=\omega$	$A'_C = \text{Max}\,(A_C, \omega_C) + c$ $c = [A_f \pm \omega_f]$
Unnormalized Floating Subtract	UFSα,β	-0302	$A'+M'=A-\omega$		$I'=I$	$\omega'=\omega$	A'_F may be $< 1/2$ and $A'_F = 0$, $A'_C \neq 0$ possible. Overflow indications are the same as FAD and FSB but AC underflow is impossible (MQ and AC lights both on). 7 cycles min 28 cycles max

THE IBM-704 INSTRUCTIONS (Continued)

Instruction	Mnemonic Code	Octal Value	AC	MQ	IRβ	R	Comments
Floating Multiply	FMPα,β	+0260	$A'+M'=M\cdot\omega$		$I'=I$	$\omega'=\omega$	$A'_C - 27 = M'_C$ if $A_F \neq 0$, $A'_F \geq 1/2$, if $M_F \cdot \omega_F \geq 1/4$ if $A'_F = 0$, $A'_C = M'_C = 0$ $A'_C - 27 = M'_C$ $A'_F \geq 1/2$ if $M_F \cdot \omega_F \geq 1/2$ } $A'_s = M'_s$; If AC overflow light only overflow; MQ light only, underflow; both lights on, $Q' = 0$ means overflow, $Q' = 1$ means underflow. 17 cycles
Unnormalized Floating Multiply	UFMα,β	-0260	$A'+M'=M\cdot\omega$		$I'=I$	$\omega'=\omega$	
Floating Divide or Halt	FDHα,β	+0240	$M'\cdot\omega+A'=A$		$I'=I$	$\omega'=\omega$	Computer stops on Divide Check } Divide Check if $A_F \geq 2\omega_F$. Does nothing on Divide Check } $A'_s = A_s$ $M_F \geq 1/2$ if $4A_F > \omega_F$ If $A_F = 0$, $A' = M' = 0$, 3 cycles; $A_F \neq 0$, 18 cycles $A'_C + 26 \leq A_C$ AC overflow light means underflow, MQ light only is overflow if $M'_C \leq 128$, underflow if $M_C > 128$
Floating Divide or Proceed	FDPα,β	+0241	$M'\cdot\omega+A'=A$		$I'=I$	$\omega'=\omega$	
Locate Drum Address	LDAα,β	+0460	$A'=A$	$M'=M$	$I'=I$	$\omega'=\omega$	Set so CPY refers to drum address of ω_{25-36}
Copy or Skip	CPYα,β	+0700	$A'=A$	$M'=?$	$I'=I$	$\omega'=\left\{\begin{matrix}\omega\\ Z\end{matrix}\right.$	Z is Word Read In, otherwise ω is written out Min. 2 cycles MQ may not be used at certain times in the CPY loop If in read mode $L' = L + 2$ if end of file condition, $L' = L + 3$ if end of record condition is met Min. 2 cycles
Copy and Add and Carry Logical or Skip	CPAα,β	-0700	$A'=A+\omega'$	$M'=?$	$I'-I$	$\omega'=\left\{\begin{matrix}\omega\\ Z\end{matrix}\right.$	Same as CPY, except word transferred is added to AC in the manner of ACL Min. 2 cycles

The following instructions do not refer to storage. The effective address α-I taken modulo 256 is in fact part of the operation code, α-I = n (mod 256) $0 \leq n < 256$ Certain of these instructions cannot have an address. Others take a small absolute integer designated by i None affect storage or index registers.

Long Left Shift	LLSα,β	+0763	$A'_j=A_{j+m}$	$M'_j=M_{j+m}$			$*A'_s = M'_s = M_s$ for $j+n > 35$, $M'_j = 0$, $A'_j = M_{j+n-35}$ } AC overflow if a 1
Logical Left Shift	LGLα,β	-0763	$A'_j=A_{j+m}$	$M'_j=M_{j+m}$			$*A'_s = A_s$ $M'_s = M_n$ fcr $j+n > 35$ $M'_j = 0$ $A'_j = M_{j+n-36}$ } is shifted into or through P
Long Right Shift	LRSα,β	+0765	$A'_j=A_{j-m}$	$M'_j=M_{j-m}$			$*A'_s = M'_s = A_s$ $j-n \nless -1$, $A'_j = 0$; $j-n < 1$, $M'_j = A_{j-n+35}$
AC Left Shift	ALSα,β	+0767	$A'_j=A_{j+m}$	$M'=M$			$*A'_s = A_s$ $j+n > 35$, $A'_j = 0$ AC overflow if a 1 shifted into or through P
AC Right Shift	ARSα,β	+0771	$A'_j=A_{j-m}$	$M'=M$			$*A'_s = A_s$ $j-n < -i$, $A'_j = 0$
Rotate MQ Left	RQLα,β	-0773	$A'=A$	$M'_j=M_{j+m}$			$*j+n$ taken modulo 36 $*2 + \frac{n-9}{12}$ cycles min. 2, max. 23
No Operation	NOPα,β	-0761					Does Nothing
Rewind Tape Unit	REW i	+0772.....220+i					1 ≤ i ≤ 10 selects one of { Max 1.2 minutes } May be delayed
Write End of File on Tape	WEF i	+0770.....220+i					ten tape units { 50 msec. on tape } if tape is not
Back Space Tape	BST i	+0764.....220+i					$A' = A, M' = M$ MQ available. { Min 50 msec. } ready
Read Select	RDS α,β	+0762					Separate Instruction for each mode
Read Tape in BCD	RTD i	+0762.....200+i					1 ≤ i ≤ 10 for one of ten tape units may be delayed if tape
Read Tape in Binary	RTB i	+0762.....220+i					$M' = 0$ not ready
Read Drum	RDR i	+0762.....300+i					1 ≤ i ≤ 8 for one of eight logical drums
Read Card	RCD	+0762.....321					
Read Printer	RPR	+0762.....361					Actually writes, but with echo check.
Write Select	WRSα,β	+0766					Separate Instruction for each mode
Write on Cathode Ray Tube	WTV	+0766.....030					
Write Tape in BCD	WTD i	+0766.....200+i					1 ≤ i ≤ 10 may be delayed if tape is not ready.
Write Tape in Binary	WTB i	+0766.....220+i					
Write on Drum	WDR i	+0766.....300+i					1 ≤ i ≤ 8
Write on Tape Simultaneously	WTS i	+0766.....320+i					1 ≤ i ≤ 4 Select one of first four tapes, may be delayed
In-Out Delay	IOD	+0766.....333					Delays Computer until MQ is available after reading tape
Write On Punch	WPU	+0766.....341					
Write on Printer	WPR	+0766.....361					

THE IBM-704 INSTRUCTIONS (Continued)

Instruction	Mnemonic Code	Octal Value	Comments
Plus Sense	PSEα,β	+0760	Separate Instruction for each n
Clear Magnitude	CLM	+0760.....000	$A'_s = A_s \quad A'_j = 0, -1 \leq j \leq 35$
Low Bit Test	LBT	+0760.....001	$L' = L+2$ if $A_{35} = 1$
Change Sign	CHS	+0760.....002	$A' = -A$
Set Sign Plus	SSP	+0760.....003	$A' = \lvert A \rvert$
Complement Magnitude	COM	+0760.....006	$A'_s = A_s \quad A'_j = \overline{A}_j \quad -1 \leq j \leq 35$
Enter Trapping Mode	ETM	+0760.....007	See *Note to transfer instructions
Round	RND	+0760.....010	$A' = A + 2^{-35} M_1$ AC overflow possible
Divide Check Test	DCT	+0760.....012	$L' = L+2$ if No Divide Check; Light is turned off if found on
Change Film Frame	CFF	+0760.....030	Index Camera connected to Cathode Ray tube
Sense Lights Off	SLF	+0760.....140	Turns all sense lights off
Sense Light On	SLN i	+0760.....140+i	$1 \leq i \leq 4$ Turn on one of four lights
Sense Switch Test	SWT i	+0760.....160+i	$L' = L+2$ if i^{th} switch is down (on) $1 \leq i \leq 6$
Sense Punch	SPU i	+0760.....340+i	$1 \leq i \leq 2$ Send impulse to hub i on punch control panel
Sense Printer Test	SPT	+0760.....360	$L' = L+2$ If there is impulse on entry hub of printer panel
Sense Printer	SPR i	+0760.....360+i	$1 \leq i \leq 10$ Send impulse to hub i on printer panel
Minus Sense	MSEα,β	-0760	Separate Instruction for each n
P Bit Test	PBT	-0760.....001	$L' = L+2$ if $P = 1$
Set Sign Minus	SSM	-0760.....003	$A' = - \lvert A \rvert$
Leave Trapping Mode	LTM	-0760.....007	See *Note to transfer instructions
Redundancy Tape Test	RTT	-0760.....012	$L' = L+2$ if Tape Check Light is off; Light turned off if found on.
Sense Light Test	SLT i	-0760.....140+i	$L' = L+2$ if Sense Light i is on, Light turned off if found on $1 \leq i \leq 4$

MIT COMPUTATION CENTER
Massachusetts Institute of Technology
Cambridge 39, Massachusetts

MEMORANDUM

To: Computation Center Staff Date: October 1, 1957

From: F. M. Verzuh

Subject: Revised and Up-Dated Index of Available Share 704 Subroutines

The purpose of this memorandum is to provide a revised and up-dated index of the Share-distributed 704 subroutines which are available in the MIT Computation Center as of October 1, 1957. This memorandum is a revision of the previously-issued memorandum CC-34, which contains a list of punched card routines.

This index presents the available subroutines in the manner defined as the Share catalogue classification which was approved at the 7th Share meeting in December, 1956. Specifically, the attached list consists of five parts:

1. A Share catalogue classification of 704 programs,
2. An index of the 704 library of punched card subroutines,
3. A listing of the subroutines available on the MIT Library Tape,
4. A listing of new subroutines and cards received,
5. A listing of addended and superseded subroutines.

The particular format used in Part 2 contains the following information (reading from left to right):

a. Share catalogue code classification,

b. Two-letter Share membership code -- on a Share membership basis,

c. The identification code number of the subroutine employed by the originating Share member,

d. Name or title of subroutine,

e. The Share distribution number (3-digit number),

f. A write-up is available unless there is an asterisk in place of the distribution number indicating no write-up,

g. A listing is available unless there is an asterisk indicating no listing,

h. The letter S indicates that symbolic cards are available at MIT,

i. The letter B indicates that absolute binary cards are available in the MIT library.

j. The letter R indicates that relocatable binary cards are available in the MIT library, Part 3 contains a list of the subroutines which are available on the Library Tape (logical tape 3). The following subroutines are available on the MIT Library Tape:

NA34.1	LIB	Square root
UAS+C1	LIB	Sine and Cosine
UABDC1	LIB	Generalized Print Program
UASTH1	LIB	BCD Tape Writing Program
UASPH1	LIB	BCD Output Program
LAS820	LIB	Floating Natural Logarithm
LAS816	LIB	Floating Exponential
CLASC1	LIB	Arcsine and Arc Cosine
LAS840	LIB	Arctangent
CLTAN1	LIB	Tangent
WHO3	LIB	Arctangent of A/B
NAO5.1	LIB	Floating to Fixed
NAO6.1	LIB	Fixed to Floating
UAINV1	LIB	Matrix Inversion
GMMEQ1	LIB	Generalized Matrix Equation
UADBC1	LIB	Decimal, Octal, BCD Loader
UATSM2	LIB	Read Tape with Redundancy Checking
UACSH2	LIB	Read BCD Tape or On-Line Card Reader
GELAG	LIB	Lagrangian Interpolation
PKPOWR	LIB	Real Power Evaluator
	END	

NOTE: For WHO3, blanks must be in column 5 and 6
For GELAG, blank must be in column 6
Letter O in PKPOWR

LIB cards must be punched as shown in this listing and may be inserted anywhere in a SAP symbolic deck. (See Share material for specifications.)

When using the MIT system, mount reel number 103 as logical tape 3.

The attached index of distributed programs will be revised periodically to keep the index reasonably up to date.

F. M. Verzuh
Assistant Director

nb

COMPUTATION CENTER

Massachusetts Institute of Technology
Cambridge 39, Massachusetts

REVISED SHARE CATALOGUE CLASSIFICATION

At the 7th Share meeting on December 13-14, 1956, the Share program catalogue classification was extended and modified as follows:

A. Programmed Arithmetic
   1. Real
   2. Complex
   3. Decimal

B. Elementary Functions
   1. Trigonometric
   2. Hyperbolic
   3. Exponential and Logarithmic
   4. Roots and Powers

C. Polynomials and Special Functions
   1. Evaluation of Polynomials
   2. Roots of Polynomials
   3. Evaluation of Special Functions

D. Operations on Functions and Solutions of Differential Equations
   1. Numerical Integration
   2. Numerical Solutions of Ordinary Differential Equations
   3. Numerical Solutions of Partial Differential Equations
   4. Numerical Differentiations

E. Interpolation and Approximations
   1. Table Look-up and Interpolation
   2. Curve Fitting
   3. Smoothing

F. Operations on Matrices, Vectors, and Simultaneous Linear Equations
   1. Matrix Operations
   2. Eigenvalues and Eigenvectors
   3. Determinants
   4. Simultaneous Linear Equations

G. Statistical Analysis and Probability
   1. Data Reduction
   2. Correlation and Regression Analysis
   3. Sequential Analysis
   4. Analysis of Variance
   5. Random Number Generators

H. Operations Research and Linear Programming

I. Input
   1. Binary
   2. Octal
   3. Decimal
   4. BCD
   5. Composite

J. Output
   1. Binary
   2. Octal
   3. Decimal
   4. BCD
   5. Analog
   6. Composite

K. Internal Information Transfer
   1. Read Write Drum
   2. Relocation

L. Executive Routines
   1. Assembly
   2. Compiling

M. Information Processing
   1. Sorting
   2. Conversion
   3. Collating and Merging

N. Debugging Routines
   1. Tracing, Trapping
   2. Dump
   3. Search
   4. Breakpoint Print

O. Simulation Programs
   1. Peripheral Equipment Simulators

P. Diagnostic Programs

Q. Service Programs
   1. Clear, Reset Programs
   2. Check Sum Programs
      Restore, Rewind, Tape Mark, Load Button Programs

Z. All Others

SHARE DISTRIBUTED PROGRAMS

OCTOBER 1, 1957

A. PROGRAMMED ARITHMETIC

A1 REAL

				W	L		
A1	CL	DPA1	DOUBLE PRECISION FLOATING ADD	223		S	
A1	CL	DPD1	DOUBLE PRECISION FLOATING DIVIDE	223		S	
A1	CL	DPM1	DOUBLE PRECISION FLOATING MULTIPLY	223		S	
A1	GL	DPPA	DOUBLE-PLUS PRECISION ARITHMETIC (FLOATING POINT	237		S	
A1	GL	DPA1	DOUBLE PRECISION FLOATING POINT ABSTRACTION	110		S	
A1	MU	DPA2	MURA DOUBLE PRECISION ADDITION (FIXED POINT)	256		S	R
A1	NA	018	DOUBLE PRECISION ARITHMETIC ABSTRACTION	096	*		
A1	NA	88.1	DOUBLE PRECISION CLA, CLS, LDQ	096		S	
A1	NA	89.1	DOUBLE PRECISION STO, STQ	096		S	
A1	NA	90.5	DOUBLE PRECISION FLOATING ADD AND SUBTRACT	169		S	
A1	NA	91.3	DOUBLE PRECISION FLOATING MULTIPLY	169		S	
A1	NA	92.1	DOUBLE PRECISION FLOATING DIVIDE	096		S	
A1	NA	92.3	DOUBLE PRECISION FLOATING DIVIDE	149		S	
A1	NA	93.1	DOUBLE PRECISION CHS	096		S	
A1	NA	94.0	DOUBLE PRECISION ABSTRCTN FOR INTERPRETIVE ROUTINE	096		S	
A1	NA	95.1	DOUBLE PRECISION TRA	096		S	
A1	NA	96.1	DOUBLE PRECISION TPL	096		S	
A1	NA	97.1	DOUBLE PRECISION TZE	096		S	
A1	NA	98.1	DOUBLE PRECISION SQUARE ROOT	096		S	
A1	NA	99.0	DOUBLE PRECISION ABSTRACTION EXIT	096		S	
A1	NA	107.	DOUBLE PRECISION TMN	096		S	
A1	NA	108.	DOUBLE PRECISION TRANSFER ON NO ZERO	096		S	
A1	NA	122.	ERROR DETECTION ROUTINE	096		S	
A1	RS	0005	DOUBLE PRECISION FLOATING BINARY ARITHMETIC	047	061	S	
A1	WB	DPA1	DOUBLE PRECISION FAD AND FSB	198			
A1	WB	DPD1	DOUBLE PRECISION FLOATING DIVIDE	198			
A1	WB	DPM1	DOUBLE PRECISION FMP	198			

A2 COMPLEX

A2	CL	DPC1	DOUBLE PRECISION COMPLEX FAD AND FMP	223		S
A2	CL	DPC2	DOUBLE PRECISION COMPLEX FAD, FMP, AND FDP	223		S
A2	GE	CPX	COMPLEX ARITHMETIC INTERPRETIVE SYSTEM	111		S
A2	NA	019	COMPLEX ARITHMETIC ABSTRACTION FLO	087	*	
A2	NA	62.1	COMPLEX SIN AND COS RADIANS FLO	087		S
A2	NA	63.1	COMPLEX N TH ROOT FLO	087		S
A2	NA	64.1	COMPLEX EXP, FLO	087		S
A2	NA	65.1	COMPLEX SINH AND COSH RADIANS, FLO	087		S
A2	NA	66.1	COMPLEX LN, FLO	087		S
A2	NA	67.0	COMPLEX ABSTRACTION FOR INTERPRETIVE ROUTINE	087		S
A2	NA	68.1	COMPLEX CLA, CLS, LDQ, FLO	087		S
A2	NA	69.1	COMPLEX STO AND STQ, FLO	087		S

A2	NA	70.1	COMPLEX ADD AND SUB, FLO	087		S
A2	NA	70.3	COMPLEX ADD AND COMPLEX SUBTRACT	190	087	
A2	NA	71.1	COMPLEX MPY, FLO	087		S
A2	NA	72.1	COMPLEX DIV, FLO	087		S
A2	NA	73.1	COMPLEX RECIPROCAL, FLO	087		S
A2	NA	74.1	COMPLEX CHS	087		S
A2	NA	75.1	COMPLEX TRANSFER ON IMAGINARY PLUS	087		S
A2	NA	76.1	COMPLEX TRANSFER ON REAL PLUS	087		S
A2	NA	77.1	COMPLEX TRANSFER ON REAL ZERO	087		S
A2	NA	78.1	COMPLEX TNZ	087		S
A2	NA	79.1	COMPLEX TRANSFER ON IMAGINARY ZERO	087		S
A2	NA	80.1	COMPLEX TRA	087		S
A2	NA	81.1	COMPLEX CONJUGATE	087		S
A2	NA	82.1	COMPLEX EXIT ABSTRACTION	087		
A2	NA	83.0	COMPLEX DEBUG (ERROR DETECTION ROUTINE)	087		S
A2	NA	84.1	COMPLEX NOP	087		S
A2	NA	85.1	COMPLEX SCALAR MPY, FLO	087		S
A2	NA	86.1	COMPLEX POLAR TO RECTANGULAR CONVERSION, RADIANS, FLO	087		S
A2	NA	87.1	COMPLEX RECTANGULAR TO POLAR CONVERSION, RADIANS, FLO	087		S
A2	NA	1930	SQUARE ROOT OF A COMPLEX NUMBER	211		S

A3 DECIMAL

B. ELEMENTARY FUNCTIONS

B1 TRIGONOMETRIC

B1	AS	AS09	SINE-COSINE,FLOATING	224	S
B1	CL	ASC1	ARC SINE AND ARC COSINE, FLO	116	S
B1	CL	TAN1	TANGENT, RADIANS, FLO	116	S
B1	CS	ART2	ARCTANGENT	092	S
B1	GE	ARCT	ARCTANGENT SUBROUTINE (FLO, RADIANS)	055	
B1	GE	SIN2	SINE-COSINE (RADIANS, FLO)	033	
B1	LA	S840	ARCTAN ROUTINE FLOATING POINT	069	SB
B1	NA	0196	FIXED POINT ARCTANGENT SUBROUTINE	194	S
B1	NA	0198	SIN COS SUBROUTINE FIXED POINT	194	S
B1	NA	30.3	SINE-COSINE SUBROUTINE RADIANS FLO	104	S
B1	NA	33.1	ARCTANGENT RADIANS FLO	051	S
B1	NA	1353	ARC SINE - ARC COSINE SUBROUTINE	246	S
B1	RL	0021	SINE COSINE RADIANS DEGREES CIRCLES FIX	046	S
B1	RL	0029	TANGENT COTANGENT RADIAN FIX	046	S
B1	RL	0041	ARCTANGENT FIXED POINT RADIANS DEGREES CIRCLE	046	S
B1	RL	0052	ARC SINE ARC COSINE FIXED POINT	046	S
B1	RL	0115	TANGENT, COTANGENT RADIANS FIX	125	S
B1	RS	0083	ARCSINE ARC-COSINE FLOATING POINT RADIANS	*	S
B1	UA	ATN1	ARC TANGENT SUBROUTINE RADIAN FLO	004	S
B1	UA	S+C1	SINE AND COSINE SUBROUTINE RADIANS FLO	013	S
B1	WH	03	ARCTAN A/B FLO	049	S

B2 HYPERBOLIC

B2	AS	AS33	HYPERBOLIC SINE-COSINE, FLOATING	224		S	
B2	BA	F113	TANH X FLO	016		S	

B3 EXPONENTIAL AND LOGARITHMIC

B3	AS	AS03	EXPONENTIAL,FLOATING	224		S	
B3	BA	F112	EXPONENTIAL FLO	012		S	
B3	BA	F114	LN X FLO	027		S	
B3	GE	EXP	EXPONENTIAL SUBROUTINE FLO	003		S	
B3	GE	EXP2	EXPONENTIAL SUBROUTINE FLO	020		S	
B3	GE	LN	NATURAL LOGARITHM FLO	003		S	
B3	LA	S816	FLOATING EXPONENTIAL FLO	069		S	
B3	LA	S820	NATURAL LOGARITHM FLOATING	069		S	
B3	NA	31.3	NATURAL LOGARITHM FLO	104		S	
B3	NA	31.5	NATURAL LOGARITHM	189		S	
B3	NA	32.3	EXPONENTIAL SUBROUTINES	104		S	
B3	RL	0037	E TO - X FIXED POINT	021		S	
B3	RL	0038	LOGARITHM FIXED POINT	106		S	
B3	UA	EXP1	EXPONENTIAL SUBROUTINE FLO	010		S	
B3	UA	LN 1	NATURAL LOGARITHM SUBROUTINE FLO	010		S	
B3	UA	LN 2	NATURAL LOGARITHM SUBROUTINE FLO	010		S	
B3	WB	EXP1	DOUBLE PRECISION FLOATING POINT EXPONENTIAL SUBROUTINE	205		S	

B4 ROOTS AND POWERS

B4	CL	SQR2	SQUARE ROOT FLO	8		S	
B4	CL	SQR3	SQUARE ROOT FLO	8		S	
B4	GE	SQR	SQUARE ROOT FLO	* 3		S	
B4	MU	EXP1	MURA EXPONENTIAL, BASE E	256		S	R
B4	MU	EXP2	MURA EXPONENTIAL, BASE 2	256		S	R
B4	NA	34.1	SQUARE ROOT FLO	051		S	
B4	PK	POWR	REAL POWER EVALUATOR	164	203	S	R
B4	RL	0022	SQUARE ROOT FIXED POINT	018		S	
B4	RL	0056	FIXED POINT SQUARE ROOT	046		S	
B4	UA	SQR1	SQUARE ROOT SUBROUTINE FLO	002		S	
B4	UA	SQR2	SQUARE ROOT SUBROUTINE FLO	002		S	
B4	UA	SQR3	SQUARE ROOT SUBROUTINE FLO	004		S	
B4	UA	SQR4	SQUARE ROOT SUBROUTINE FLO	004		S	

C. POLYNOMIALS AND SPECIAL FUNCTIONS

C	CS	BSLS	BESSEL FUNCTIONS, ALL ORDERS FOR ONE ARGUMENT	177		S	
C	GE	BPY	BIVARIATE POLYNOMINAL FLO	003		S	
C	GE	BSL	BESSEL FUNCTIONS	111		S	
C	GL	ROP1	ROOTS OF POLYNOMIALS NEWTONS METHOD	110		S	
C	GM	POL1	POLYNOMIAL EVALUATION FLO	043		SB	
C	NA	152	GSMMS FUNCTION	155		S	
CO	CL	AEQ1	ROOTS OF A POLYNOMIAL FLO	116		S	

C0	NA	152	GSMMS FUNCTION	155		S
C0	CL	RAN1	RANDOM NUMBER GENERATOR	139		S

C1 EVALUATION OF POLYNOMIALS

C1	AS	AS14	POLYNOMIAL COEFFICIENT REDUCTION	224		SB
C1	GE	BPY	BIVARIATE POLYNOMINAL FLO	003		S
C1	GM	POL1	POLYNOMIAL EVALUATION FLO	043		SB

C2 ROOTS OF POLYNOMIALS

C2	CL	AEQ1	ROOTS OF A POLYNOMIAL FLO	116		S
C2	CL	AEQ2	POLYNOMIAL WHERE THE COEFFICIENTS ARE EITHER REAL	223		S
C2	GL	ROP1	ROOTS OF POLYNOMIALS NEWTONS METHOD	110		S
C2	GM	ZER1	ZEROS OF A COMPLEX POLYNOMIAL	225		S

C3 EVALUATION OF SPECIAL FUNCTIONS

C3	GE	BSL	BESSEL FUNCTIONS	111		S
C3	GM	CFR1	CONTINUED FRACTION SUBROUTINE	225		S
C3	GM	IEF1	INCOMPLETE ELLIPTIC INTEGRALS	225		S

D. OPERATIONS ON FUNCTIONS AND SOLUTIONS OF DIFFERENTIAL EQUATIONS

D0	CL	SMD1	SMOOTH AND DIFFERENTIATE DATA POINTS	139		S

D1 NUMERICAL INTEGRATION

D1	CL	INT1	INTEGRAL EVAL., TRAPEZ. RULE (EQU. INTERVALS)	116		S
D1	CL	INT2	INTEGRAL EVAL., TRAPEZ. RULE (UNEQU. INTERV.)	116		S
D1	CL	INT3	INTEGRAL EVAL., SIMPSONS RULE (EQU. INTERV.)	116		S
D1	CL	INT4	INTEGRAL EVAL., SIMPSONS RULE (UNEQU. INTERV.)	116		S
D1	CL	INT4	INTEGRAL EVAL., SIMPSONS RULE (UNEQU. INTERV.)	222		S
D1	GL	GAUS	INTEGRATION SUBROUTINE, GAUSS QUADRATURE METHOD	237		S
D1	LA	S888	GAUSS INTEGRATION OF MULTIPLE INTEGRALS.	141		S
D1	NO	SIG	SIMULTANEOUS MULTIPLE INTEGRATION, FLOATING PT	240		S
D1	PK	HEQ1	POINT HERMITE-GAUSS QUADRATURE INTEGRATION	175		S

D2 NUMERICAL SOLUTIONS OF ORDINARY DIFFERENTIAL EQUATIONS

D2	AT	TPI	TWO POINT BOUNDRY CONDITION DIFFERENTIAL EQUATIR	238		S
D2	CL	DEQ	DIFFERENTIAL EQUATIONS ROUTINE	248		S
D2	GM	DEQ1	DIFFERENTIAL EQUATION FLO	063		S
D2	LA	S887	INTEGRATION OF SPECIAL FORM OF 2ND ORDER EQU.	141		S
D2	PK	NIDA	DIFFERENTIAL EQUATION SOLVING SYSTEM	144	203	SBR

D3 NUMERICAL SOLUTIONS OF PARTIAL DIFFERENTIAL EQUATIONS

D3	AT	MG1	MESH GENERATOR	233		S	

D4 NUMERICAL DIFFERENTIATIONS

D4	CL	SMD1	SMOOTH AND DIFFERENTIATE DATA POINTS	139		S	

E• INTERPOLATION AND APPROXIMATIONS

E1 TABLE LOOK-UP AND INTERPOLATION

E1	BA	F016	TABLE LOOK UP AND FAMILY INTERPOLATION FIX	007	039	S	
E1	CL	DD12	TABLE LOOK UP DIVIDED DIFFERENCE INTERPOLATION	116		S	
E1	CL	DDT1	DIVIDED DIFFERENCE TABLE FORMATION	116		S	
E1	GE	LAG	LAGRANGIAN INTERPOLATION SUBROUTINE FLO	003		S	
E1	GM	DIN1	DOUBLE INTERPOLATION FLO	043		SB	
E1	GM	HVP1	AIR TABLE SUBPROGRAM	121		S	
E1	GM	HVT1	AIR TABLE SUBPROGRAM	121		S	
E1	GM	PVH1	AIR TABLE SUBPROGRAM	121		S	
E1	GM	PVT1	AIR TABLE SUBPROGRAM	121		S	
E1	GM	TIN1	TABLE INTERPOLATION FLO	043		SB	
E1	GM	TIN2	INTERPOLATION SUBROUTINE	259		S	
E1	GM	TVH1	AIR TABLE SUBPROGRAM	121		S	
E1	GM	TVP1	AIR TABLE SUBPROGRAM	121		S	
E1	LA	S880	TABLE LOOK UP BINARY LINEAR INTERPOLATION FLO	069		S	
E1	NA	04•1	BINARY SEARCH ROUTINE	051		S	
E1	PK	ANIP	AITKENS INTERPOLATION FOR N EQUAL INTERVALS FLO	122		S	R
E1	UA	BPE1	BIVARIATE POLYNOMIAL EVALUATION SUBROUTINE	010		S	
E1	UA	BPE3	BIVARIATE TABLE INTERPOLATION	081		S	
E1	WK	LIN1	LAGRANGIAN INTERPOLATION	197		S	

E2 CURVE FITTING

E2	CL	LSQ1	LEAST SQUARES POLYNOMIAL FIT	116		S	
E2	CL	PIN1	PARABOLIC INTERPOLATION	248		S	
E2	CL	PIN2	BIVARIATE PARABOLIC INTERPOLATION	248		S	
E2	GM	HAS1	HARMONIC ANALYSIS SUBROUTINE	121		S	
E2	GM	ITR1	ITERATION SUBROUTINE	121		S	
E2	GM	ITR1	ITERATION SUBROUTINE	225		S	
E2	GM	LSQ1	GENERAL LEAST SQUARES CURVE FITTING ROUTINE	121		S	
E2	GM	SL30	SMOOTHING WEIGHT TABLE SL30	156	121	S	
E2	GM	SL31	SMOOTHING WEIGHT TABLE SL31	156	121	S	
E2	GM	SL32	SMOOTHING WEIGHT TABLE SL32	156	121	S	
E2	GM	SMT1	SMOOTHING SUBROUTINE FLO	121		S	
E2	LA	S872	GENERAL LEAST SQUARES	141		S	
E2	NA	0197	FIXED POINT FOURIER ANALYSIS	194		S	
E2	NY	FSC1	FIXED POINT FOURIER COEFFICIENTS	250		S	R
E2	PK	4EA1	FOURIER SERIES COEFFICIENT SUBROUTINE	175		S	R
E2	SC	CUF1	CURVE FITTING PROGRAM	201		SB	

E3 SMOOTHING

F. OPERATIONS ON MATRICES,
VECTORS AND SIMULTANEOUS LINEAR EQUATIONS

F	GE	SMQ	SIMULTANEOUS EQUATION SOLUTION	003	S
F	GL	DEV1	DETERMINANT EVALUATION	110	S
F	GM	MEQ1	GENERALIZED MATRIX EQUATION	043	S R
F	NY	CRV1	CHARACTERISTIC ROOTS AND VECTORS	148	B
F	NY	CSM2	CHANGE SIGNS OF MATRIX ELEMENT	178	S R
F	NY	FMA1	FLOATING POINT, SINGLE PRECISION MATRIX ADDITION	170	S R
F	NY	MXL1	MATRIX CONVERSION FIT	173	S R
F	PK	CIMX	COMPLEX ELEMENT MATRIX INVERSION	122	S R
F	PK	FLIP	COMPLEX ELEMENT MATRIX INVERSION	122	SB
F0	CL	DET1	DETERMINANT AND EIGENVECTOR FOR REAL MATRIX	116	S
F0	CL	DET2	DETERMINANT AND EIGENVECTOR FOR COMPLEX MATRIX.	116	S
F0	CL	FSC1	FRACTION SERIES SOLUTION COMPLEX	139	S
F0	CL	FSR1	FRACTION SERIES SOLUTION, REAL	139	S
F0	CL	LSQ2	LEAST SQUARES SOL. OF SIMULTANEOUS EQUATIONS	116	S
F0	CL	LSQ3	LEAST SQUARES SOL. OF SIMULTANEOUS EQUATIONS	116	S
F0	CL	MAD1	MATRIX ADDITION	085	S
F0	CL	MBH1	MATRIX HEADING REMOVAL	085	S
F0	CL	MCP1	MATRIX PUNCH	085	S
F0	CL	MCR1	MATRIX CARD READ	085	S
F0	CL	MEX1	MATRIX EXPAND	085	S
F0	CL	MIN1	MATRIX INTERCHANGE OF ROWS AND COLUMNS	085	S
F0	CL	MIV1	MATRIX INVERSE	085	S
F0	CL	MK01	TIMES UNIT MATRIX	085	S
F0	CL	MLD1	LOAD MATRIX TO C.S. FROM DRUM OR TAPE	085	S
F0	CL	MLP1	MATRIX LOOP TEST	085	S
F0	CL	MMP1	MATRIX MULTIPLICATION	085	S
F0	CL	MPR1	MATRIX PRINT	085	S
F0	CL	MSB1	MATRIX SUBTRACTION	085	S
F0	CL	MSM1	SCALAR MATRIX MULTIPLICATION	085	S
F0	CL	MST1	STORE MATRIX FROM C.S. TO C.S., DRUM, OR TAPE	085	S
F0	CL	MTR1	MATRIX TRANSPOSE	085	S
F0	CL	MTX1	INTERPRETATION MATRIX ABSTRACTION	085	S
F0	CL	SME1	SIMULTANEOUS REAL EQUATIONS, DETERMINANT	116	S
F0	CL	SME1	SIMULTANEOUS REAL EQUATIONS, DETERMINANT	222	S
F0	CL	SME2	SIMULTANEOUS EQUATIONS COMPLEX	116	S
F0	CL	SME3	SIMULTANEOUS REAL EQUATIONS	116	S
F0	LA	S885	SOLUTION OF GENERAL MATRIX EQUATION AX = B.	141	S
F0	MB	MTX1	GENERALIZED MATRIX ABSTRACTION, REAL COMPLEX	138	S
F0	PK	CIMX	COMPLEX ELEMENT MATRIX INVERSION	122	S
F0	PK	FLIP	COMPLEX ELEMENT MATRIX INVERSION	122	SB
F0	UA	INV1	MATRIX INVERSION	058	S

F1 MATRIX OPERATIONS

F1	CL	MAD1	MATRIX ADDITION	085	S

F1	CL	MBH1	MATRIX HEADING REMOVAL	085	S
F1	CL	MCP1	MATRIX PUNCH	085	S
F1	CL	MCR1	MATRIX CARD READ	085	S
F1	CL	MEX1	MATRIX EXPAND	085	S
F1	CL	MIN1	MATRIX INTERCHANGE OF ROWS AND COLUMNS	085	S
F1	CL	MIV1	MATRIX INVERSE	085	S
F1	CL	MIV2	INVERSE, REAL	223	S
F1	CL	MIV3	INVERSE, REAL OR COMPLEX.	223	S
F1	CL	MK01	TIMES UNIT MATRIX	085	S
F1	CL	MLD1	LOAD MATRIX TO C.S. FROM DRUM OR TAPE	085	S
F1	CL	MLP1	MATRIX LOOP TEST	085	S
F1	CL	MMP1	MATRIX MULTIPLICATION	085	S
F1	CL	MNR1	NORMALIZE MATRIX	223	S
F1	CL	MNR3	NORMALIZE MATRIX BY COLUMNS.	236	S
F1	CL	MPR1	MATRIX PRINT	085	S
F1	CL	MSB1	MATRIX SUBTRACTION	085	S
F1	CL	MSM1	SCALAR MATRIX MULTIPLICATION	085	S
F1	CL	MST1	STORE MATRIX FROM C.S. TO C.S., DRUM, OR TAPE	085	S
F1	CL	MST2	STORE SUBMATRICES IN A LARGE MATRIX	223	S
F1	CL	MST3	STORE ROW MATRICES INTO A LARGE MATRIX	223	S
F1	CL	MTR1	MATRIX TRANSPOSE	085	S
F1	CL	MTR1	MATRIX TRANSFER	223	S
F1	CL	MTX1	INTERPRETATION MATRIX ABSTRACTION	085	S
F1	CL	MVP1	VECTOR DOT PRODUCT	223	S
F1	CL	PMC1	EIGENVALUE SOLUTION, COMPLEX	248	S
F1	LA	S885	SOLUTION OF GENERAL MATRIX EQUAT+ON AX = B.	141	S
F1	MB	MTX1	GENERALIZED MATRIX ABSTRACTION, REAL COMPLEX	138	S
F1	NY	CMI1	COMPLEX MATRIX INVERSION	185	B
F1	NY	DMI1	MATRIX INVERSION	232	S
F1	UA	INV1	MATRIX INVERSION	058	S

F2 EIGENVALUES AND EIGENVECTORS

F2	CL	DET1	DETERMINANT AND EIGENVECTOR FOR REAL MATRIX	116	S
F2	CL	FSC1	FRACTION SERIES SOLUTION COMPLEX	139	S
F2	NY	CRV1	CHARACTERISTIC ROOTS AND VECTORS	148	B
F2	NY	CRV3	CHARACTERISTIC ROOTS AND VECTORS	218	B
F2	GM	EIG2	EIGENVALUE SUBROUTINE	225	S

F3 DETERMINANTS

F3	CL	DET2	DETERMINANT AND EIGENVECTOR FOR COMPLEX MATRIX.	116	S
F3	CL	DET3	DETERMINANT AND EIGENVECTOR, REAL	223	S
F3	CL	SMD2	SMOOTH AND DIFFERENTIATE DATA POINTS	223	S
F3	CL	MDT1	DETERMINANT AND EIGENVECTOR EVALUTION	223	S
F3	GL	DEV1	DETERMINANT EVALUATION	110	S

F4 SIMULTANEOUS LINEAR EQUATIONS

F4	CL	SME1	SIMULTANEOUS REAL EQUATIONS, DETERMINANT	116	S

F4	CL	SME2	SIMULTANEOUS EQUATIONS COMPLEX	116		S
F4	CL	SME3	SIMULTANEOUS REAL EQUATIONS	116		S
F4	CL	SME4	SIMULTANEOUS EQUATIONS, REAL	223		S
F4	CL	SME5	SIMULTANEOUS EQUATIONS, REAL	223		S
F4	GE	SMQ	SIMULTANEOUS EQUATION SOLUTION	003		S
F4	GM	MEQ1	GENERALIZED MATRIX EQUATION	043		S R

G. STATISTICAL ANALYSIS AND PROBABILITY

G	NY	MR1	MULTIPLE REGRESSION AND CORRELATION ANALYSIS	151		B

G1 DATA REDUCTION

G2 CORRELATION AND REGRESSION ANALYSIS

G2	NY	MR1	MULTIPLE REGRESSION AND CORRELAT+ON ANALYSIS	151		B

G3 SEQUENTIAL ANALYSIS

G4 ANALYSIS OF VARIANCE

G5 RANDOM NUMBER GENERATORS

G5	CL	RAN1	RANDOM NUMBER GENERATOR	139	*	S

H. OPERATIONS RESEARCH AND LINEAR PROGRAMMING

H0	RS	LPS1	LINEAR PROGRAMMING SYSTEM	108		
H	RS	LNP1	LINEAR PROGRAMMING PROGRAM	161	108	
H1	RS	LPS1	LINEAR PROGRAMMING SYSTEM	108		

I. INPUT

I1 BINARY

I1	DM	CSB1	ONE CARD ABSOLUTE BINARY LOADER	137		B
I1	DS	CBL1	CHINESE BINARY ON-LINE LOADER	162		SB
I1	GL	BUL1	ONE CARD ABSOLUTE BINARY UPPER LOADER	028		B
I1	GL	BUL2	ONE CARD ABSOLUTE BINARY UPPER LOADER	044		B
I1	MU	LBL3	MURA LOWER BINARY LOADER (ONE CARD)	251		SB
I1	MU	UBL1	MURA UPPER BINARY LOADER (ONE CARD)	251		SB
I1	NY	BL1		*	*	B
I1	NY	RBL1	RELOCATABLE BINARY LOADER	183		B
I1	PK	CSB1	ABSOLUTE BINARY CARD + TRANSFER CARD LOADER	019		B
I1	PK	CSB2	ABSOLUTE BINARY + CORRECTION CARD LOADER	019		B
I1	PK	CSB3	RELOCATING BINARY LOADER, LOWER	208		SB
I1	PK	CSB4	GENERAL BINARY CARD LOADER	019		B
I1	PK	CSBR	RELOCATING BINARY LOADER, UPPER	208		SB
I1	RA	BCSC	BINARY LOADER AND CHECK SUM CORRECTOR (LBLCSC)	082		B
I1	RL	0058	ABSOLUTE BINARY LOADER	106		B

I1	UA	CSE1	ABSOLUTE BINARY LOADER	066	SB
I1	UA	CSB2	ABSOLUTE BINARY LOADER	066	S
I1	UA	RWT1	BINARY READ-WRITE TAPE PROGRAM	120	S
I1	WK	TL1	AUTOMATIC TAPE LOADER-TO. WRITE A SELF-LOADING R	214	SB
I1	UA	TSB3	LOAD BINARY CARD IMAGES FROM TAPE	119	SB

I2 OCTAL

I2	DM	OCHG	704 ONE-CARD OCTAL LOADER	101	B
I2	RS	112X	AND 112Y ONE CARD OCTAL CORRECTORS	249	B
I2	WH	02	SEQUENTIAL DATA INPUT-VARIABLE FIELD	134	S R

I3 DECIMAL

I3	NO	INP	VARIABLE FIELD DECIMAL INPUT	241	S
I3	MU	RDI1	MURA READ DECIMAL INTEGER ROUTINE	256	S R
I3	NO	VNPT	A VARIABLE FIELD PERIPHERAL INPUT	209	S
I3	NY	BLI1	BASIC LOOP INITIALIZER	145	S R
I3	NY	BLU1	BASIC LOOP UPDATER	145	R
I3	NY	DCR2	DIRECT CARD READER	145	S R
I3	NY	DL1	DECIMAL DATA INPUT PROGRAM	152	SB
I3	NY	INS1	INTEGER TO NUMBER SCALER	145	S R
I3	NY	LWR1	LOCATION TO WORKSPACE RETRIEVER	145	S R
I3	NY	NFS2	NUMBER TO FRACTION SCALER	145	S R
I3	NY	PCR2	PERIPHERAL CARD READER	145	S R
I3	RS	0001	CARD TO QUASI BCD	018	S
I3	RS	0046	FLOATING POINT + FIXED POINT DECIMAL INPUT.	040	S

I4 BCD

I4	GL	IN4	BCD TAPE INPUT PROGRAM	182	S
I4	NA	0180	CARD PROGRAMMED CONVERTER	150	S
I4	NA	1801	CARD PROGRAMMED CONVERTER	245	S
I4	NY	DBD1	HOLLERITH TO BCD CONVERSION	235	S R
I4	NY	ISC1	INPUT SCALER	*	S R
I4	RS	0001	CARD TO QUASI BCD	018	S
I4	UA	DBC1	DECIMAL, OCTAL, BCD LOADER	073	S
I4	WH	001	CARD DATA INPUT-VARIABLE FIELD	057	S R
I4	WH	02	SEQUENTIAL DATA INPUT-VARIABLE FIELD	134	2

I9 COMPOSITE

I9	EL	BOL1	TWO CARD SELF LOADING PROGRAM TO LOAD OBSOLUTE BINARY	182	S
I9	GL	FILE	COMPOSITE INPUT PROGRAM	181	S
I9	GS	IN2	SCHENECTADY DECIMAL INPUT PROGRAM-VARIABLE FORMAT	204	S
I9	NY	BOL1	BINARY OCTAL LOADER	215	B
I9	NY	INP1	INPUT PROGRAM UNDER SENSE SWITCH CONTROL	206	S R
I9	NY	INP2	INPUT PROGRAM UNDER SENSE LIGHT CONTROL	206	S R
I9	UA	CSH2	READ BCD TAPE OR ON-LINE CARD READER	073	S
I9	UA	DBC1	DECIMAL, OCTAL, BCD LOADER	073	S
I9	WH	001	OCTAL DATA INPUT VARIABLE FIELD	057	S R

I9	UA	TSM2	READ TAPE WITH REDUNDANCY CHECKING	073	S

J. OUTPUT

J	GM	CAP1	COMMENT ATTACHED PRINTER	121	S
J0	CL	PLT1	POINT PLOT	131	S

J1 BINARY

J1	LA	A720	REPRODUCE BINARY CARDS WITH CORRECT CHECK SUM.	069	SB
J1	MU	BPU1	MURA BINARY PUNCH ROUTINE	256	S R
J1	MU	BPU2	MURA BINARY PUNCH ROUTINE	256	S R
J1	NA	03.1	ABSOLUTE BINARY CARD PUNCH	051	S
J1	NY	BPU1	BINARY PUNCH	075	S R
J1	NY	BPU3	BINARY PUNCH	075	S R
J1	NY	BPU4	BINARY PUNCH PROGRAM	212	SB
J1	NY	BPU5	BINARY PUNCH PROGRAM	212	SB
J1	NY	BTD1	BINARY TAPE OR DRUM DUMP	075	S R
J1	NY	BTD2	BINARY TAPE OR DRUM DUMP	075	S R

J2 OCTAL

J2	NA	12.1	ABSOLUTE OCTAL CARD PUNCH	051	S
J2	NA	12.2	ABSOLUTE OCTAL CARD PUNCH	150	
J2	RL	0010	PRINT TAPE IN OCTAL	018	S
J2	RL	0065	OCTAL TAPE PRINT	106	B

J3 DECIMAL

J3	GM	CAP1	COMMENT ATTACHED PRINTER	121	S
J3	LA	S110	PRINT FLOATING DECIMAL DATA	069	S
J3	LA	S111	PRINT FLOATING DECIMAL DATA	069	S
J3	MU	PIF1	MURA VARIABLE COLUMN INTEGER-FRACTION PRINT	258	S R
J3	MU	PRF1	MURA VARIABLE COLUMN FRACTION PRINT	258	S R
J3	MU	PRF2	MURA SIX COLUMN FRACTION PRINT	258	S R
J3	MU	PRF3	MURA VARIABLE COL. AND DIGIT ROUNDED FRACTION P	258	S R
J3	MU	PRI1	MURA VARIABLE COLUMN INTEGER PRINT	258	S R
J3	NA	0117	WRITE 6-DIGIT DECIMAL INTEGER ON CRT	150	S
J3	NY	BLI1	BASIC LOOP INITIALIZER	145	S R
J3	NY	BLU1	BASIC LOOP UPDATER	145	R
J3	NY	DB01	FIXED POINT OUTPUT FOR ATTACHED PRINTER	152	S R
J3	NY	DCP2	DIRECT CARD PUNCHER	145	S R
J3	NY	DLP2	DIRECT LINE PRINTER	145	S R
J3	NY	FP01	FLOATING POINT OUTPUT - ATTACHED PRINTER	075	S R
J3	NY	FP02	FLOATING POINT OUTPUT - ATTACHED PUNCH	075	S R
J3	NY	FP03	FLOATING POINT OUTPUT - PERIPHERAL PRINTER	075	S R
J3	NY	FP04	FLOATING POINT OUTPUT - PERIPHERAL PUNCH	075	S R
J3	NY	NFS2	NUMBER TO FRACTION SCALER	145	S R
J3	NY	PCP2	PERIPHERAL CARD PUNCHER	145	S R
J3	NY	PLP2	PERIPHERAL LINE PRINTER	145	S R

J3	NY	WLD1	WORKSPACE TO LOCATION DISPERSER	145	S R
J3	RL	0007	NORMALIZED FLOATING POINT PRINT	018	S
J3	RL	0020	FRACTIONAL FIXED POINT PRINT	018	S
J3	RL	0023	NORMALIZED FLOATING POINT PRINT	021	S
J3	RL	0063	FLOATING POINT TAPE PRINT	106	B
J3	RS	0006	BCD TO PRINTER OR PUNCH	018	S
J3	RS	0129	FLOATING POINT DECIMAL PUNCH	257	S
J3	WB	SPF1	DOUBLE PRECISION FLOATING POINT PRINT	205	S

J4 BCD

J4	GL	OUT1	GENERAL PURPOSE OUTPUT PROGRAM	084	S
J4	GL	OUT2	GENERAL PURPOSE OUTPUT PROGRAM	084	S
J4	GM	GPR1	GENERAL PRINT PROGRAM	070	B
J4	GM	GPR2	GENERAL PRINT PROGRAM	070	B
J4	GM	GPR3	GENERAL PRINT PROGRAM	070	B
J4	GM	GPR4	GENERAL PRINT PROGRAM	070	B
J4	GM	GPR5	GENERAL PRINT PROGRAM	070	B
J4	NA	0109	WRITE A SINGLE BCD CHARACTER ON CRT	150	S
J4	NA	0110	WRITE BCD CHARACTES STORED IN N-704 WORDS ON CRT	150	S
J4	NA	0111	PLOT PT. GIVEN BY SET OF COORDINATES IN FL. PT.	150	S
J4	NA	0112	GENERATE GRID ON CRT	150	S
J4	NA	1391	CONVERT BINARY CARDS TO OCTAL CARDS	150	S
J4	NY	CIG1	CARD IMAGE GENERATOR	*	S R
J4	NY	DHL1	BCD TO HOLLERITH	235	S R
J4	NY	OSC1	OUTPUT SCALER	174	S R
J4	NY	PCP2	PERIPHERAL CARD PUNCHER	145	S R
J4	NY	PLP2	PERIPHERAL LINE PRINTER	145	S
J4	NY	TRC1	BCD RECORD TO CARD IMAGE	*	S R
J4	NY	TRG1	BCD TAPE RECORD GENERATOR	*	S R
J4	RS	0006	BCD TO PRINTER OR PUNCH	018	S
J4	UA	BDC1	GENERALIZED PRINT PROGRAM	072	S R
J4	UA	STH1	BCD TAPE WRITING PROGRAM	072	S

J5 ANALOG

J5	CL	PLT2	POLAR PLOT	236	S

J9 COMPOSITE

J9	GL	OUT1	GENERAL PURPOSE OUTPUT PROGRAM	084	S
J9	GL	OUT2	GENERAL PURPOSE OUTPUT PROGRAM	084	S
J9	GM	GPR1	GENERAL PRINT PROGRAM	070	B
J9	GM	GPR2	GENERAL PRINT PROGRAM	070	B
J9	GM	GPR3	GENERAL PRINT PROGRAM	070	B
J9	GM	GPR4	GENERAL PRINT PROGRAM	070	B
J9	GM	GPR5	GENERAL PRINT PROGRAM	070	B
J9	GS	OUTR	GENERAL PURPOSE OUTPUT PROGRAM	204	S
J9	NS	006	BINARY PROGRAM LISTER	102	SB
J9	NY	OUT1	DECIMAL OUTPUT PROGRAM UNDER SENSE SWITCH CONTROL	206	S R

J9	NY	OUT2	DECIMAL OUTPUT PROGRAM UNDER SENSE LIGHT CONTROL	206		S R
J9	UA	BDC1	GENERALIZED PRINT PROGRAM	072		S
J9	UA	SPH1	BCD OUTPUT PROGRAM	072		S

K. INTERNAL INFORMATION TRANSFER

K	RL	0044	TAPE COPY	106		B
K	RL	0059	BINARY CHECK SUM CORRECTOR	106		B
K	RL	0080	WORD INSERTION	106		S
K	RL	0081	ECHO CHECK PRINTER COPY LOOP	106		S
K	RL	0116	CHECK SUM TAPE COPY	133		R
K	RS	0075	ADJUST TAPE	091		SB
K	RS	0077	REVERSE TAPE	091		SB
K0	NO	RWT	READ WRITE TAPE SUBROUTINE.	209		S
K0	NY	TFD1	TAPE FILE DUPLICATOR	255		SB
K0	NY	TFD2	TAPE FILE DUPLICATOR	255		S R
K0	UA	CCB1	BINARY CHECK SUM CORRECTOR	010		B
K0	UA	CSH2	READ BCD TAPE OR ON-LINE CARD READER	073		S
K0	UA	CTH1	OFF-LINE CARD READER SIMULATOR	024		SB
K0	UA	RWD2	READ-WRITE DRUM PROGRAM	080		S
K0	UA	RWT1	BINARY READ-WRITE TAPE PROGRAM	120		S
K0	UA	SPH1	BCD OUTPUT PROGRAM	072		S
K0	UA	STH1	BCD TAPE WRITING PROGRAM	072		S
K0	UA	TCH1	OFF-LINE PUNCH SIMULATOR	071		SB
K0	UA	TPH1	OFF-LINE PRINTER SIMULATOR	071		SB
K0	UA	TSB3	LOAD BINARY CARD IMAGES FROM TAPE	119		SB
K0	UA	TSM2	READ TAPE WITH REDUNDANCY CHECKING	073		S

K1 READ WRITE DRUM

K1	NY	BTD4	BINARY TAPE OR DRUM DUMP	213		S R
K1	UA	RWD1	READ-WRITE DRUM	054		S
K1	UA	RWD2	READ WRITE DRUM PROGRAM	080		S

K2 RELOCATION

L. EXECUTIVE ROUTINES

L0	CW	DIS1	RELOCATABLE TO SYMBOLIC DISASSEMBLER	153		B
L0	PK	DSMB	BINARY CARD DISASSEMBLY PROGRAM	158	*	
L0	RS	0128	DE RELATIVIZE PROGRAM	230		SB

L1 ASSEMBLY

L1	NA	PREA	PRE-ASSEMBLY PROGRAM	176		SB
L1	NA	SAP1	SYMBOLIC ASSEMBLY PROGRAM	176		SB
L1	NA	SAP2	SYMBOLIC ASSEMBLY PROGRAM NAA VERSION	176		SB
L1	NA	1780	WRITE BINARY LIBRARY TAPE	176		SB
L1	RN	019	701-704 SYMBOLIC ASSEMBLY PROGRAM - 1 FRAM	014		
L1	UA	SAP1	SHARE ASSEMBLER	056	036	SB

```
                L2 COMPILING

                M. INFORMATION PROCESSING

                M1 SORTING

M1 NA 20.1 SORTING,MINIMUM SPACE                                 051        S
M1 NO SORT A MEMORY-SORT SUBROUTINE                              209        S
M1 NS MRG1 MERGE PROGRAM                                         129        SB
M1 NS SRT1 SORT PROGRAM                                          129        SB

                M2 CONVERSION

M2 NA 05.1 FLOATING TO FIXED                                     051        S
M2 NA 06.1 FIXED TO FLOATING                                     051        S
M2 NA 07.1 FIXED INPUT OUTPUT SCALING SUBROUTINE                 051        S
M2 NA 15.1 FLOATING INPUT SCALING                                051        S
M2 NA 16.1 FLOATING OUTPUT SCALING SUBROUTINE                    051        S
M2 NO BCDI PACKED BCD TO INTEGER BINARY SUBROUTINE.              209        S
M2 NO FCD  FLOATING NUMBER TO PACKED BCD SUBROUTINE.             209        S
M2 NY CIG1 CARD IMAGE GENERATOR                                   *         S R
M2 NY OSC1 OUTPUT SCALER                                         174        S R
M2 NY TRC1 BCD RECORD TO CARD IMAGE                               *         S R
M2 NY TRG1 BCD TAPE RECORD GENERATOR                              *         S R
M2 RS 0002 PACKED BCD TO INTEGER BINARY                          018        S
M2 RS 0003 POSITIVE BINARY INTEGER TO UNPACKED BCD               018        S
M2 RS 0009 POSITIVE BINARY INTEGER TO UNPACKED BCD               018        S
M2 UA CTQ1 QUADOCTAL TAPE WRITING PROGRAM                        221        SB
M2 UA TSQ1 QUADOCTAL TAPE READING PROGRAM                        221        SB

                M3 COLLATING AND MERGING

                N. DEBUGGING ROUTINES

NO LA D481 TRAP DECIMAL OR OCTAL MEMORY PRINT                    095   *     B
NO NS  006 BINARY PROGRAM LISTER                                 102        SB
NO UA SPM1 TRAP DECIMAL MEMORY PRINT                             113        SB

                N1 TRACING, TRAPPING

N1 GS HEJ  TRAPPING MODE CONTROL SUBROUTINES.                    204        S
N1 LA D080 LOGIC TRACE                                           069        SB
N1 LA D081 LOGIC TRACE WITH PARTIAL PRINT                        069        SB
N1 LA D481 DYNAMIC PRINT MONITOR                                 095   *
N1 MU EAS2 MURA EFFECTIVE ADDRESS SEARCH ROUTINE                 253        SB
N1 MU TTV1 MURA TRANSFER TEST (VISUAL)                           253        SB
N1 NS TRC1 TRACE PROGRAM, SUPERCEDES NS001                       129         B
N1 NY FTR1 HIGH-SPEED FLOW TRACE                                 147        SB
N1 UA SPO2 FLOW TRACE                                            026        SB
```

N1	WK	BP	LEVELS OF BREAKPOINT PRINTING	154		S
			N2 DUMP			
N2	GA	VALT	STATUS STORING ROUTINE	112		S
N2	LA	A420	DUMP MEMORY ON A SELECTED LOGICAL DRUM	069		B
N2	LA	D770	MEMORY PRINT OUT	069		B
N2	MU	FRD1	MURA FRACTION DUMP	253		SB
N2	NS	005	MEMORY VERIFICATION PROGRAM	102		SB
N2	PK	SPOA	PRINT AND RESTORE CONSOLE	208		S R
N2	RS	0071	PUNCH CONSOLE	067		SB
N2	UA	SP01	CONTROL PANEL PRINT + OCTAL MEM. PRINT (SCOOP)	029		SB
			N3 SEARCH			
N3	LA	D620	TRANSFER SEARCH PROGRAM	069		B
N3	LA	D621	SEARCH MEMORY	069		SB
N3	NS	005	MEMORY VERIFICATION PROGRAM	102		SB
			N4 BREAKPOINT PRINT			
N4	UA	SPM1	TRAP DECIMAL MEMORY PRINT	113		SB
			O. SIMULATION PROGRAMS			
O	EL	TEST	36 SIMULATED LOGICAL SWITCHES	220		S
			O1 PERIPHERAL EQUIPMENT SIMULATORS			
O1	NY	PCV1	PERIPHERAL CARD VERIFIER	262		S R
O1	UA	CTH1	OFF-LINE CARD READER SIMULATOR	024		SB
O1	UA	TCH1	OFF-LINE PUNCH SIMULATOR	071		SB
O1	UA	TPH1	OFF-LINE PRINTER SIMULATOR	071		SB
			P. DIAGNOSTIC PROGRAMS			
			Q. SERVICE PROGRAMS			
			Q1 CLEAR, RESET PROGRAMS			
Q1	CL	OUD1	OVERFLOW, UNDERFLOW, AND DIVIDE CHECK TEST	248		S
Q1	UA	ZCS1	CLEAR CORE STORAGE AND MAIN FRAME	048		SB
Q1	UA	ZCS2	SET CORE STORAGE TO ZERO	119		SB
Q1	UA	ZDR1	CLEAR N DRUMS	065		S
			Q2 CHECK SUM PROGRAMS			
Q2	NY	BL2	BINARY LOADER AND ZERO CHECKSUM CORRECTOR	*	*	B
Q2	NY	BL3	BINARY LOADER AND CHECKSUM CORRECTOR	*	*	B
Q2	UA	CCB1	BINARY CHECK SUM CORRECTOR	010		B
Q2	RL	0059	BINARY CHECK SUM CORRECTOR	106		B

```
Q2 RL 0116 CHECK'SUM TAPE COPY                                           133
Q2 UA PCS1 PUNCH DRUM CHECK SUM VERIFIER                                 065       S
Q2 UA VCS1 VERIFY DRUM CHECK SUM                                         065       S

           Q3 RESTORE, REWIND, TAPE MARK, LOAD BUTTON PROGRAMS.

Q3 NY PLB3 NY BOL1 TRANSITION                                              *   *    B
Q3 RS 0075 ADJUST TAPE                                                    91       SB
Q3 RS 0077 REVERSE TAPE                                                  091       SB
Q3 UA OTM2 TAPE REWIND CONTROL                                           064       SB

           Z. ALL OTHERS

ZO CL REL  RELATIVIZE SYMBOLIC DECK                                      116        B
ZO MU 704R MURA REFLECTIVE 704                                           253       SB

Z  CL THA1 THERMAL ANALYZER                                              248        B
Z  GL FIDO UTILITY PACKAGE. FUNCTIONS, INPUT, DIAGNOSTICS, OUTPUT 181       SB
Z  NA 011  ASSEMBLY TRANSLATOR - NYAP1 TO UA SAP1                       001   *    B
Z  NY BLI1 BASIC LOOP INITIALIZER                                        145       S R
Z  NY BLU1 BASIC LOOP UPDATER                                            145         R
Z  NY INS1 INTEGER TO NUMBER SCALER                                      145       S R
Z  NY LWR1 LOCATION TO WORKSPACE RETRIEVER                               145       S R
Z  NY NFS2 NUMBER TO FRACTION SCALER                                     145       S R
Z  NY WLD1 WORKSPACE TO LOCATION DISPERSER                               145       S R
Z  PK DSMB BINARY CARD DISASSEMBLY PROGRAM                               158   *
Z  RL 0079 TAPE COMPARE                                                  106        B
Z  RS 0076 TEST TAPE FOR READABILITY  (COUNT RECORDS)                    091       SB
Z  RS 0084 COUNT RECORDS FOR TAPE READABILITY                            091  127 SB
ZO CL REL  RELATIVIZE SYMBOLIC DECK                                      236        B
ZO UA OTM2 TAPE REWIND CONTROL                                           064       SB
ZO UA OTM4 TAPE REWIND CONTROL                                           097       SB
ZO UA PCS1 PUNCH DRUM CHECK SUM VERIFIER                                 065       S
ZO UA RWD1 READ-WRITE DRUM                                               054       S
ZO UA VCS1 VERIFY DRUM CHECK SUM                                         065       S
ZO UA ZCS1 CLEAR CORE STORAGE AND MAIN FRAME                             048       SB
ZO UA ZCS2 SET CORE STORAGE TO ZERO                                      119       SB
ZO UA ZDR1 CLEAR N DRUMS                                                 065       S
```

NEW SUBROUTINE CARDS RECEIVED

NAME		DIST. NO.	CLASS	NAME		DIST. NO.	CLASS
AT	MG1	233	D3	MU	RDI1	256	I3
AT	TPI	238	D2	MU	DPA2	↓	A1
AS	03	224	B3	MU	EXP1&2		B4
AS	09	↓	B1	MU	BPU1&2		J1
AS	14		C1	MU	PRF1, 2, &3	258	J3
AS	33		B2	NO	FCD	231	M2
CL	DPA1&2	223	A1	NO	SIG	240	D1
CL	DPD1	↓	A1	NO	INP	241	I3
CL	DPC1		A2	NA	135, 3	246	B1
CL	DET3		F3	NA	180. 1	245	I4
CL	SMD2		F3	NY	DMI1	232	F1
CL	SME2, 4, 5		F4	NY	DBD1	235	I4
CL	MVP1		F1	NY	DHL1	235	J4
CL	MIV3		F1	NY	PCV1	262	01
CL	MST3		F1	NY	PLV1	262	01
CL	MTR1		F1	NY	TFD1&2	255	KO
CL	MRT1		Q3	NY	FSC1	250	E2
CL	MNR3	236	F1	RS	0128	230	LO
CL	EL1&2	↓	ZO	RS	0129	230	J3
CL	PLT2		J5	RS	112X	249	I2
GL	DPPA	237	A1	RS	112Y	249	I2
GL	GAUS	237	D1	MU	PIF	258	J3
GM	CFR1	225	C3	GM	ZER1	↓	C2
GM	EIG2	↓	F2	CL	THA1	248	Z
GM	IEF1		C3	CL	OUD1	↓	Q1
GM	ITR1		E2	CL	PIN1&2		E2
GM	TIN2	247	E1	CL	DEQ		D2
MU	LBL3	251	I1	CL	PMC1		F1
MU	UBL1	251	Il	GM	DIN1	239	E1
MU	TTV1	253	N1	NA	1891	260	F2
MU	PRI1	258	J3	MU	LBL4	263	I1

NAME	DIST. NO.	CLASS	NAME	DIST. NO.	CLASS
MU RAT1	253	K2	MU RON1	263	I2
MU FRD1	│	N2	MU BPU3	263	J1
MU EAS2	│	N1	MU SQR2	263	B4
MU 704R	↓	ZO	MU ATN1	263	B1
MU RDI2	263	I3			
AS 0049	264	N1			
NO INTP	265	E1			
NA 65.1	266	A2			
PK EDIT	267	J9			
PK HILO	267	QO			
GI DBUG	270	N1			
CL SME6	273	F4			
CL MMD1	273	F1			
CL MMP2	273	F1			
RS 0140	274	I1			

NEW WRITE-UPS WITH NO CARDS

NAME	DIST. NO.
MU SBL2	251
MU CSC2	│
MU OCD1	│
MU IND1	│
WB CFT2	↓
NY SNAP	275

Since the New England Colleges, other than MIT, do not receive Share letters concerning changes and corrections, the following information is particularly for their benefit. These changes have been noted in the MIT SHARE Library.

SHARE DIST.S WHICH ARE ADDENDED BY LATER DIST.S

WH	001	#57	see #118, 126
GM	GPR2	#70	see #163
CL	MLD1	#85	see #187
NA	086.1	#87	see #191
NA	087.1	#87	see #191
RS	0077	#91	see #127
NY	OSC1	#79	see #174
CL	SME1	#116	see #222
CL	INT4	#116	see #222
CL	LSQ2	#116	see #146, 187
PK	NIDA	#144, 203	see #195
CL	RAN1	#139	see #187
NA	092.3	#149	see #192
NA	090.3	#149	see #169 - superseded by 90.5
NA	090.5	#169	see #192
NA	091.3	#169	see #192
NA	098.1	↓	see #192
NA	70.3	↓	see #190
PK	POWR	↓	see #203
NY	MR1	#151	see #217
UA	TSM2	↓	see #78
UA	SPH1	↓	see #86
NY	BPU3	#75	see #88
CL	AEQ1	↓	see #167
NY	PCP2	#145	see #188
NY	PLP2	#145	see #188
GE	ARCTN	#3	superseded by #55
UA	BDC1	↓	see #99
NY	CIG1	↓	see #109
NY	TRG1	↓	see #109

LETTERS ABOUT CORRECTIONS OR CHANGES

C- 4
6
7
12
13
14
58
60
61
68
70
71

C-22
23
24
25
28
29
74
77
78
79
80
86

C-32
35
38
41
44
47
88
89
90
92
96
97

C-48
50
54
55
56
57
98
99
100
112
113
114

C-125

C-131

SUPERSEDED ROUTINES

NA 030.1 s.s.	by	NA 30.3	#104
NA 31.1 s.s.	by	NA 31.3	#104
NA 32.1 s.s.	by	NA 32.2	#104
RL 0039 s.s.	by	RL 0078	#106
		&	
		RL 0115	#125
RL 0042 s.s.	by	RL 0065	#106
RS 0004 s.s.	by	RS 0004	#40
NA 012 s.s.	by	NA 012.2	#150
NA 90.3 s.s.	by	NA 90.5	#169
RS 0083 s.s.	by	RS 0083	#179
GE ARCTN s.s.	by	GE ARCTN	#55
GE SIN2 s.s.	by	GE SIN2	#33
UA TSM1 s.s.	by	UADBC1	#73
CS ARTN1 s.s.	by	CS ART2	#92
NA 91.1 s.s.	by	91.3	#169
NA 92.1 s.s.	by	92.3	#149
NA 90.1 s.s.	by	91.5	#169
RS 0084 s.s.	by	RS 0084	#127