I-L85(A)

# PROGRAMMING MANUAL

# LINC-8

DIGITAL EQUIPMENT CORPORATION • MAYNARD, MASSACHUSETTS

I-L85(A)

# LINC-8
# PROGRAMMING MANUAL

DIGITAL EQUIPMENT CORPORATION • MAYNARD, MASSACHUSETTS

# PREFACE

This document is derived from several works developed by persons outside of DEC. "Programming the LINC" LINC V16 Section 2 Programming and Use, April 1965, by Mary Allen Wilkes and Wesley A. Clark, Washington University, St. Louis, Mo., was used, with some changes felt appropriate, for the discussion of the instruction set of the LINC portion of the LINC-8. "A LINC Utility System" Technical Report 1, March 19, 1965, written by M.D. McDonald, S.R. Davisson, and J.R. Cox, Jr., Biomedical Computer Laboratory, Washington University, St., Louis, Mo., was used to provide a basis for the discussion of the LAP4 and GUIDE systems. To the above individuals, as well as others at the Computer Research Laboratory of Washington University, the National Institutes of Health, the National Aeronautics and Space Administration, and individual LINC users, we are greatly indebted.

# CONTENTS

CONTENTS (continued)

# CHAPTER 1
# INTRODUCTION

The Digital Equipment Corporation LINC-8 system is comprised of two subsystems: a standard Programmed Data Processor-8 (PDP-8); and the LINC subsystem consisting of a central processing portion, a display scope, and a dual tape transport. The two subsystems are interconnected by a special interface section which mediates the interchange of data and control, and both share a dual console. The LINC-8 is designed to operate in one of two modes. In the first mode, it operates as a standard basic PDP-8 computer system. In the second, it operates essentially as a LINC having certain special in/out and speed characteristics. Despite these differences and improvements, the LINC subsystem will often be referred to simply as the LINC throughout the manual.

In the LINC mode, the LINC section is controlled by an ordinary LINC program held in the upper half of the PDP-8 memory (which is arranged to correspond exactly to the standard LINC memory of 2048 words). The PDP-8 memory can be expanded to 32,768 words. The LINC section is designed to call and carry out all instructions of the LINC program except MTP, OPR, and a new instruction called EXC. Instructions of this excepted class, called the execute class, are carried out by interpretive routines held in the lower half of the PDP-8 memory, which also holds programs for the interpretation of console switch actions and for the conversion of Teletype input to LINC keyboard code. The interpretive program is named PROGOFOP (PROGram OF OPeration), and is automatically read into the PDP-8 memory from magnetic tape by a wired-in LOAD mode initiated at the console.

## 1-1 MANUAL ORGANIZATION

This manual presents programming information relating to the LINC subsystem of the LINC-8 computer. The registers, switches, and indicators referenced in this document are associated with the LINC section and are located on the left half of the LINC-8 dual console. The RIGHT SWITCHES used in the LINC mode of operation also serve as the switch register for PDP-8 operation. Programming information for the PDP-8 subsystem of the LINC-8 can be obtained from the PDP-8 Users Handbook, F-85.

The first two chapters of this document acquaint the reader with number systems, instructions, and programming examples. Chapters 3 and 4 discuss the LINC Utility System (GUIDE and LAP4) which provides the user with information necessary to use these basic system programs for compiling and manipulating LINC-8 programs.

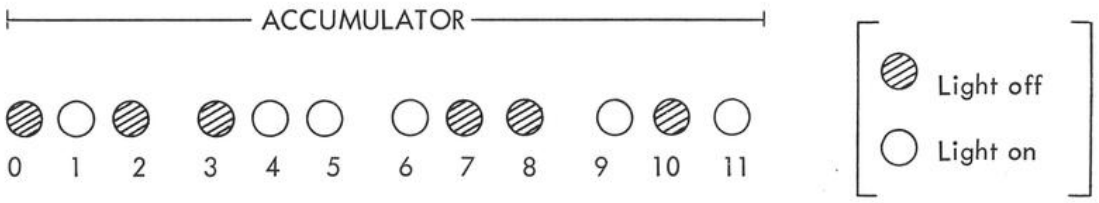
Like most digital computers, the LINC-8 operates by manipulating binary numbers held in various registers under the control of a program of instructions which are themselves coded as binary numbers and stored in other registers. LINC-8 instructions generally fall into types or classes, the instructions of a class having certain similarities. In this description, however, instructions are introduced as they are

relevant to the discussion. Reference to chart I appendix 2 is therefore recommended when class characteristics are described. Furthermore, not all LINC-8 instructions are described here in detail; therefore this document should be read in conjunction with the LINC-8 order code summary in appendix 2.

## 1-2 NUMBER SYSTEMS

The best way to begin studying number systems is to consider only a few of the registers and switches which are shown on the LINC-8 control console: the ACCUMULATOR (ACC) which is a register of twelve lights; the LINK bit (L); the LEFT and RIGHT SWITCHES, which are rows of twelve toggle switches each; and one lever switch labeled DO. The number systems and operation of several of the instructions can be understood in terms of these few elements.

The elements (bits) of each register or row of toggle switches are to be thought of as numbered from left to right starting with 0. This will serve to identify the elements and to relate them to the numerical value of the binary integer held in the register. C(ACC) denotes the contents of the accumulator register, etc. If the accumulator is illuminated thus




then the binary number stored in the accumulator is

C(ACC) = 010 011 100 101 (binary)

which has the decimal value

$$C(ACC) = 2^{10} + 2^7 + 2^6 + 2^5 + 2^2 + 2^0$$

$$= 1024 + 128 + 64 + 32 + 4 + 1$$

$$= 1253 \text{ (decimal)}$$

This can also be considered as an octal number by considering each group of three bits in turn. In this example, grouping and factoring proceed as follows:

$$C(ACC) = (2^{10}) + (2^7+2^6) + (2^5) + (2^2+2^0)$$

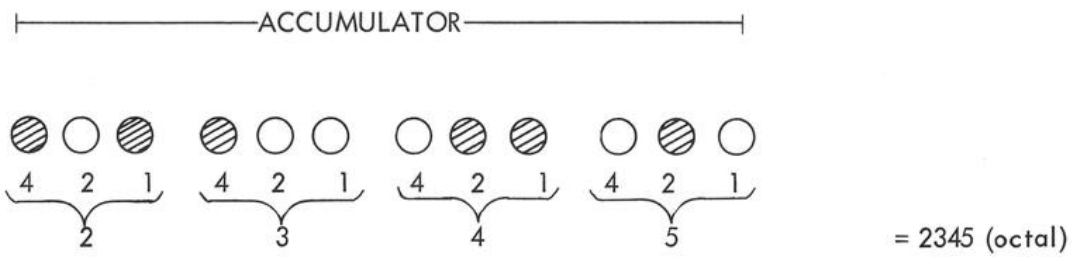
$$= (2^1)\cdot 2^9 + (2^1+2^0)\cdot 2^6 + (2^2)\cdot 2^3 + (2^2+2^0)\cdot 2^0$$

$$= (2)\cdot 8^3 + (3)\cdot 8^2 + (4)\cdot 8^1 + (5)\cdot 8^0$$

$$= 2 \quad 3 \quad 4 \quad 5$$

$$= 2345 \text{ (octal)}$$

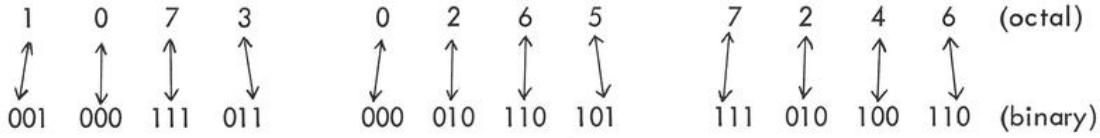
To put this more simply, each octal digit can be treated as an independent 3-bit binary number whose value (0, 1, ..., 7) can be obtained from the weights $2^2$, $2^1$, and $2^0$:




= 2345 (octal)

This ease of representation (the eight possible combinations within a group are easily perceived and remembered) is the principal reason for using octal numbers; the octal system can be viewed simply as a convenient notational system for representing binary numbers. Of course, octal numbers can also be manipulated arithmetically.

Translation from one system to the other is easily accomplished in either direction. Here are some examples:




Sometimes it is useful to view the contents of a register as a signed number. One of the bits must be reserved for the sign of the number. The leftmost bit is therefore identified as the sign bit (0 for +, 1 for −). To change the sign of a binary number, complement the number (replace all 0's by 1's and vice versa). Examples:

000 000 000 011 = +3

111 111 111 100 = -3

011 111 111 111 = +3777
100 000 000 000 = -3777

} The largest positive and negative octal integers in the 12-bit signed-number system.

The pair of binary numbers 101111110011 and 010000001100 ($5763_8$ and $2014_8$) are complements of each other, and denote the complement of the number N by $\overline{N}$. Note that the sum of each binary digit and its complement is the number 1, and that the sum of each octal digit and its complement is the number 7. Note also that there are two representations of 0:

000 000 000 000 = +0

111 111 111 111 = -0

Note finally that the sum of any binary number and its complement is always −0 in this system.

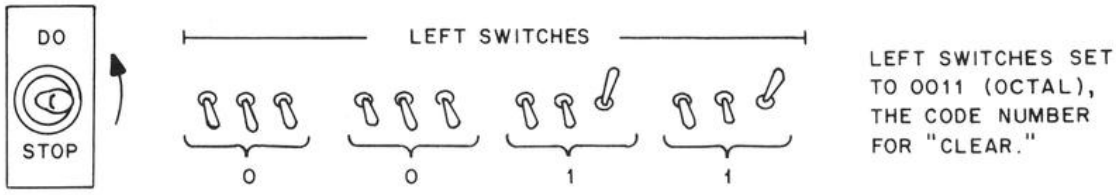
CLR, COM,
ATR, RTA, RSW

# CHAPTER 2
# INSTRUCTIONS

## 2-1 SIMPLE INSTRUCTIONS

LINC-8 instructions themselves are encoded as binary numbers and held in various registers. The simplest of these instructions, namely those which operate only on the accumulator, are described first with reference to the LEFT SWITCHES.

Raising the DO lever (DO means "do toggle instruction") causes the LINC to execute the instruction whose binary code number is held in the LEFT SWITCHES. The LINC then halts. For example, if the LEFT SWITCHES are set to the code number for the instruction CLEAR, which happens to be $0011_8$, and the DO lever is then momentarily raised, the ACCUMULATOR lights all go out as does the LINK bit light, so that C(ACC) = 0, and C(L) = 0. In setting a switch, up corresponds to 1.



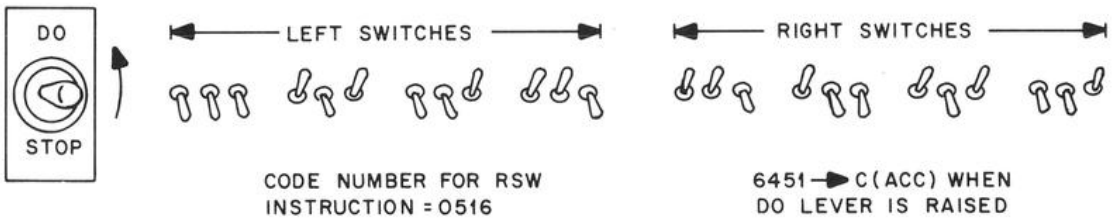

Briefly: If C(LEFT SWITCHES) = $0011_8$, DO has the effect $0 \rightarrow$ C(ACC) and $0 \rightarrow$ C(L). (Read "0 replaces the contents of the accumulator," etc.).

Clear (or CLR) is an instruction of the class known as miscellaneous instructions. A second miscellaneous class instruction, COM (complement), with the code number $0017_8$, directs the LINC to complement the contents of the accumulator and therefore has the effect $\overline{C(ACC)} \rightarrow C(ACC)$. (Read: "the complement of the contents of the accumulator replaces the contents of the accumulator.")

Two other instructions of this class transfer information between the accumulator and the relay register. The relay register, displayed on the control console, operates six relays which can be used to control or run external equipment. An instruction with the code $0014_8$, called ATR (accumulator to relay), directs the LINC to transfer the contents of the right half of the accumulator, i.e., the rightmost six bits, into the relay register. The accumulator itself is not changed when the instruction is executed. Another instruction, called RTA (relay to accumulator), $0015_8$, causes the LINC to clear the accumulator and then transfer the contents of the relay register into the right half of the accumulator. In this case the relay register is not changed and the left half of the accumulator remains cleared (i.e., contains 0's).

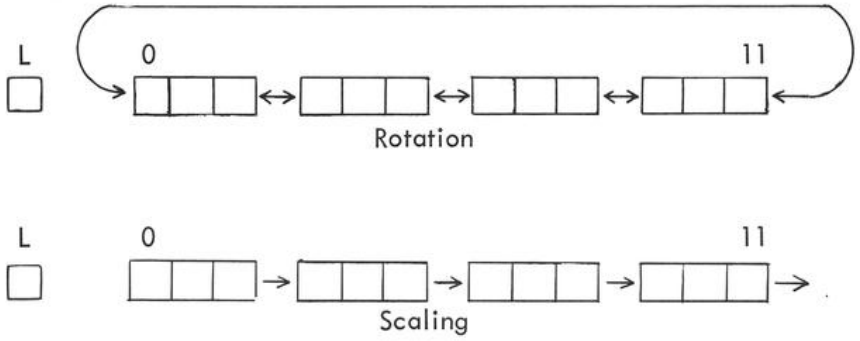
Another instruction called RSW (right switches), $0516_8$, directs the LINC to copy the contents of the RIGHT SWITCHES into the accumulator. By setting the LEFT SWITCHES to $0516_8$, the RIGHT SWITCHES to whatever value wanted in the accumulator, and then momentarily raising the DO lever,

the operator can change the contents of the accumulator to any desired new value. The drawing shows how the switches should be set to put the number $6451_8$ into the accumulator:




## 2-2 SHIFTING

After a number has been put into the accumulator it can be repositioned (shifted) to the right or left. There are two ways of shifting: rotation, in which the end-elements of the accumulator are connected together to form a closed ring, and scaling, in which the end-elements are not so connected.




Examples of shifts of one place:

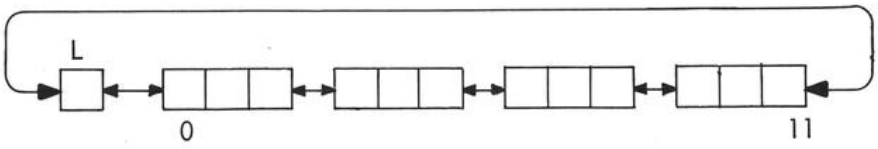
| | Effect of rotating right 1 place | Effect of scaling right 1 place | |
|---|---|---|---|
| Before | 000 000 011 001 | 000 000 011 001 = +25 | (decimal) |
| After | 100 000 001 100 | 000 000 001 100 = +12 | |
| Before | 111 111 100 110 | 111 111 100 110 = −25 | (decimal) |
| After | 011 111 110 011 | 111 111 110 011 = −12 | |

Note that, in scaling, bits are lost to the right, which amounts to an error of rounding off; the original sign is preserved in the sign bit and replicated in the bit positions to the right of the sign bit. This has the effect of reducing the size of the number by powers of two (analogous to moving the decimal point in decimal calculations).

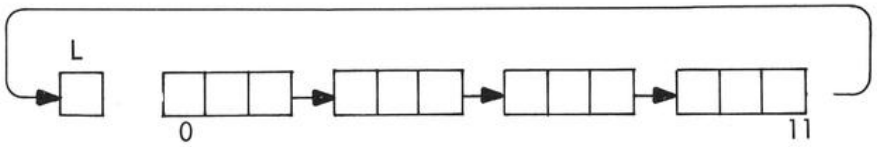
## ROR, ROL, SCR

The LINC has three instructions, called the shift class instructions, which shift the contents of the accumulator: rotate right, rotate left, and scale right. Unlike the simple instructions considered so far, the code number for a shift class instruction includes a variable element which specifies the number of places to shift. For example, write ROL n (rotate the contents of the accumulator n places to the left), where n can be any number from $0\text{-}17_8$.

As a further variation of the shift class instructions, the link bit can be adjoined to the accumulator during rotation to form a 13-bit ring as shown below, or to bit 11 of the accumulator during scaling to preserve the low order bit scaled out of the accumulator:

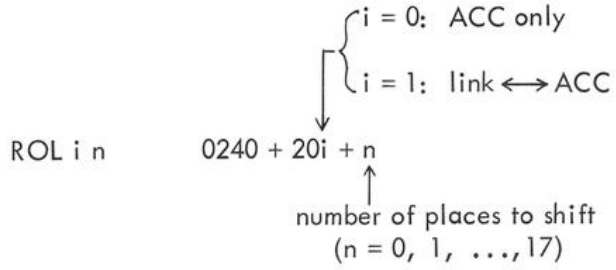



Rotation with Link Bit

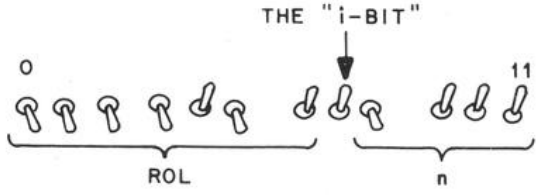



Scaling with Link Bit

The code number of a shift class instruction, e.g., rotate left, therefore includes the number of places to shift and an indication of whether or not to include the link bit. Use the full expression ROL i n, which has the octal coding:




so that, for example, rotate ACC left 3 places has the code 0243, and rotate ACC with link left 7 places has the code 0267. Note the correspondence between the code terms and bit positions of the binary-coded instruction as it appears, for example, in the LEFT SWITCHES:




ROL i 7

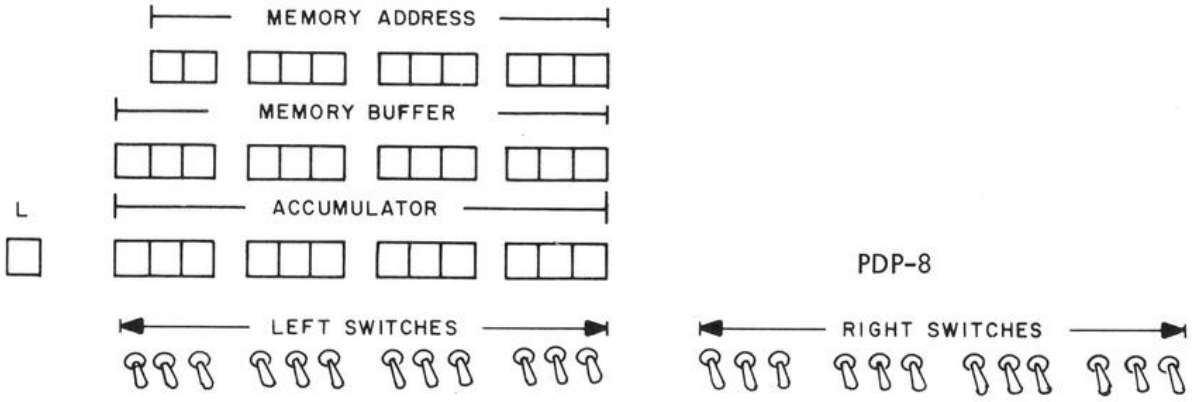
CODE NUMBER = 0267

Similar coding is used with ROR i n (rotate right), 300 + 20i + n, and SCR i n (scale right), 340 + 20i + n.

## 2-3 LINC MEMORY AND MEMORY REFERENCE INSTRUCTIONS

Before proceeding to other instructions, it is necessary to introduce the LINC memory. This memory is to be regarded as a set of $1024_{10}$ registers† each holding 12-bit binary numbers in the manner of the accumulator. These memory registers are numbered 0, 1, ..., $1023_{10}$, or 0, 1, ..., $1777_8$, and reference is made to "the contents of register 3," C(3), "the contents of register X," C(X), etc., referring to "3" and "X" as memory addresses.

The memory actually consists of a remotely-located array of magnetic storage elements with related electronics, but for introductory purposes view it in terms of two registers of lights, namely the memory address register and the memory buffer register:




By using these two registers in conjunction with the LEFT SWITCHES it is possible to find out what values the memory registers contain. For example, to find the contents of register 3, set the RIGHT SWITCHES to memory address 0003 and then operate the key labeled EXAM. As 0003 appears in the memory address register, the contents of register 3 appear in the memory buffer register. By setting the RIGHT SWITCHES to a memory address and pushing EXAM, the contents of any register in the LINC memory may be examined.

The contents of any selected memory register may be changed by using both the LEFT and RIGHT SWITCHES and the key marked FILL. For example, to make the memory register whose address is 700 contain −1 (i.e., $7776_8$) set memory address 0700 into the RIGHT SWITCHES. Set the LEFT SWITCHES to 7776 and operate the FILL key. A 0700 appears in the memory address register and 7776 appears in the memory buffer register, indicating that the contents of register 700 are now 7776. Whatever value register 700 may have contained before FILL was pushed is lost, and the new value takes its place. In this way any register in the LINC memory can be filled with a new number.

None of the LINC instructions make explicit reference to the memory address register or memory buffer register; rather, in referring to memory register X, an instruction may direct the LINC implicitly to put the address X into the memory address register and the contents of register X, C(X), into the memory buffer register.

†See appendix 3 for the discussion of extended memory programming.

## 2-3.1 The Store-Clear Instruction (4000 + X)

Now it is possible to describe the first of the memory reference instructions, STC X (store-clear X), which has the code number 4000 + X, where $0 \leq X \leq 1777_8$. (From now on only octal numbers will be used for addresses.) Execution of STC X has two effects: 1) the contents of the accumulator are copied into memory register X, C(ACC)→C(X), and 2) the accumulator is then cleared, 0→C(ACC). (The link bit is not cleared.) Thus, for example, if C(ACC) = 0503 and C(671) = 2345, and the code number for STC 671, i.e., 4671, is set into the LEFT SWITCHES, raising the DO level puts 0 into the accumulator and 0503 into register 671. The original contents of register 671 are lost.

It will be clear that the memory can be filled with new numbers at any time either by using the FILL key and the switches, or by loading the accumulator from the RIGHT SWITCHES with the RSW instruction and the DO lever and then storing the accumulator contents with the STC X instruction and the DO lever.

## 2-3.2 The ADD Instruction and Binary Addition (2000 + X)

STC is one of three full-address class instructions. Another instruction in this class, ADD X, has the code number 2000 + X where $0 \leq X \leq 1777$. Execution of ADD X has the effect of adding the contents of memory register X to the contents of the accumulator, i.e., C(X) + C(ACC)→ C(ACC). If the accumulator is first cleared, ADD X has the effect of merely copying into the accumulator the contents of memory register X, i.e., C(X)→C(ACC). In any case, the contents of memory register X are unaffected by the instruction.

The addition itself takes place in the binary system,* within the limitations of the 12-bit registers. The basic rules for binary addition are simple: 0 + 0 = 0; 1 + 0 = 1; 1 + 1 = 10 (i.e., zero, with one to carry). A carry arising from the leftmost column (end-carry) is brought around and added into the rightmost column (end-around carry). Some examples (begin at the rightmost column as in decimal addition):

```
001  111  010  001
000  010  111  001
 11  111  1     1   (Carries)
010  010  001  010  (Sum)
```

```
111  100  010  011
001  010  010  000
1 11       1        (Carries)
000  110  100  011
                1    (End-around carry)
               11    (Carries)
000  110  100  100  (Sum)
```

*See Volume 16, Section 1, "An Introduction to Binary Numbers and Binary Arithmetic," Irving H. Thomae.

HLT

The reader should try some examples of his own, and verify the fact that adding a number to itself with end-around carry is equivalent to rotating left one place. With signed-integer interpretation, some other examples are:

```
000 000 000 101 = +5          111 111 111 010 = −5
111 111 111 100 = −3          111 111 111 100 = −3
1 111 111 111                 1 111 111 11
  000 000 000 001               111 111 110 110
                ←1                            ←1
                1
  000 000 000 010 = +2          111 111 110 111 = −8 (decimal)
```

It can be seen that subtraction of the number N is accomplished by addition of the complement of N, $\overline{N}$. Of course, if either the sum or difference is too large for the accumulator to hold, the result of the addition may not be quite the desired number. For example, adding 1 to the largest positive integer in this system ($+3777_8$) results in the largest negative integer ($-3777_8$). This is sometimes called overflowing the capacity of the accumulator.

## 2-3.3 Instruction Location Register

It is clear that the code numbers of a series of different instructions can be stored in consecutive memory registers. The LINC-8 is designed to execute this stored program of instructions by fetching and carrying out each instruction in sequence, using a special 10-bit register called the instruction location register (P), to hold the address of the next instruction to be executed. Using the FILL key and the LEFT and RIGHT SWITCHES already discussed, can, for example, put into memory registers 20-24 the code numbers for a series of instructions which divide by 8 the number held in memory register 30 and store the result in memory register 31:

| Memory Address | Memory Buffer | | Effect |
|---|---|---|---|
| Start → 20 | CLR | 0011 | Clear the accumulator. |
| 21 | ADD 30 | 2030 | Add the contents of register 30 to the accumulator. |
| 22 | SCR 3 | 0343 | Scale C(ACC) right 3 places to divide by 8. |
| 23 | STC 31 | 4031 | Store in register 31. |
| 24 | HLT | 0000 | Halt the computer. |
| ⋮ | ⋮ | ⋮ | |
| 30 | N | N | Number to be divided by 8. |
| 31 | → N/8 | N/8 | Result. |

Example 1 Simple Sequence of Instructions

Use the FILL key and the LEFT and RIGHT SWITCHES to put the code numbers for the instructions into memory registers 20-24 and the number to be divided into register 30. Operating the console key labeled START 20 directs the LINC to begin executing instructions at memory register 20. That is, the value 20 replaces the contents of the instruction location register. As each instruction of the stored program is executed, the instruction location register is increased by 1, $C(P) + 1 \rightarrow C(P)$. When the instruction location register contains 24, the computer encounters the instruction HLT, code 0000, which halts the machine. To run the program again, merely operate the START 20 key. (The code numbers for the instructions stay in memory registers 20-24 unless they are deliberately changed.)

## 2-3.4 The Jump Instruction (6000 + X)

The last full-address instruction, JMP X, code 6000 + X, has the effect of setting the instruction location register to the value X; $X \rightarrow C(P)$. That is, the LINC, instead of increasing the contents of the instruction location register by one and executing the next instruction in sequence, is directed by the JMP instruction to get its next instruction from memory register X. In the above example having a JUMP to 20 instruction, code 6020, in memory register 24 (in place of HLT) would cause the computer to repeat the program endlessly. If the program were started with the START 20 switch, the instruction location register (P) would hold the succession of values: 20, 21, 22, 23, 24, 20, 21, etc. (Later instructions will be introduced which increase C(P) by extra amounts, causing it to skip.)

JMP X has one further effect: if JMP 20, 6020, is held in memory register 24, then its execution causes the code for JMP 25 to replace the contents of register 0; i.e., $6025 \rightarrow C(0)$. More generally, if JMP X is in any memory register p, $0 \leq p \leq 1777$, then its execution causes JMP p+1 $\rightarrow C(0)$.

| Memory Address | Memory Buffer | | Effect |
|---|---|---|---|
| 0 | JMP p+1 | 6000 + p+1 | |
| ⋮ | ⋮ | ⋮ | |
| → p | JMP X | 6000 + X | X → C(P), and JMP p+1 → C(0). |
| p+1 | ⋮ | ⋮ | |
| ⋮ | | | |
| X | → - | - | Next instruction. |

This JMP p+1 code replaces the contents of register 0 every time a JMP X instruction is executed unless X = 0, in which case the contents of 0 are unchanged. Use of memory register 0 in this way is relevant to a programming technique involving subroutines which is described later†.

†See appendix 3 for a discussion of JMP X when using extended memory.

The following programming example illustrates many of the features described so far. It finds one-fourth of the difference between two numbers $N_1$ and $N_2$, which are located in registers 201 and 202, and leaves the result in register 203 and in the accumulator. After filling consecutive memory registers 175-210 with the appropriate code and data numbers, the program must be started at memory register 175. Since there is no START 175 key on the console, this is done by setting the RIGHT SWITCHES to 0175 and operating the console key labeled START RS (start RIGHT SWITCHES).

| Memory Address | Memory Buffer | | Effect |
|---|---|---|---|
| Start → 175 | CLR | 0011 | 0 → C(ACC). |
| 176 | ADD 201 | 2201 | $N_1$ → C(ACC). |
| 177 | COM | 0017 | Forms $-N_1$. |
| 200 | JMP 204 | 6204 | Jumps around data; 204 → C(P), and JMP 201 → C(0). |
| 201 | $N_1$ | $N_1$ | Data and result. |
| 202 | $N_2$ | $N_2$ | |
| 203 | $(N_2-N_1)/4$ | $(N_2-N_1)/4$ | |
| 204 | ADD 202 | 2202 | $(N_2-N_1)$ → C(ACC). |
| 205 | SCR 2 | 0342 | Divides by 4. |
| 206 | STC 203 | 4203 | Stores result in 203; C(ACC) → C(203); 0 → C(ACC). |
| 207 | ADD 203 | 2203 | Recovers result in ACC. |
| 210 | HLT | 0000 | Halts the LINC. |

Example 2 Simple Sequence Using the Jump Instruction

In executing this program, the instruction location register holds the succession of numbers: 175, 176, 177, 200, 204, 205, 206, 207, 210.

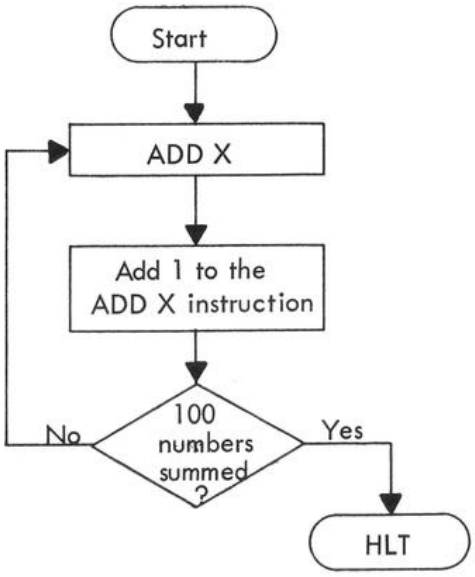
## 2-4 ADDRESS MODIFICATION AND PROGRAM LOOPS

Frequently a program of instructions must deal with a large set of numbers rather than just one or or two. For example, suppose one wishes to add $100_8$ numbers and that the numbers are stored in the memory in registers 1000-1077. The sum is to go into memory register 1100. It is possible, of course, to write out all the instructions necessary to do this,

| Memory Address | Memory Buffer | | Effect |
|---|---|---|---|
| → 20 | CLR | 0011 | 0→C(ACC); 0→C(L). |
| 21 | ADD 1000 | 3000 | Add 1st number. |
| 22 | ADD 1001 | 3001 | Add 2nd number. |
| 23 | ADD 1002 | 3002 | Add 3rd number. |
| 24 | ADD 1003 | 3003 | Add 4th number. |
| | etc. | etc. | etc. |

but it is easy to see that the program would be more than $100_8$ registers long. A more complex, but considerably shorter, program can be written using a programming technique known as address modification. Instead of writing $100_8$ ADD X instructions, write only one ADD X instruction, which is repeated $100_8$ times, modifying the X part of the ADD X instruction each time it is repeated. In this case the computer first executes an ADD 1000 instruction; the program then adds one to the ADD instruction itself and restores it, so that it is now ADD 1001. The program then jumps back to the location containing the ADD instruction and the computer repeats the entire process, this time executing an ADD 1001 instruction. In short, the program is written so that it changes its own instructions while running.

The process might be diagrammed:




This technique introduces the additional problem of deciding when all 100 numbers have been summed and halting the computer. In this context a new instruction AZE (accumulator zero), code 0450, should be introduced. This is one of a class of instructions known as skip instructions; it directs the LINC to skip the instruction in the next memory register when C(ACC) = ±0 ($0000_8$ or $7777_8$). If C(ACC) ≠ 0, the computer does not skip. For example, if C(ACC) = 7777, and one writes:

| Memory Address | Memory Buffer | |
|---|---|---|
| → p | AZE | 0450 |
| p+1 | - | - |
| p+2 | - ← | - |

the computer takes the next instruction from p+2. That is, when the AZE instruction in register p is executed, p+2 replaces the contents of the instruction location register, and the computer skips the instruction at p+1. If C(ACC) $\neq$ 0, then p+1 → C (P) and the computer executes the next instruction in sequence as usual.

The following example sums the numbers in memory registers 1000-1077 and puts the sum into memory register 1100, using address modification and the AZE instruction to decide when to halt the computer. (Square brackets indicate registers whose contents change while the program is running.)

| Memory Address | Memory Buffer | | Effect |
|---|---|---|---|
| 10 | ADD 1000 | 3000 | Constants used by program. |
| 11 | 1 | 0001 | |
| 12 | -(ADD 1100) | 4677. | |
| ⋮ | ⋮ | ⋮ | |
| Start → 20 | CLR | 0011 | Code for ADD 1000 → C(25). 0 → C(ACC). |
| 21 | ADD 10 | 2010 | |
| 22 | STC 25 | 4025 | |
| 23 | STC 1100 | 5100 | 0 → C(1100), for accumulating sum. |
| 24 | → CLR | 0011 | Clear ACC and add C(X) to C(ACC). |
| 25 | [ADD X] | [2000+X] | |
| 26 | ADD 1100 | 3100 | Sum so far + C(ACC) → C(ACC). |
| 27 | STC 1100 | 5100 | Sum so far → C(1100). |
| 30 | ADD 25 | 2025 | ADD X instruction in register 25 → C(ACC). Add 1 to C(ACC) and replace in register 25. |
| 31 | ADD 11 | 2011 | |
| 32 | STC 25 | 4025 | |
| 33 | ADD 25 | 2025 | C(25) + C(12) C(ACC). If C(25) = ADD 1100, then C(ACC) = 7777. |
| 34 | ADD 12 | 2012 | |
| 35 | AZE | 0450 | Skip to register 37 if C(ACC) = 7777. |
| 36 | JMP 24 | 6024 | If not, return and add next number. |
| 37 | HLT ← | 0000 | When C(ACC) = 7777, all numbers have been summed. Halt the computer. |
| ⋮ | ⋮ | ⋮ | |
| 1000 | $N_1$ | $N_1$ | Numbers to be summed. |
| 1001 | $N_2$ | $N_2$ | |
| ⋮ | ⋮ | ⋮ | |
| 1076 | $N_{77}$ | $N_{77}$ | |
| 1077 | $N_{100}$ | $N_{100}$ | |
| 1100 | [Sum] | [Sum] | |

Example 3 Summing a Set of Numbers Using Address Modification

The instructions at locations 20-22 initially set the contents of memory register 25 to the code for ADD 1000. At the end of the program, register 25 will contain 3100, the code for ADD 1100. Adding (in registers 33 and 34) C(25) to C(12), which contains the complement of the code for ADD 1100, results in the sum 7777 only when the program has finished summing all $100_8$ numbers. This repeating sequence of instructions is called a loop, and instructions such as AZE can be used to control the number of times a loop is repeated. In this example the instructions in locations 24-36 will be executed $100_8$ times before the computer halts.

The following program scans the contents of memory registers 400 through 450 looking for registers which do not contain zero. Any non-zero entry is moved to a new table beginning at location 500; this has the effect of packing the numbers so that no registers in the new table contain zero. When the program halts, the accumulator contains the number of non-zero entries.

| Memory Address | Memory Buffer | | Effect |
|---|---|---|---|
| 4 | ADD 400 | 2400 | Constants used by the program. |
| 5 | STC 500 | 4500 | |
| 6 | 1 | 0001 | |
| 7 | -(ADD 451) | 5326 | |
| 10 | -(STC 500) | 3277 | |
| ⋮ | ⋮ | ⋮ | |
| Start → 100 | CLR | 0011 | Code for ADD 400→C(106). |
| 101 | ADD 4 | 2004 | |
| 102 | STC 106 | 4106 | |
| 103 | ADD 5 | 2005 | Code for STC 500→C(112). |
| 104 | STC 112 | 4112 | |
| 105 | → CLR | 0011 | |
| 106 | [ADD 400] | [2000+X] | C(X)→C(ACC). |
| 107 | AZE | 0450 | If C(ACC) = 0, skip to location 111. |
| 110 | JMP 112 | 6112 | C(ACC) ≠ 0, therefore JMP to location 112. |
| 111 | JMP 116 ↲ | 6116 | C(ACC) = 0, therefore JMP to location 116. |
| 112 | →[STC 500] | [4000+X] | Store non-zero entry in new table. |
| 113 | ADD 6 | 2006 | Add 1 to the STC instruction in register 112. |
| 114 | ADD 112 | 2112 | |
| 115 | STC 112 | 4112 | |
| 116 | →ADD 6 | 2006 | Add 1 to the ADD instruction in register 106. |
| 117 | ADD 106 | 2106 | |
| 120 | STC 106 | 4106 | |
| 121 | ADD 106 | 2106 | C(106) + C(7)→C(ACC). If C(106) = ADD 451, then C(ACC) = 7777. |
| 122 | ADD 7 | 2007 | |
| 123 | AZE | 0450 | If C(ACC) = 7777, skip to location 125. |
| 124 | JMP 105 | 6105 | If not, return to examine next number. |
| 125 | ADD 112 ↲ | 2112 | If C(ACC) = 7777, then number of non-zero entries →C(ACC) and computer halts. |
| 126 | ADD 10 | 2010 | |
| 127 | HLT | 0000 | |

Example 4 Packing a Set of Numbers

ADA

At the end of the program, register 106 contains the code for ADD 451, and all numbers in the table have been examined. If, say, 6 entries were found to be non-zero, registers 500-505 will contain the non-zero entries, and register 112 will contain the code for STC 506. Therefore by adding C(112) to the complement of the code for STC 500 (in registers 125-126 above), the accumulator is left containing 6, the number of non-zero entries.

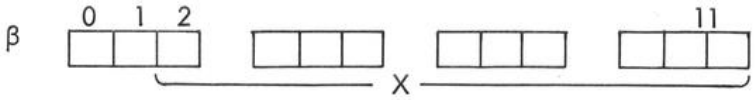
## 2-5 INDEX CLASS INSTRUCTIONS I

### 2-5.1 Indirect Addressing

The largest class of LINC instructions, index class, addresses the memory in a somewhat involved manner. The instructions ADD X, STC X, and JMP X are called full address instructions because the 10-bit address X, $0 \leq X \leq 1777$, can address directly any register in the $2000_8$ register memory. The index class instructions, however, have only 4 bits reserved for an address, and can therefore address only memory registers $1_8$-$17_8$. The instruction ADA i $\beta$ (add to accumulator), $1100_8 + 20i + \beta$, is typical of the index class:

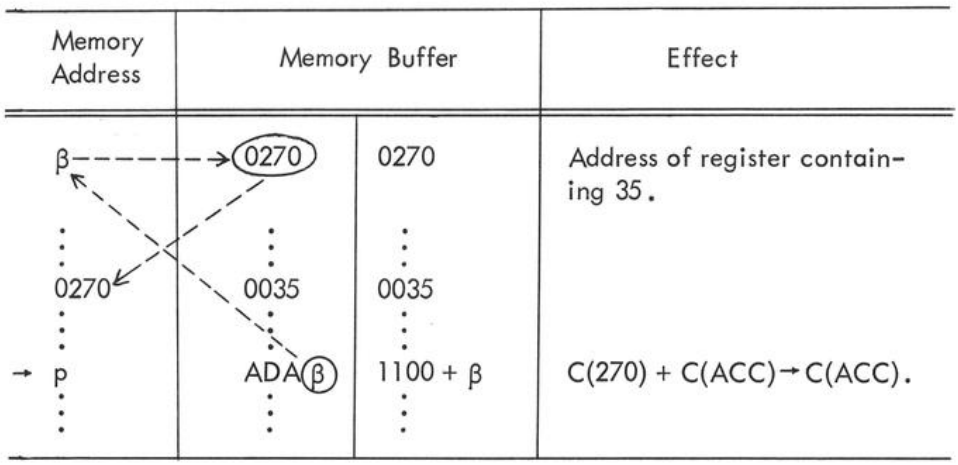
$$\text{ADA i } \beta \qquad \underset{\text{ADA}}{1100} + \underset{i = 0 \text{ or } 1}{20i} + \underset{1 \leq \beta \leq 17}{\beta}$$

Memory register $\beta$ should be thought of as containing a memory address, X, in the rightmost 10 bits,




and X($\beta$), as meaning the right 10-bit address part of register $\beta$. The leftmost bit can have any value, and, for the present, bit 1 must be 0. In addressing memory register $\beta$, an index class instruction tells the computer where to find the memory address to be used in executing the instruction. This is called indirect addressing.

For example, to add the value 35 to the contents of the accumulator, with 35 held in memory register 270, use the ADA instruction in the following manner:




Note that the ADA instruction does not tell the computer directly where to find the number 35; it tells the computer instead where to find the address of the memory register which contains 35. By using memory registers 1-17 in this way, the index class instructions can refer to any register in the memory.

Two other index class instructions, LDA i β (load accumulator), and STA i β (store accumulator), are used in the following program which adds the contents of memory register 100 to the contents of register 101 and stores the result in 102. The LDA i β instruction, code 100 + 20i + β, clears the accumulator and copies into it the contents of the specified memory register. STA i β, code 1040 + 20i + β, stores the contents of the accumulator in the specified memory register; it does not, however, clear the accumulator. Addition with ADA uses 12-bit end-around carry arithmetic.

| Memory Address | Memory Buffer | | Effect |
|---|---|---|---|
| 10 | $X_1$ | 0100 | Address of $N_1$. |
| 11 | $X_2$ | 0101 | Address of $N_2$. |
| 12 | $X_3$ | 0102 | Address of $(N_1+N_2)$. |
| Start → 30 | LDA 10 | 1010 | $N_1$, i.e., C(100), → C(ACC). |
| 31 | ADA 11 | 1111 | $N_2$, i.e., C(101), + C(ACC) → C(ACC). |
| 32 | STA 12 | 1052 | $N_1+N_2$ → C(102). |
| 33 | HLT | 0000 | |
| 100 | $N_1$ | - | |
| 101 | $N_2$ | - | |
| 102 | $[N_1+N_2]$ | [-] | |

Example 5 Indirect Addressing

SAE

## 2-5.2 Index Registers and Indexing

When i is used with an index class instruction, that is, when i = 1, the computer is directed to add 1 to the X part of memory register β before it is used to address the memory. This process is called indexing, and registers 1-17 are frequently referred to as index registers. In the example below, −6 is loaded into the accumulator after index register β is indexed from 1432 to 1433 by the LDA i β instruction.

| Memory Address | Memory Buffer | | Effect |
|---|---|---|---|
| β | [X] | [1432] | Address minus 1 of register containing 7771. |
| ⋮ | ⋮ | ⋮ | |
| → p | LDA i β | 1020+β | X + 1, i.e., 1433, →C(β), and C(1433) →C(ACC). |
| ⋮ | ⋮ | ⋮ | |
| 1432 | - | - | |
| 1433 | −6 | 7771 | |

When the LDA i β instruction is executed, the value X(β) + 1 replaces the address part of register β (the leftmost 2 bits of register β are unaffected). This new value, 1433, is now used to address the memory. Note that if the LDA instruction at p were repeated, it would deal with the contents of register 1434, then 1435, etc. Utility of index registers in scanning tables of numbers should be obvious.

Indexing involves only 10-bit numbers, and does not involve end-around carry. Therefore the address following 1777 is 0000. (The same kind of indexing takes place in the instruction location register, which counts from 1777 to 0000.)

The following example using indexing introduces another index class instruction, SAE i β (skip if accumulator equals), code 1440 + 20i + β. This instruction causes the LINC to skip one register in the sequence of programmed instructions when the contents of the accumulator exactly match the contents of the specified memory register. If there is no match, the computer goes to the next instruction in sequence as usual. The program example clears (stores 0000 in) the set of memory registers 1400-1777; the SAE instruction is used to decide whether the last 0000 has been stored.

| Memory Address | Memory Buffer | | Effect |
|---|---|---|---|
| 3 | [X] | [1377] | Initial address minus 1 for the STA instruction. |
| 4 | 356 | 0356 | Address of test number. |
| ⋮ | ⋮ | ⋮ | |
| Start → 350 | →CLR | 0011 | Clear the accumulator. |
| 351 | STA i 3 | 1063 | Index the contents of register 3; store C(ACC) in the memory register whose address = X(3). |
| 352 | ADD 3 | 2003 | C(3) → C(ACC). |
| 353 | SAE 4 | 1444 | Skip to 0355 if C(ACC) = C(356). |
| 354 | JMP 350 | 6350 | If not, return to store 0000 in next register. |
| 355 | HLT | 0000 | Halt the computer. |
| 356 | 1777 | 1777 | |

Example 6 Indexing to Clear a Set of Registers

When the program halts at register 355, register 3 will contain 1777. The SAE instruction is used here (as the AZE instruction was used in earlier examples) to decide when to stop the computer. The instructions in registers 350-354, the loop, are executed $400_8$ times before the program halts. A 0 is first stored in register 1400, next in 1401, etc.

Another program scans the memory to see if a particular number, Q, appears in any memory register 0-1777. Q is to be set in the RIGHT SWITCHES, and the address of any register containing Q is to be left in the accumulator.

| Memory Address | Memory Buffer | | Effect |
|---|---|---|---|
| 17 | [X] | [-] | Address of register whose contents are to be compared with RIGHT SWITCHES. |
| Start → 20 | RSW | 0516 | C(RS) → C(ACC). |
| 21 | →SAE i 17 | 1477 | Index register 17, and compare C(ACC) with C(X). |
| 22 | JMP 21 | 6021 | If not equal, return for next test. |
| 23 | CLR | 0011 | If equal, clear ACC, copy address of register containing Q into ACC, and halt. |
| 24 | ADD 17 | 2017 | |
| 25 | HLT | 0000 | |

Example 7 Memory Scanning

ADM, BCL,
BSE, BCO

If no memory register 0-1777 contains the number Q, the program will run endlessly. The location of the first register to be tested depends on the initial contents of index register 17.

An index class instruction, ADM i β (add to memory), code 1140 + 20i + β, adds the contents of the specified memory register to C(ACC), using 12-bit end-around carry arithmetic (as ADD or ADA). The result is left, however, not only in the accumulator but in the specified memory register as well. The bit clear instruction, BCL i β, code 1540 + 20i + β, is one of three index class instructions which performs a so-called "logical" operation. BCL is used to clear selected bits of the accumulator. For every bit of the specified memory register which contains 1, the corresponding bit of the accumulator is set to 0.

In the following program two sets of numbers are summed term by term. The first set of numbers, each 6 bits long, is in registers 500-577, bits 6 through 11; bits 0-5 contain unwanted information. The second set of numbers is in registers 600-677, and the sums replace the contents of registers 600-677.

| Memory Address | Memory Buffer | | Effect |
|---|---|---|---|
| 3 | $[X_1]$ | [0477] | Initial address minus 1 of first set. |
| 4 | 0410 | 0410 | Address of BCL pattern. |
| 5 | $[X_2]$ | [0577] | Initial address minus 1 of second set. |
| 6 | 0411 | 0411 | Address of test number for halting. |
| ⋮ | ⋮ | ⋮ | |
| Start → 400 | → LDA i 3 | 1023 | Index X(3) and load number from first set into AC. |
| 401 | BCL 4 | 1544 | Clear the left 6 bits of the ACC. |
| 402 | ADM i 5 | 1165 | Index X(5). Add number from second set to C(ACC), and replace in memory. |
| 403 | CLR | 0011 | Check to see if finished. |
| 404 | ADD 3 | 2003 | |
| 405 | SAE 6 | 1446 | |
| 406 | JMP 400 | 6400 | C(3) ≠ C(411), i.e., ≠ 0577. |
| 407 | HLT | 0000 | C(3) = 0577; halt the program. |
| 410 | 7700 | 7700 | BCL pattern for clearing left half of ACC. |
| 411 | 0577 | 0577 | Test number for halting. |

Example 8 Summing Sets of Numbers Term by Term

## 2-5.3 Logic Instructions

The three logic instructions, BCL i β, BSE i β, and BCO i β, are best understood by studying the following examples. These instructions affect only the accumulator; the memory register M containing the bit pattern is unchanged.

BCL i $\beta$ bit clear code: $1540 + 20i + \beta$

Clear corresponding bits of the accumulator:

If C(M) = 010 101 010 101
and C(ACC) = 111 111 000 000
then C(ACC) = 101 010 000 000

BSE i $\beta$ bit set code: $1600 + 20i + \beta$

Set to 1 corresponding bits of the accumulator:

If C(M) = 010 101 010 101
and C(ACC) = 111 111 000 000
then C(ACC) = 111 111 010 101

BCO i $\beta$ bit complement code: $1640 + 20i + \beta$

Complement corresponding bits of the accumulator:

If C(M) = 010 101 010 101
and C(ACC) = 111 111 000 000
then C(ACC) = 101 010 010 101

These instructions have a variety of applications, some of which will be demonstrated later.

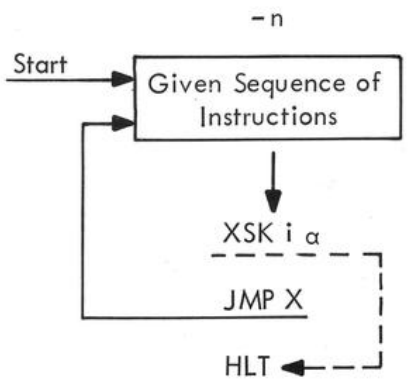
## 2-6 SPECIAL INDEX REGISTER INSTRUCTIONS

Before continuing with the index class, two special instructions which facilitate programming with the index class instructions will be introduced. These instructions do not use the index registers to hold memory addresses; rather they deal directly with the index registers and are used to change or examine the contents of an index register.

### 2-6.1 The Index and Skip Instruction

The index and skip instruction, XSK i $\alpha$, refers to registers 0-17 ($0 \leq \alpha \leq 17$).* It tests to see whether the address part of register $\alpha$ has its maximum value, i.e., 1777, and directs the LINC to skip the next register in the instruction sequence if 1777 is found. It will also, when i = 1, index the address part (X) of register $\alpha$ by 1. Like the index class instructions, XSK indexes register $\alpha$ before examining it, and it indexes from 1777-0000 without affecting the leftmost 2 bits. These 2 bits can therefore have any value. In particular, both can be set to the value 1 and XSK i $\alpha$ can be assumed to have the effect of skipping the next instruction when it finds the number 7777, (−0), in register $\alpha$. Now it is easy to see how to execute any given sequence of instructions exactly n times, where $n \leq 1777$ (octal):

*cf. $\beta$, $1 \leq \beta \leq 17$, which does not refer to register 0.




} -n stored in register α.

} Given sequence held in register X, X+1, etc.

} Index α and test. After 1st pass C(α) = -n + 1, after 2nd pass C(α) = -n + 2. After n passes C(α) = -n + n = -0 so skip over the JMP X instruction and halt.

For example, to store the contents of the accumulator in registers 350-357, using register 6 to count, the following short program can be written.

| Memory Address | Memory Buffer | | Effect |
|---|---|---|---|
| 5 | [X] | [0347] | Initial address minus 1 for STA instruction. |
| 6 | [-10] | [7767] | -n, where n = number of times to store C(ACC). |
| ⋮ | ⋮ | ⋮ | |
| Start → 200 | STA i 5 | 1065 | Index register 5 and store C(ACC). |
| 201 | XSK i 6 | 0226 | Index register 6 and test for X(6) = 1777. |
| 202 | JMP 200 | 6200 | X(6) ≠ 1777, return. |
| 203 | HLT | 0000 | X(6) = 1777, halt. |

Example 9 Index Registers Used as Counters

Using the XSK instruction with i = 0, which tests X(α) without indexing, example 6 which stores 0 in memory registers 1400-1777, can be more efficiently written:

| Memory Address | Memory Buffer | | Effect |
|---|---|---|---|
| 3 | [X] | [1377] | Initial address minus 1 for STA instruction. |
| ⋮ | ⋮ | ⋮ | |
| Start → 350 | CLR | 0011 | 0 → C(ACC). |
| 351 | → STA i 3 | 1063 | Index register 3 and store zero. |
| 352 | XSK 3 | 0203 | Test for X(3) = 1777. |
| 353 | JMP 351 | 6351 | X(3) ≠ 1777, return. |
| 354 | HLT ← | 0000 | X(3) = 1777, halt. |

Example 10 Indexing and Counting to Clear a Set of Registers

Here register 3 is indexed by the STA instruction; the XSK then merely tests to see whether X(3) = 1777, without indexing X(3). The reader should see that example 8 on page 22 can also be more efficiently programmed using XSK.

## 2-6.2 The SET Instruction

The second special instruction which is often used with the index class instructions is SET i $\alpha$, code $40 + 20i + \alpha$, where $\alpha$ again refers directly to the first $20_8$ memory registers, $0 \leq \alpha \leq 17$. In some of the examples presented earlier, the contents of index registers are changed, either as counter values or as memory addresses, while the program is running. Therefore, in order to rerun the program the index registers must be reset to their initial values.

The SET instruction directs the LINC to set register $\alpha$ to the value in any specified memory register. It is different from the instructions so far presented in that the instruction itself always occupies two consecutive memory registers, say p and p + 1:

| Memory Address | Memory Buffer | |
|---|---|---|
| p | SET i $\alpha$ | $40 + 20i + \alpha$ |
| p + 1 | c | c |
| p + 2 | – | – |
| ⋮ | ⋮ | ⋮ |

The computer automatically skips over the second register of the pair, p + 1; that is the contents of p + 1 are not interpreted as the next instruction. The next instruction after SET is always taken from p + 2.

The i-bit in the SET instruction does not control indexing. Instead, it tells the LINC how to interpret the contents of register p + 1. When i = 0, the LINC is directed to interpret C(p + 1) as the memory address for locating the value which will replace C(α). That is, register p + 1 is thought of as containing X,

| Memory Address | Memory Buffer | | Effect |
|---|---|---|---|
| 10 | [N] | [-] | |
| ⋮ | ⋮ | ⋮ | |
| → p | SET 10 | 0050 | C(X), i.e., N, → C(10). |
| p + 1 | X | X | |
| ⋮ | ⋮ | ⋮ | |
| X | N | N | |

and the contents of register X replace the contents of 10, C(X) → C(10). In this case X is the rightmost 10 bits, the address part, of register p + 1; the leftmost bit of C(p + 1) may have any value and, for the present, bit 10 must be 0.

In the second case, when i = 1, the LINC is directed to interpret C(p + 1) as the value which replaces C(α). Thus, below, C(p + 1) → C(5):

| Memory Address | Memory Buffer | | Effect |
|---|---|---|---|
| 5 | [N] | [-] | |
| ⋮ | ⋮ | ⋮ | |
| → p | SET i 5 | 0065 | C(p + 1), i.e., N, → C(5). |
| p + 1 | N | N | |

The following program scans $100_8$ memory registers looking for a value which matches C(ACC). It halts with the location of the matching register in the accumulator if a match is found, or with −0 in the accumulator if a match is not found. The numbers to be scanned are in registers 1000-1077.

| Memory Address | Memory Buffer | | Effect |
|---|---|---|---|
| 3 | [-100] | [7677] | -(number of registers to scan). |
| 4 | [X] | [0777] | Scanning address. |
| ⋮ | ⋮ | ⋮ | |
| Start → 400 | SET i 3 | 0063 | C(401), i.e., -100, → C(3). |
| 401 | -100 | 7677 | |
| 402 | SET i 4 | 0064 | C(403), i.e., 777, → C(4). |
| 403 | 777 | 0777 | |
| 404 | SAE i 4 | 1464 | Index X(4) and compare C(X) with C(ACC). |
| 405 | JMP 411 | 6411 | C(ACC) ≠ C(X), jump to 411. |
| 406 | CLR← | 0011 | C(ACC) = C(X), copy location of matching register into ACC and halt. |
| 407 | ADD 4 | 2004 | |
| 410 | HLT | 0000 | |
| 411 | XSK i 3 | 0223 | Index register 3 and test for X(3) = 1777. |
| 412 | JMP 404 | 6404 | X(3) ≠ 1777, return. |
| 413 | CLR← | 0011 | X(3) = 1777; all numbers have been scanned so -0 → C(ACC) and halt. |
| 414 | COM | 0017 | |
| 415 | HLT | 0000 | |

Example 11 Setting Initial Index Register Values

The two SET instructions are executed once every time the program is started at 400; initially registers 3 and 4 may contain any values since the program itself sets them to the correct values.

Suppose the programmer had wanted to SET two index registers to the same value, say -100. He could write either:

| Memory Address | Memory Buffer | | Effect |
|---|---|---|---|
| 11 | [-100] | [7677] | |
| 12 | [-100] | [7677] | |
| ⋮ | ⋮ | ⋮ | |
| → 20 | SET i 11 | 0071 | C(21), i.e., -100, → C(11). |
| 21 | -100 | 7677 | |
| 22 | SET 12 | 0052 | C(21), i.e., -100, → C(12). |
| 23 | 21 | 0021 | |

or:

| | | | |
|---|---|---|---|
| → 20 | SET i 11 | 0071 | C(21), i.e., -100, → C(11). |
| 21 | -100 | 7677 | |
| 22 | SET 12 | 0052 | (C11), i.e., -100, → C(12). |
| 23 | 11 | 0011 | |

The programmer could also, of course, have written SET i 12 in register 22 with -100 in register 23, but there are applications appropriate to each form.

## 2-7 INDEX CLASS INSTRUCTIONS II

### 2-7.1 Double Register Forms

The index class instructions have been thought of as addressing an index register $\beta$, $1 \leq \beta \leq 17$, which contains a memory address X to be used by the instruction. They have been presented as single register instructions (unlike SET). However, when an index class instruction is written with $\beta = 0$, it becomes a double register instruction like SET, whose operand address depends on i and p + 1. These two interpretations are shown for STA.

Case: $i = 0$, $\beta = 0$

| Memory Address | Memory Buffer | | Effect |
|---|---|---|---|
| 450 | STA | 1040 + 20(0) + 0 | C(ACC) → C(330). |
| 451 | 330 | 0330 | |

When i = 0, the LINC is directed to use C(p + 1), i.e., C(451) as the memory address at which to store C(ACC). The leftmost bit of C(p + 1) may have any value, and, for the present, bit 1 must be 0.

Case: i = 1, $\beta$ = 0

| Memory Address | Memory Buffer | | Effect |
|---|---|---|---|
| 450 | STA i | 1060 | C(ACC) → C(451) |
| 451 | [-] | [-] | |

When i = 1, the LINC is directed to use p + 1, i.e., 451, directly as the memory address, and the contents of the accumulator are stored in 451. Note that when $\beta$ = 0 in an index class instruction, it does not refer to memory register 0. In fact, when $\beta$ = 0, no reference is necessarily made to the index registers. As with SET, the computer automatically takes the next instruction from register p + 2.

Index class instructions may be thought of as having four alternative ways of addressing the memory, which depend on i and $\beta$, and which are summarized below:

| Index Class Address Variations | | | | |
|---|---|---|---|---|
| Case | i, $\beta$ | Example | Form | Comments |
| 1 | i = 0<br>$\beta \neq 0$ | LDA $\beta$ | Single Register | Register $\beta$ holds operand address. |
| 2 | i = 1<br>$\beta \neq 0$ | LDA i $\beta$ | Single Register | First, index register $\beta$ by 1. Then, register $\beta$ holds operand address. |
| 3 | i = 0<br>$\beta$ = 0 | LDA X | Double Register | Second register holds operand address. |
| 4 | i = 1<br>$\beta$ = 0 | LDA i N | Double Register | Second register holds operand. |

The next programming example scans memory registers 1350-1447, counting the number of instances in which register contents are found to exceed some threshold value, T. In other words if C(X) > T, X = 1350, 1351, ..., 1447, then C(CTR) + 1 → C(CTR), where CTR is a memory register used as a counter, initially set to 0. The count, N, is to appear in the accumulator upon program completion.

| Memory Address | Memory Buffer | | Effect |
|---|---|---|---|
| 14 | [X] | [-] | Address of register to be tested. |
| 15 | [-n] | [-] | -(number of registers to test). |
| Start → 30 | SET i 14 | 0074 | Set index register 14 to initial address minus 1. |
| 31 | 1347 | 1347 | |
| 32 | SET i 15 | 0075 | Set index register 15 to -100. |
| 33 | -100 | 7677 | |
| 34 | CLR | 0011 | Clear CTR; 0→C(51). |
| 35 | STC 51 | 4051 | |
| 36 | →LDA i | 1020 | C(37), i.e., -T,→C(ACC). |
| 37 | -T | -T | |
| 40 | ADA i 14 | 1134 | Index the address in register 14 and form C(X)-T in ACC. |
| 41 | BCL i | 1560 | Clear all but the sign bit in ACC; C(42) = the bit pattern for clearing. Then if C(X) > T, C(ACC) = 0000, but if C(X) < T, C(ACC) = 4000. |
| 42 | 3777 | 6777 | |
| 43 | SAE i | 1460 | Does C(ACC) = C(44)? If so, skip to 46. |
| 44 | 0000 | 0000 | |
| 45 | JMP 52 | 6052 | If not, C(X) ≤ T. Jump to 52. |
| 46 | LDA i← | 1020 | If so, C(X) > T; 1→C(ACC). |
| 47 | 1 | 0001 | |
| 50 | ADM i | 1160 | C(ACC) + C(51), i.e., N,→ C(51) and→C(ACC). |
| 51 | [N] | [-] | |
| 52 | →XSK i 15 | 0235 | Index register 15 and test for 7777. |
| 53 | JMP 36 | 6036 | C(15) ≠ 7777. Return to check next register. |
| 54 | HLT←— | 0000 | C(15) = 7777, therefore halt. C(CTR), i.e., C(51), left in ACC. |

Example 12 Scanning for Values Exceeding a Threshold

Note that since the SAE instruction in locations 43 and 44 is written as a double register instruction, the LINC skips to location 46 (not 45) when the skip condition is satisifed. The next instruction in sequence is, in this case, at location 45.

Note also that if a double register instruction is written following a skip instruction such as XSK, the LINC tries to interpret the second register as an instruction:

| Memory Address | Memory Buffer | Effect |
|---|---|---|
| ⋮ | ⋮ | |
| p | XSK i β | |
| p + 1 | LDA i | Go to p + 1 when X(β) ≠ 1777. |
| p + 2 | 3 | Go to p + 2 when X(β) = 1777. |
| ⋮ | ⋮ | |

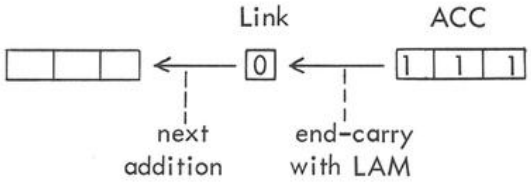
Since the XSK instruction sometimes directs the LINC to skip to p + 2, care must be taken to make sure that the LINC does not skip or jump to the second register of a double register instruction.

It is interesting to compare the above statement of the program made in rather detailed machine language with the following compact but entirely adequate restatement:

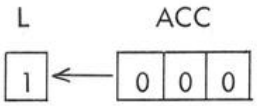
1. 0→C(CTR).
2. If C(X) > T then C(CTR) + 1→C(CTR), for X = 1350, 1351, ..., 1447.
3. C(CTR)→C(ACC).
4. HALT

## 2-7.2 Multiple Length Arithmetic

An index class instruction, LAM i β (link add to memory), code 1200 + 20i + β, makes arithmetic possible with numbers which are more than 12 bits long. Using LAM, one can work with 24-bit numbers for example, using 2 memory registers to hold right and left halves. It should be remembered that addition with ADD, ADA, or ADM always involves end-around carry. With LAM, however, a carry from bit 0 of the accumulator during addition is saved in the link bit; it is not added to bit 11 of the accumulator. This carry, then, could be added to the low-order bit of another number, providing a carry linkage between right and left halves of a 24-bit number. For simplicity, the illustration uses 3-bit registers; the principles are the same for 12 bits:



If, for example, the number in this 3-bit accumulator is 7 (all 1's) and C(L) = 0, and 1 is added with LAM, the link bit and accumulator will then look like:




Furthermore, LAM is an add-to-memory instruction, so that the memory register to which the LAM instruction refers will now contain 0 (as does the accumulator).

In addition to saving the carry in the link bit the LAM instruction also adds the contents of the link bit to the low order bit of the accumulator. That is, if, when the LAM instruction is executed C(L) = 1, then 1 is added to C(ACC). Using the result pictured above, add 2, where 2 is the contents of some memory register M:

| | L | ACC | M |
|---|---|---|---|
| Given: | 1 | 000 | 010 |

Using LAM, the LINC is directed first to add C(L) to C(ACC), giving:

| L | ACC | M |
|---|---|---|
| 0 | 001 | 010 |

There is no end-carry from this operation, so the link bit is cleared. The LINC then adds C(ACC) to C(M), giving:

| L | ACC | M |
|---|---|---|
| 0 | 011 | 011 |

which replaces both C(ACC) and C(M). Again there is no end-carry so the link bit is left unchanged.

The operation of LAM may be summarized:

1. C(L) + C(ACC) → C(ACC).
2. End-carry → C(L). If no end-carry, 0 → C(L).
3. C(ACC) + C(M) → C(ACC), and → C(M).
4. End-carry → C(L). If no end-carry, the link bit is left unchanged.

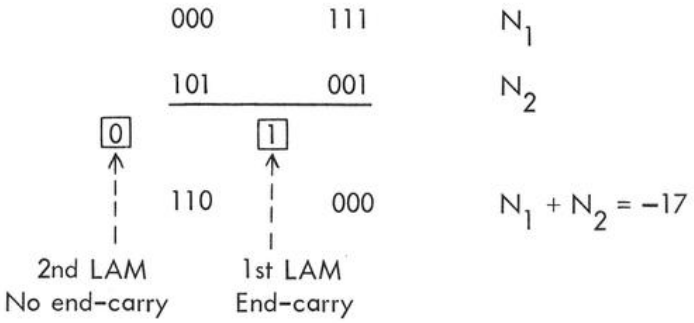
As an example of double length arithmetic, postulate 2 numbers, $N_1$ and $N_2$, each 6 bits long, which occupy a total of four 3-bit memory registers, $M_1$ through $M_4$:

| $M_2$ | $M_1$ | |
|---|---|---|
| 000 | 111 | $N_1 = +7$ |

| $M_4$ | $M_3$ | |
|---|---|---|
| 101 | 001 | $N_2 = -26$ |

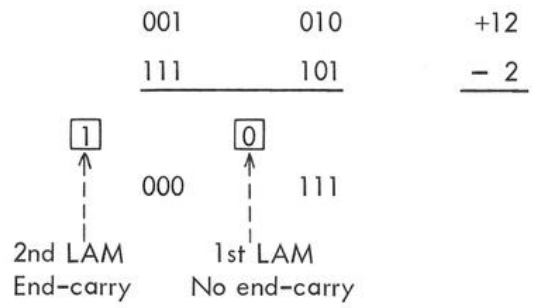
The sum (octal) of +7 and −26 is −17. Using the LAM instruction to get this:

1. Clear the link bit.
2. Add C($M_1$) to C($M_3$) with LAM, saving any carry in the link bit. This sums the right halves of $N_1$ and $N_2$.
3. Add C($M_2$) to C($M_4$) with LAM, which also adds in any carry from step 2. This sums the left halves of $N_1$ and $N_2$. Any new carry again replaces C(L).




Note that only the first LAM produced an end-carry.

To complete the illustration, consider a case in which the final carry appears in the link bit, as in the addition of +12 and −2.




whose sum, in 1's complement notation is 001 000, or $+10_8$, but which with LAM results in +7 and an end-carry in the link bit. Since 1's complement representation depends on end-around carry, some extra programming must be done to restore the result to a true 1's complement number. This is, of course, the equivalent of adding 1 to the 2-register result. Assuming that the result is in $M_1$ and $M_2$,

| L | $M_2$ | $M_1$ |
|---|---|---|
| 1 | 000 | 111 |

again use the LAM instruction. First clear the accumulator without clearing the link bit (this can be done with an STC instruction). Then execute LAM with C($M_1$) which gives

| L | ACC | $M_1$ |
|---|---|---|
| 1 | 000 | 000 |

producing a new end-carry in the link bit. Again clear the accumulator (but not the link bit) and execute LAM with C($M_2$) which gives

| L | ACC | $M_2$ |
|---|---|---|
| 0 | 001 | 001 |

The result in $M_2$ and $M_1$ now looks like:

| $M_2$ | $M_1$ | |
|---|---|---|
| 001 | 000 | = +10 (octal) |

It should be clear to the reader that adding in a final end-carry as an end-around carry cannot itself give rise to a new final end-carry.

The following program illustrates the technique of double length arithmetic with tables of numbers; similar techniques would be used for other multiples of 12. Assume that $100_8$ 24-bit numbers, $N_0, N_1, \ldots, N_{77}$, are to be added term by term to $100_8$ numbers, $R_0, R_1, \ldots, R_{77}$, so that $N_0 + R_0 = S_0$, $N_1 + R_1 = S_1$, etc. All numbers occupy 2 registers: the left halves of $N_0, N_1, \ldots, N_{77}$ are in registers 100-177, the right halves in 200-277. The left halves of $R_0, R_1, \ldots, R_{77}$ are in 1000-1077, the right halves in 1100-1177. The left halves of the sums, $S_0, S_1, \ldots, S_{77}$, replace the contents of 1000-1077, the right halves replace the contents of 1100-1177.

| Memory Address | Memory Buffer | | Effect |
|---|---|---|---|
| 10 | $[X_1]$ | [-] | |
| 11 | $[X_2]$ | [-] | |
| 12 | $[X_3]$ | [-] | |
| 13 | $[X_4]$ | [-] | |
| 14 | [-n] | [-] | |
| ⋮ | ⋮ | ⋮ | |
| 377 | [-] | [-] | |
| Start → 400 | SET i 10 | 0070 | Set index registers to initial addresses minus 1 for the 4 tables. |
| 401 | 77 | 0077 | |
| 402 | SET i 11 | 0071 | |
| 403 | 177 | 0177 | |
| 404 | SET i 12 | 0072 | |
| 405 | 777 | 0777 | |
| 406 | SET i 13 | 0073 | |
| 407 | 1077 | 1077 | |
| 410 | SET i 14 | 0074 | Set index register 14 as a counter for 100 loop repetitions. |
| 411 | -100 | 7677 | |
| 412 | → CLR | 0011 | 0 → C(ACC); 0 → C(L). |
| 413 | LDA i 11 | 1031 | Right half of $N_i$ → C(ACC). |
| 414 | LAM i 13 | 1233 | Right half of $N_i$ + right half of $R_i$ → C(ACC), and → right half of $R_i$. End-carry → C(L). |
| 415 | LDA i 10 | 1030 | Left half of $N_i$ → C(ACC). |
| 416 | LAM i 12 | 1232 | C(L) + C(ACC) + left half of $R_i$ → C(ACC), and → left half of $R_i$. End-carry → C(L). |
| 417 | STC 377 | 4377 | Clear accumulator by storing in 377. Do not clear link bit. |
| 420 | LAM 13 | 1213 | C(L) + right half of $S_i$ → C(ACC), and right half of $S_i$. End-carry → C(L). |
| 421 | STC 377 | 4377 | Clear accumulator. |
| 422 | LAM 12 | 1212 | C(L) + left half os $S_i$ → C(ACC), and left half of $S_i$. |
| 423 | XSK i 14 | 0234 | Index 14 and test for 7777. |
| 424 | JMP 412 | 6412 | C(14) ≠ 7777, return to form next sum. |
| 425 | HTL←--- | 0000 | C(14) = 7777, so halt. |

Example 13 Summing Sets of Double Length Numbers Term by Term

MUL

The instructions in locations 412-416 produce an initial 24-bit sum leaving any final carry in the link bit. The instructions in locations 417-422 then complete the sum by adding in the final end-carry. The link bit always contains 0 after the computer executes the last LAM in location 422. Register 377 is used simply as a "garbage"register so that the accumulator can be cleared without clearing the link bit.

## 2-7.3 Multiplication

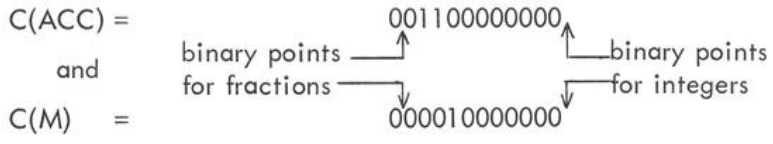
Another index class instruction which needs special explanation is MUL i $\beta$ (multiply), code $1240 + 20i + \beta$. This instruction directs the LINC to multiply C(ACC) by the contents of the specified memory register, and to leave the result in the accumulator. The multiplier and multiplicand are treated as signed 11-bit 1's complement numbers, and the sign of the product is left in both the accumulator (bit 0) and the link bit.

The LINC may be directed to treat both numbers either as integers or fractions; it may not, however, be directed to mix a fraction with an integer. The leftmost bit (bit 0) of register $\beta$ is used to specify the form of the numbers.

When bit 0 of register $\beta$ contains 0, the numbers are treated as integers; that is, the binary points are assumed to be to the right of bit 11 of the accumulator and the specified memory register. Given C(ACC) = −10, C($\beta$) = 400 (bit 0 of register $\beta$ = 0), and C(400) = +2, the instruction MUL $\beta$ leaves −20 in the accumulator, and 1 in the link bit. Overflow is, of course, possible when the product exceeds ±3777. Multiplying +3777 by +2, for example, produces +3776 in the accumulator; note that the sign of the product is correct, and that the overflow effectively occured from bit 1, not from bit 0.

When bit 0 of register $\beta$ contains 1, the LINC treats the numbers as fractions; that is, the binary point is assumed to be to the right of the sign bit (between bit 0 and bit 1) of the accumulator and the specified memory register. Given C(ACC) = +.2, C($\beta$) = 5120 (bit 0 of register $\beta$ = 1), and C(1120) = +.32, execution of MUL $\beta$ leaves +.064 in the accumulator and 0 in the link bit.

When the LINC multiplies two 11-bit signed numbers, a 22-bit product is formed. For integers the rightmost, or least significant, 11 bits of this product are left with the proper sign in the accumulator, and for fractions the most significant 11 bits of the product are left with the proper sign in the accumulator. If, for example,




then C(ACC) can be thought of as either $+.3_8$ or $+1400_8$, and C(M) can be thought of as either $+.04_8$ or $+200_8$. The 22-bit product of these numbers looks like:

$$\underbrace{.000\ 001\ 100\ 0}_{.014}\quad\underbrace{00\ 000\ 000\ 000.}_{0.}$$

and if bit 0 of register β contains 1, the most significant 11 bits with the proper sign are left in the accumulator:

$$C(ACC) = 0.\underbrace{000}\ \underbrace{001}\ \underbrace{100}\ 00$$
$$(+.3)\times(+.04) = +.\ 0\quad 1\quad 4$$

Had bit 0 of register β contained 0, the accumulator would be left with +0 as the result of multiplying (1400)x(200). It is the programmer's responsibility to avoid integer overflow by programming checks on his data and/or by scaling the values to a workable size.

Use of bit 0 of register β is new to the concept of index registers and should be noted in connection with the four memory addressing alternatives which index class instructions employ. When $\beta \neq 0$ then bit 0 of C(β), that is, bit 0 of the register which contains the memory address, is used. The same is true when i = 0 and β = 0, as in:

| Memory Address | Memory Buffer | |
|---|---|---|
| p | MUL | 1240 |
| p + 1 | h, X | 4000h + X |

That is, bit 0 of C(p + 1), the register containing the memory address, is used. This bit is sometimes called the h-bit, whether in an index register or in register p + 1. When, however, i = 1 and β = 0, it will be recalled that p + 1 is itself the memory address:

| Memory Address | Memory Buffer | |
|---|---|---|
| p | MUL i | 1240 |
| p + 1 | N | N |

There is no memory register which actually contains the memory address, and therefore there is no h-bit. The computer always assumes in this case that h = 0, and the operands are treated as integers.

In the following program, registers 1200-1377 contain a table of fractions whose values are in the range ±.0176, that is, whose most significant five bits after the sign (bits 1-5) duplicate the sign. Each number is to be multiplied by a constant, −.62, and the products stored at locations 1000-1177. To retain significance, the values are first shifted left 5 places.

| Memory Address | Memory Buffer | | Effect |
|---|---|---|---|
| 6 | $[X_1]$ | [-] | |
| 7 | $[X_2]$ | [-] | |
| 10 | [-n] | [-] | |
| ⋮ | ⋮ | ⋮ | |
| Start → 500 | SET i 6 | 0066 | Initial address minus 1 of table of fractions → C(6). |
| 501 | 1177 | 1177 | |
| 502 | SET i 7 | 0067 | Initial address minus 1 for STA instruction → C(7). |
| 503 | 777 | 0777 | |
| 504 | SET i 10 | 0070 | -n → C(10). |
| 505 | -200 | 7577 | |
| 506 | LDA i 6 | 1026 | Fraction → C(ACC). |
| 507 | ROL 5 | 0245 | $C(ACC) \cdot 2^5 \rightarrow C(ACC)$. |
| 510 | MUL | 1240 | Multiply, as fractions, C(ACC) by C(516). |
| 511 | 4000+516 | 4516 | |
| 512 | STA i 7 | 1067 | Store product. |
| 513 | XSK i 10 | 0230 | |
| 514 | JMP 506 | 6506 | If not finished, return. |
| 515 | HLT | 0000 | If finished, halt. |
| 516 | -.62 | 4677 | |

Example 14 Multiplying a Set of Fractions by a Constant

The ROL instruction at location 507 rotates 0's or 1's, depending on the sign, into the low-order 5 bits of the accumulator. Since this amounts to a scale left operation, it introduces no new information which might influence the product. The reader should also note that the original values remain unchanged at locations 1200-1377.

Another example demonstrates the technique of saving both halves of the product. Fifty (octal) numbers, stored at locations 1000-1047, are to be multiplied by a constant, +1633. The left halves of the products (the most significant halves) are to be saved at locations 1100-1147; the right halves (the least significant halves) at locations 1200-1247.

| Memory Address | Memory Buffer | | Effect |
|---|---|---|---|
| 3 | $[X_1]$ | [1077] | Addresses of products. |
| 4 | $[X_2]$ | [1177] | |
| 5 | $[4000+X_3]$ | [4777] | Addresses of multiplier as fraction and integer. |
| 6 | $[X_3]$ | [0777] | |
| 7 | [−n] | [7727] | Counter. |
| ⋮ | ⋮ | ⋮ | |
| → 1400 | SET i 3 | 0073 | Set addresses for storing products. |
| 1401 | 1077 | 1077 | |
| 1402 | SET i 4 | 0074 | |
| 1403 | 1177 | 1177 | |
| 1404 | SET i 5 | 0075 | Set 5 to address multiplier as fraction. |
| 1405 | 4000+777 | 4777 | |
| 1406 | SET i 6 | 0076 | Set 6 to address multiplier as integer. |
| 1407 | 777 | 0777 | |
| 1410 | SET i 7 | 0077 | |
| 1411 | −50 | 7727 | |
| 1412 | →LDA i | 1020 | Form left half of product$_i$ in accumulator. |
| 1413 | 1633 | 1633 | |
| 1414 | MUL i 5 | 1265 | |
| 1415 | SCR i 1 | 0361 | C(bit 11 of ACC) → C(L). |
| 1416 | STA i 3 | 1063 | Store left half of product$_i$. |
| 1417 | STC 1434 | 5434 | 0 → C(ACC). |
| 1420 | ROR i 1 | 0321 | C(L) → C(bit 0 of ACC). |
| 1421 | STC 1427 | 5427 | 4000 or 0000 → C(1427). |
| 1422 | ADD 1413 | 3413 | Form right half of product$_i$ in accumulator. |
| 1423 | MUL i 6 | 1266 | |
| 1423 | MUL i 6 | 1266 | |
| 1424 | BCL i | 1560 | Clear bit 0 of right half. |
| 1425 | 4000 | 4000 | |
| 1426 | BSE i | 1620 | C(bit 11 of left half) → C(bit 0 of right half). |
| 1427 | [-] | [-] | |
| 1430 | STA i 4 | 1064 | Store right half of product$_i$. |
| 1431 | XSK i 7 | 0227 | Return if not finished. |
| 1432 | JMP 1412 | 7412 | |
| 1433 | HLT← | 0000 | |
| 1434 | [-] | [-] | |

Example 15 Multiplication Retaining 22-Bit Products

ZTA

The instructions at locations 1415, 1420-1421, and 1424-1427 have the effect of making the two halves of the product contiguous; the sign bit value of the right half is replaced by the low-order bit value of the left half, so that the product may be subsequently treated as a true double length number.

Through the use of another instruction, ZTA, it is possible to do a double precision multiplication using only one MUL instruction. When the LINC performs a multiplication, it uses three basic registers: the accumulator for the multiplicand, the memory buffer register for the multiplier, and the Z register for partial containment of the initial 22-bit plus sign answer. The LINC then decides if the multiplication was fractional or integer and puts into the accumulator the correct half of the answer properly signed. In a fractional multiply, the most significant bits of the product are found in the accumulator; however, the low-order protion of the product is not lost but is still in the Z register as an unsigned number. By executing a Z to A instruction, ZTA (MSC 005), code 0005, the accumulator is cleared, and the contents of the Z register are copied into bits 1-11 of the accumulator. Bit 0 is always 0 and the number is unsigned. However, the link contains the sign of the product so that, if necessary, the low-order portion of the product may be complemented.

Since bit 0 of the low-order portion does not contain a significant bit after a ZTA instruction (unless the product was 3777 or less), it is useful to transfer bit 11 of the most significant portion of the product into bit 0 of the low-order portion. The following example multiplies the number in the LEFT SWITCHES by the number in the RIGHT SWITCHES and stores the double precision product in memory into two consecutive locations.

| Memory Address | Memory Buffer | | Effect |
|---|---|---|---|
| 100 | $[X_1]$ | [C(R.S.)] | Contents of RIGHT SWITCHES |
| 101 | $[X_2]$ | [H.O.P.] | High order product |
| 102 | $[X_3]$ | [L.O.P.] | Low order product |
| ⋮ | ⋮ | ⋮ | |
| Start → 400 | RSW | 0516 | Read RIGHT SWITCHES into A |
| 401 | STC 100 | 4100 | Store into location 100 |
| 402 | LSW | 0517 | Read LEFT SWITCHES into A |
| 403 | MUL | 1240 | Multiply (fractional) C(C(p + 1)) by C(A) |
| 404 | [4000+0100] | 4100 | |
| 405 | STC 101 | 4101 | Store high order product into location 101 |
| 406 | ZTA | 0005 | C(Z) → C(A) |
| 407 | LZE | 0452 | Was product positive? |
| 410 | COM | 0017 | No, complement A |
| 411 | STC 102 | 4102 | Store low order product in location 102 |
| 412 | ADD 101 | 2101 | Get back high order product |
| 413 | ROR i 1 | 0321 | Rotate bit 11 into link, link (sign bit) into bit 0 |
| 414 | STC 101 | 4101 | Store into 101 |
| 415 | ADD 102 | 2102 | Get low order product |
| 416 | ROL 1 | 0241 | Rotate bit 0 into bit 11 |
| 417 | ROR i 1 | 0321 | Rotate link into bit 0, bit 11 into link |
| 420 | STC 102 | 4102 | Store into 102 |
| 421 | HLT | 0000 | |

Example 16 Multiplication for 22 Bit-Product Using ZTA

There are two remaining index class instructions, SRO i β (skip rotate), and DSC i β (display character), which is discussed later in connection with programming the oscilloscope display.

## 2-8 HALF-WORD CLASS INSTRUCTIONS

The LINC has 3 instructions which deal with 6-bit numbers or half-words (word is another term for contents of a register). These instructions use the index registers and have the same four addressing variations as the index class, but specify in addition either the left half or right half of the contents of memory register X as the operand. Think of LH(X) as meaning the contents of the left 6 bits of register X, and RH(X), meaning the contents of the right 6 bits. Then it is possible to think of C(X) = LH|RH, or C(X) = 100 LH+RH.

LDH, STH

Half-word instructions always use the right half of the accumulator. The load half instruction, LDH i β, code 1300 + 20i + β, clears the accumulator and copies the specified half-word into the right half of the accumulator; which half of C(X) to use is specified by bit 0, the h-bit, of register β.

When h = 0, LH(X)→RH(ACC). When h = 1, RH(X) → RH(ACC):

| Memory Address | Memory Buffer | | Effect |
|---|---|---|---|
| β | h, X | 4000h+X | h = 1. |
| ⋮ | ⋮ | ⋮ | |
| p | LDH β | 1300+β | RH(X)→ RH(ACC) and 0→ LH(ACC). |
| ⋮ | ⋮ | ⋮ | |
| X | LH\|RH | 100LH+RH | C(X) unchanged. |

The same interpretation of the h-bit applies when i = 0 and β = 0, i.e., when the instruction occupies two registers:

| Memory Address | Memory Buffer | | Effect |
|---|---|---|---|
| 40 | LDH | 1300 | Since h = 1, RH(500), i.e., 76, →RH(ACC). 0→ LH(ACC). |
| 41 | 1,500 | 4500 | |
| ⋮ | ⋮ | ⋮ | |
| 500 | 32\|76 | 3276 | |

If register 41 contained 500, i.e., h = 0, then LH(500), or 32, would replace RH(ACC).

The store half instruction, STH i β, code 1340 + 20i + β, stores the right half of C(ACC) in the specified half of memory register X. C(ACC) and the other half of memory register X are unaffected. To illustrate the case of i = 1 and β = 0, write:

| Memory Address | Memory Buffer | | Effect |
|---|---|---|---|
| 1000 | STH i | 1360 | RH(ACC) → LH(1001) |
| 1001 | 6015 | 6015 | |

This case, it will be remembered, uses p + 1 itself as the memory address. Since there is no h-bit, the computer assumes that h = 0, and therefore the left half of C(1001) is affected. If, for example, C(ACC) = 5017, 17 replaces LH(1001), and the contents of register 1001 become 1715.

SHD i β (skip if half differs), code 1400 + 20i + β, causes the LINC to skip one memory register in the program sequence when the right half of the accumulator does not match the specified half of memory register X. When it does match, the computer goes to the next memory register in sequence for the next instruction. Neither C(ACC) nor C(X) is affected by the instruction. If C(ACC) = 4371, and the programmer writes:

| Memory Address | Memory Buffer | | Effect |
|---|---|---|---|
| 376 | 7152 | 7152 | |
| → 377 | SHD | 1400 | Skip to 402 if RH(376) ≠ RH(ACC). |
| 400 | 4376 | 4376 | |
| 401 | - | - | |
| 402 | - ← | - | |

The computer skips because RH(376), i.e., 52, ≠ RH(ACC), or 71. Had he written 376 in location 400, that is, h = 0, RH(ACC) would equal LH(376) and the computer would not skip.

When β ≠ 0, and when i = 1, the half-word class instructions cause the LINC to index the contents of memory register β, but in a more complex way than that used by the index class instructions. In order to have half-word indexing refer to consecutive half-words, the computer adds 4000 to C(β) with end-around carry. This has the effect of complementing h(β) every time register β is indexed, and stepping X(β) every other time. Suppose, for example, that the instruction is LDH i 3, and that register 3 initially contains 4377, that is, it points to the right half of register 377. The computer first adds 4000 to C(3):

```
 4377   Original C(3) = 1,377
 4000   Index H(3)
 0377
    1   End-around carry
 0400   New C(3) = 0,400
```

which leaves h = 0 and X = 400; C(3) now points to the left half of register 400. The computer therefore loads the accumulator from LH(400). Repeating the instruction, C(3) is indexed to 4400 and the accumulator is loaded from RH(400). Continuing, register 3 would contain the following succession of values or half-word references:

4400 : RH(400)
0401 : LH(401)
4401 : RH(401)
0402 : LH(402)
4402 : RH(402)
0403 : LH(403)
etc. etc.

Since half-word indexing occurs before the contents of register β are used to address the memory, the memory address, when i = 1, can be described as

$\overline{h}$, X+h

where $\overline{h}$ represents the indexed value of h, and X+h represents the indexed value of X. The succession of values which appear in register β can be written:

$\underline{\overline{h}, X+h}$

1, X+0
0, X+1
1, X+1
0, X+2
1, X+2
etc.

The four address variations for half-word class instructions are summarized in the following table.

Half-Word Class Address Variations

| Case | i, β | Example | Form | Comments |
|---|---|---|---|---|
| 1 | i = 0<br>β ≠ 0 | LDH β | Single Register | Register β holds half-word operand address. |
| 2 | i = 1<br>β ≠ 0 | LDH i β | Single Register | First, index register β by 4000 with end-around carry.<br>Then, register β holds half-word operand address. |
| 3 | i = 0<br>β = 0 | LDH h, X | Double Register | Second register holds half-word operand address. |
| 4 | i = 1<br>β = 0 | LDH i<br>LH\|RH | Double Register | Left half of second register holds half-word operand. |

For h = 0, the operand is held in the left half of the specified memory register. For h = 1, the operand is held in the right half of the specified memory register.

KBD

## 2-9 THE KEYBOARD INSTRUCTION

Before continuing with half-word class programming examples, the keyboard instruction, KBD i, code 515 + 20i, is introduced. The LINC-8 uses the ASR 33 as a keyboard for the LINC section. Each key has an eight level code which is converted to a 6-bit code by the interpretive program in the PDP-8 (PROGOFOP) (see chart II). When a key is struck the 6-bit code for that character is transferred into the right half of the LINC's accumulator by the KBD i. The i-bit is used here in a special way to synchronize the keyboard with the computer. When i = 1, if a key has not been struck, the computer will wait for a key to be struck before trying to read a key code into the LINC accumulator. When i = 0, the computer does not wait, and the programmer must insure that a key has been struck before the computer tries to execute the KBD instruction; otherwise a 0 will be transferred to the LINC accumulator. Use of the i-bit to cause the computer to pause is unique to a class of instructions known as the operate instructions, of which KBD is a member. As a class they are used to control or operate external equipment.

The following program reads in key code numbers as keys are struck on the keyboard, and stores them at consecutive half-word locations, LH(100), RH(100), LH(101), ..., until the Z, code number $55_8$, is struck, which stops the program.

| Memory Address | Memory Buffer | | Effect |
|---|---|---|---|
| 7 | [h,X] | [-] | Half-word index register. |
| → 20 | SET i 7 | 0677 | Set index register 7 to one half-word location less than initial location. |
| 21 | 1,077 | 4077 | |
| 22 | →KBD i | 0535 | Read code number of struck key into RH(ACC), and release the key. |
| 23 | SHD i | 1420 | Skip to location 26 if code number ≠ 55. |
| 24 | 5500 | 5500 | |
| 25 | HLT | 0000 | Code = 55, so halt. |
| 26 | STH i 7← | 1367 | Half-word index register 7, store code number, and return to read next key. |
| 27 | JMP 22 | 6022 | |

Example 17 Filling Half-Word Table from the Keyboard

Another example reads key code numbers and stores at consecutive half-word locations only those code numbers which represent the letters A-Z, codes $24_8$-$55_8$. Other key codes are discarded, and the program stops when $100_8$ letters have been stored.

DIS

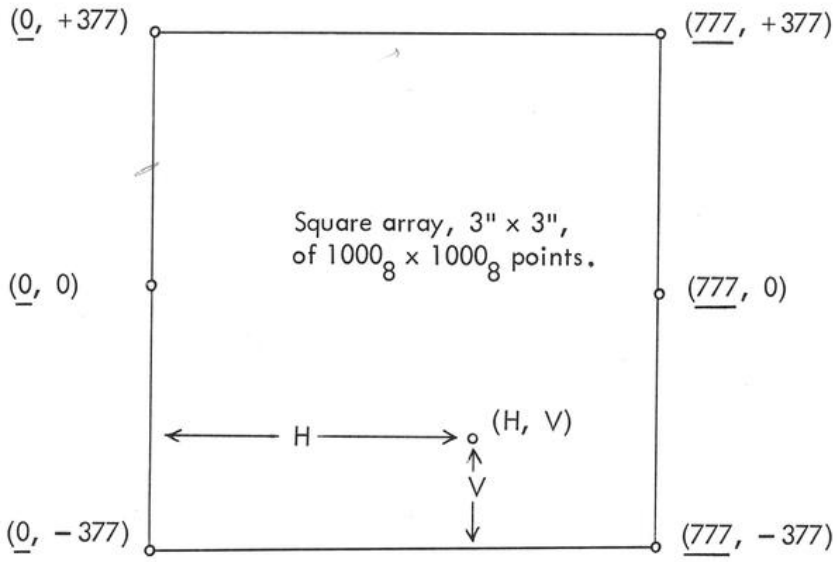
| Memory Address | Memory Buffer | | Effect |
|---|---|---|---|
| 5 | [h,X] | [-] | |
| 6 | [-n] | [-] | |
| ⋮ | ⋮ | ⋮ | |
| → 100 | SET i 6 | 0066 | Set 6 to count 100 times. |
| 101 | -100 | 7677 | |
| 102 | SET i 5 | 0065 | Set 5 for storing letters beginning at LH(1000). |
| 103 | 1,777 | 4777 | |
| 104 | →KBD i | 0535 | Read keyboard. |
| 105 | STA i | 1060 | C(ACC) →C(106); store key code in 106. |
| 106 | [-] | [-] | |
| 107 | ADA i | 1120 | C(ACC) − 23 →C(ACC). |
| 110 | −23 | 7754 | |
| 111 | BCL i | 1560 | Clear all but the sign bit in ACC. |
| 112 | 3777 | 3777 | |
| 113 | AZE | 0450 | If C(ACC) = 0, skip to location 115. |
| 114 | JMP 104 | 6104 | C(ACC) ≠ 0, so key code was less than 24. Return to read next key. |
| 115 | LDH← | 1300 | Key code 23 represents a letter. Therefore RH(106) RH(ACC). |
| 116 | 1106 | 4106 | |
| 117 | STH i 5 | 1365 | Half-word index register 5 and store code for letter. |
| 120 | XSK i 6 | 0226 | Index register 6 and return if 100 letters have not been struck. |
| 121 | JMP 104 | 6104 | |
| 122 | HLT← | 0000 | |

Example 18 Selective Filling of Half-Word Table from the Keyboard

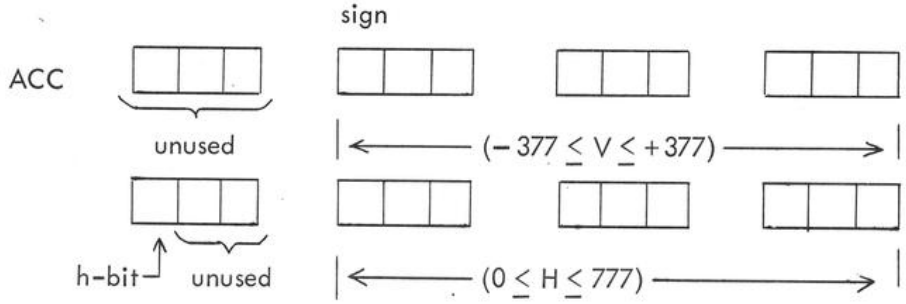
## 2-10 THE LINC SCOPES AND THE DISPLAY INSTRUCTIONS

The LINC has a cathode ray tube display device called a display scope, which is capable of presenting a square array of $512_{10}$ by $512_{10}$ spots ($1000_8$ by $1000_8$). A special instruction, DIS i $\alpha$ (display), code 140 + 20i + $\alpha$, momentarily produces a bright spot at one point in this array. The horizontal (H) and vertical (V) coordinates are specified in the accumulator and in $\alpha$. The vertical coordinate,

$-377_8 \leq V \leq +377_8$, is held in the accumulator during a DIS i $\alpha$ instruction; the horizontal coordinate, $0 \leq H \leq 777_8$, is held in register $\alpha$, $0 \leq \alpha \leq 17$. The spot in the lower left corner of the array has the coordinates (0, −377):




The coordinates are held in the rightmost 9 bits of register $\alpha$ and the accumulator,




so that if C(ACC) = 641, i.e., −136, and C(5) = 430, DIS 5 causes a spot to be intensified at (430, −136) on the scope.

Both channels are positioned at the same time. The production of a bright spot on either channel depends upon the state of the leftmost bit (the h-bit) of register $\alpha$ and an external channel selector located on the face of the display scope. If h = 0, then the spot is produced via display channel 0; if h = 1, then the spot is produced via display channel 1. The scope may be manually set to intensify channel 0, channel 1, or both.

The i-bit in DIS i $\alpha$ is used in the usual way to specify whether to index the right 10 bits of register $\alpha$ before brightening the spot. This indexing, of course, also increases the horizontal coordinate by one. To illustrate, the following program will display a continuous horizontal line through the middle (V=0) of the scope via display channel 0:

| Memory Address | Memory Buffer | | Effect |
|---|---|---|---|
| 5 | [0, H] | [-] | Horizontal coordinate and channel selection. |
| → 20 | SET i 5 | 0065 | Set 5 to channel 0 and horizontal coordinate = 0. |
| 21 | 0 | 0000 | |
| 22 | CLR | 0011 | Vertical coordinate = 0→C(ACC). |
| 23 | →DIS i 5 | 0165 | Index H (actually index entire rightmost 10 bits) and display. Repeat endlessly. |
| 24 | JMP 23 | 6023 | |

Example 19 Horizontal Line Scope Display

Another example displays as a curve the values found in a set of consecutive registers, 1400-1777. The vertical coordinates are the most significant 9 bits of each value. Since these are only $400_8$ points to display, the curve will be positioned in the middle of the scope. Channel 1 is used.

| Memory Address | Memory Buffer | | Effect |
|---|---|---|---|
| 10 | [X] | [-] | Address of vertical coordinates. |
| 11 | [1, H] | [4000+H] | Channel select and horizontal coordinate. |
| →300 | →SET i 10 | 0070 | Set 10 to beginning address minus 1. |
| 301 | 1377 | 1377 | |
| 302 | SET i 11 | 0071 | Set 11 to select channel 1 and to begin curve at H = 200. |
| 303 | 1,177 | 4177 | |
| 304 | →LDA i 10 | 1030 | Load ACC with value and scale right 3 places to position it as vertical coordinate. |
| 305 | SCR 3 | 0343 | |
| 306 | DIS i 11 | 0171 | Index the H coordinate and display. |
| 307 | XSK 10 | 0210 | Check to see if X(10) = 1777. |
| 310 | JMP 304 | 6304 | If $400_8$ points have not been displayed, return to get next point. |
| 311 | JMP 300 | 6300 | If X(10) = 1777, return to repeat entire display. |

Example 20 Curve Display of a Table of Numbers

## 2-10.1 Character Display

Display scopes are frequently used to display characters, for example keyboard characters, as well as data curves. Character display is somewhat more complicated since the point pattern must be carefully worked out in conjunction with the vertical and horizontal coordinates for each point. For example, to display the letter A, the array on the scope might look like:

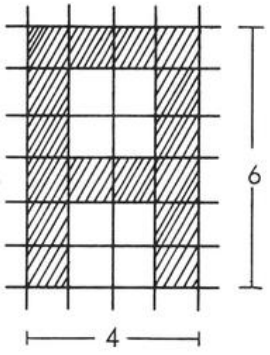



Figure a

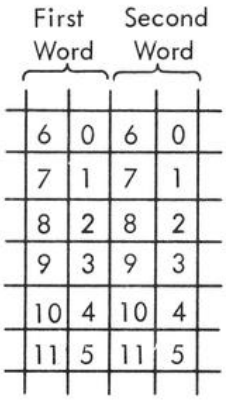
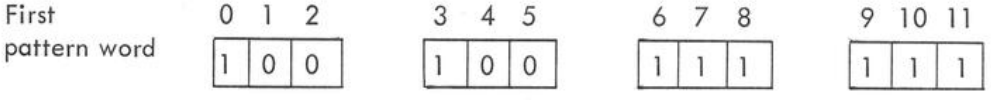



Figure b

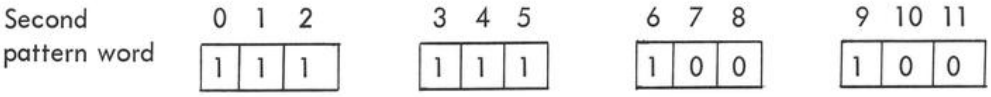
where the shaded areas of figure a represent points which are intensified, and the white areas points not intensified; the total area represented is 6 vertical positions by 4 horizontal positions. If, for example, the lower left point has the coordinates (400, 0), then the upper right point has the coordinates (403, 5).

The programmer could, of course, store the H and V coordinates for every intensified point of the character in a table in the memory, but the letter A alone, for instance, would require $32_{10}$ registers to hold both coordinates for all the points which are intensified. Instead he arbitrarily decides upon a scope format, say 4 x 6, and makes up a pattern word in which 1's represent points to be intensified and 0's points which are not intensified. To specify a 4 x 6 pattern of 24 bits requires 2 memory registers. For efficiency of programming, the points are displayed in the order shown numerically in figure b, i.e., from lower left to upper right, column by column. Examining bit 11 of the pattern word first, bit 10 next, bit 9, etc., the pattern word for the left half of the letter A (the left two columns) looks like:




The pattern word for the right half of the letter looks like:

SRO




An index class instruction, SRO i β (skip rotate), code 1500 + 20i + β, facilitates character display with the kinds of pattern words described above. SRO i β directs the LINC to skip the next register in the instruction sequence when bit 11 of the specified memory register contains 0. If bit 11 contains 1, the computer does not skip. In either case, however, after examining bit 11, the contents of the specified memory register are rotated 1 place to the right. Therefore, repeating the SRO instruction (with reference to the same memory register) has the effect of examining first bit 11, then bit 10, bit 9, etc. Executing the SRO instruction twelve times, of course, restores the memory word to its original configuration.

The following example repeatedly displays the letter A in the middle of the scope, using register 7 to hold the address of the first pattern word and register 6 to hold the H coordinate. Since 4 x 6 contiguous points on the scope array define an area too small to be readable, a delta of 4 is used to space the points, so that if the first point is intensified at coordinates (370, 0) the second point will be at (370, 4), the 7th point at (374, 0), etc. (This produces characters approximately 0.5 cm. high.)

| Memory Address | Memory Buffer | | Effect |
|---|---|---|---|
| 6 | [0, H] | [-] | Channel selection and H coordinate. |
| 7 | [X] | [-] | Address of pattern word. |
| → 60 | → SET i 6 | 0066 | Set H coordinate = 370 for lower left point. Select channel 0. |
| 61 | 0, 370 | 0370 | |
| 62 | SET i 7 | 0067 | Set 7 to address of first half of pattern. |
| 63 | 110 | 0110 | |
| 64 | → LDA i | 1020 | Initial V coordinate = −10→ C(ACC). |
| 65 | −10 | 7767 | |
| 66 | → SRO 7 | 1507 | Skip to location 70 if bit 11 of pattern word is 0. Rotate the pattern word 1 place to right. |
| 67 | DIS 6 | 0146 | If bit 11 of pattern word was 1, display one point. |
| 70 | ADD 75 ↵ | 2075 | Add 4 to V coordinate in ACC. |
| 71 | SRO i | 1520 | Skip to location 74 when 6 bits of pattern word have been examined. Rotate C(72) 1 place to right. |
| 72 | 3737 | 3737 | |
| 73 | JMP 66 | 6066 | Return to examine next bit of pattern word when bit 0 of C(72) = 1. |
| 74 | LDA i ↵ | 1020 | When bit 11 of C(72) = 0, 6 points have been examined. Increase H coordinate by 4 to do next column. |
| 75 | 4 | 0004 | |
| 76 | ADM | 1140 | |
| 77 | 6 | 0006 | |
| 100 | SRO i | 1520 | Check to see if 2 columns have been displayed. Rotate C(101) 1 place to right. |
| 101 | 2525 | 2525 | |
| 102 | JMP 64 | 6064 | Two columns have not been displayed; return to do next column. |
| 103 | XSK i 7 ↵ | 0227 | Two columns have been displayed; index address of the pattern word. |
| 104 | SRO i | 1520 | Skip to 107 is both halves of pattern have been displayed. |
| 105 | 2525 | 2525 | |
| 106 | JMP 64 | 6064 | Return to display 2nd half of pattern. |
| 107 | JMP 60 | 6060 | Entire pattern has been displayed once. Return and repeat. |
| 110 | 4477 | 4477 | Pattern words for letter A. |
| 111 | 7744 | 7744 | |

Example 21 Character Display of the Letter A

DSC

The SRO instructions at locations 71, 100, and 104 determine when 1 column, 2 columns, and 4 columns have been displayed. After each column the H coordinate is increased by 4 and the V coordinate reset to −10. After 2 columns the address of the pattern word is indexed by one, and after 4 columns the entire process is repeated.

DSC i β (display character), code 1740 + 20i + β, is the last of the index class instructions; it directs the LINC to display the contents of one pattern word, or 2 columns of points. Register β holds the address of the pattern word and the i-bit is used in the usual way to index X(β). The points are displayed in the format described above, i.e., 2 columns of 6 points each with a delta of 4 between points. The pattern word is examined from right to left beginning with bit 11 and points are plotted from lower left to upper right, as above. When executing a DSC instruction the computer always takes the H coordinate and channel selection from register 1. The delta of 4 is automatically added to X(1) every time a new column is begun; furthermore, this indexing is done before the first column is displayed, so that if register 1 initially contains 0364, the first column is displayed at H = 370, the second at H = 374, and register 1 contains 0374 at the end of the instruction.

The vertical coordinate is, as usual, taken from the accumulator, and again +4 is automatically added to C(ACC) between points. The rightmost 5 bits (bits 7-11) of the accumulator are always cleared at the beginning of a DSC instruction, so that if initially C(ACC) = +273, the first point will be displayed at V = 240, the second at V = 244, etc. Characters can therefore be displayed using the DSC instruction only at vertical spacings of 40 on the scope, e.g., at initial vertical coordinates equal to −77, −37, 0, +40, +100, etc. The rightmost 5 bits of the accumulator always contain $30_8$ at the end of a DSC instruction, so that if the initial C(ACC) = +273, the initial V equals +240 and C(ACC) equals +270 at the end of the instruction.

To display a character defined by a 4 x 6 pattern two DSC instructions are needed. The following example repeatedly displays the letter A in the middle of the scope, just as the program on page 48 (example 20) does, but with greater efficiency using the DSC instruction. Since an initial V = −10 is not possible with DSC, the program uses V = 0.

| Memory Address | Memory Buffer | | Effect |
|---|---|---|---|
| 1 | [0, H] | [-] | Channel selection and H coordinate. |
| 7 | [X] | [-] | Address of pattern word. |
| → 60 | CLR | 0011 | Initial V = 0 → C(ACC). |
| 61 | →SET i 1 | 0061 | Set 1 to initial H coordinate minus 4, and select channel 0. |
| 62 | 0364 | 0364 | |
| 63 | SET i 7 | 0067 | Set 7 to address of first half of pattern. |
| 64 | 110 | 0110 | |
| 65 | DSC 7 | 1747 | Display, using 1st pattern word, the left 2 columns of the letter A, at initial coordinates of (370, 0). |
| 66 | DSC i 7 | 1767 | Index address of pattern word, X(7), and display right 2 columns of the letter A at initial coordinates of (400, 0). |
| 67 | JMP 61 | 6061 | Return and repeat. |
| 110 | 4477 | 4477 | Pattern words for letter A. |
| 111 | 7744 | 7744 | |

Example 22 Character Display of the Letter A Using DSC

After the first DSC instruction (at location 65), C(1) = 0374 and C(ACC) = 30. After the second DSC instruction, C(1) = 0404, C(7) = 0111, and C(ACC) = 30. C(110) and C(111) are unchanged. By adding more pattern words at locations 112 and following locations, and repeating the DSC i 7 instruction, it is possible to display an entire row of characters.

The following program repeatedly displays a row of six digits. The pattern words for the charactors 0-9 are located in a table beginning at 1000; i.e., the pattern words for the character 0 are at 1000 and 1001, for the character 1 at 1002 and 1003, etc. Keyboard codes for the characters to be displayed are located in a half-word table from 1400-1402; i.e., the first code value is LH(1400), the second RH(1400), etc. The program computes the address of the first pattern word for each character as it is retrieved from the table at 1400.

| Memory Address | Memory Buffer | | Effect |
|---|---|---|---|
| 1 | [1, H] | [-] | Channel selection and H coordinate. |
| 2 | [-n] | [-] | Counter for number of characters. |
| 3 | [h, X] | [-] | Address of keyboard code values. |
| 4 | [X] | [-] | Address of pattern word. |
| → 20 | →SET i 2 | 0062 | Set 2 to count number of characters displayed. |
| 21 | -6 | 7771 | |
| 22 | SET i 3 | 0063 | Set 3 for loading code values beginning at LH(1400). |
| 23 | 1, 1377 | 5377 | |
| 24 | SET i 1 | 0061 | Set 1 to initial H coordinate minus 4, and select channel 1. |
| 25 | 1, 344 | 4344 | |
| 26 | →LDH i 3 | 1323 | Half-word index register 3 and put code value into accumulator. |
| 27 | ROL 1 | 0241 | Compute address of pattern word by multiplying code value by 2 and adding beginning address of pattern table. |
| 30 | ADA i | 1120 | |
| 31 | 1000 | 1000 | |
| 32 | STC 4 | 4004 | Address of pattern word → C(4); 0 → C(ACC). |
| 33 | DSC 4 | 1744 | Display character at initial V = 0, and initial H = C(1) + 4. |
| 34 | DSC i 4 | 1764 | |
| 35 | LDA i | 1020 | |
| 36 | 4 | 0004 | |
| 37 | ADM | 1140 | Increase H by 4 to provide space between characters. |
| 40 | 1 | 0001 | |
| 41 | XSK i 2 | 0222 | Index X(2) and check to see if six characters have been displayed. If not, return to get next character. If so, return to repeat entire display. |
| 42 | JMP 26 | 6026 | |
| 43 | JMP 20 ↵ | 6020 | |

Example 23 Displaying a Row of Characters

Suppose, for example, that one of the six code values is 07. The pattern words for the character 7 are at locations 1016 and 1017. Multiplying the code value 07 by 2 ($7 \times 2 = 16_8$) and adding the beginning address of the pattern table (16 + 1000 = 1016) gives us the address of the first pattern word for the character 7. It should be clear that pattern words for all the keyboard characters could be added to

the pattern table; by organizing the pattern table to correspond to the ordering of the keyboard code values, the same technique of "table look-up" using the code values to locate the pattern could be used to display any characters on the keyboard.*

## 2-11 ANALOG INPUT AND THE SAMPLE INSTRUCTION

The sample instruction, SAM i n, refers to the LINC's miscellaneous inputs. The LINC has 16 input lines ($0\text{-}17_8$) through which external analog signals may be received. The sample instruction samples the voltage on any one of these lines, and supplies the computer with instantaneous digitalized "looks" at analog information. Input lines 0-7 are built to receive signals in the range +5 to -5. These eight lines are equipped with potentiometers (which appear on the display panel as numbered black knobs) whose voltage is varied by turning the knobs. Lines 10-17, located at the data terminal module, are for high frequency signals which may range from -1 to +1v at a maximum of circa 20,000 cps.

The number n in the sample instruction specifies which line to sample. Built into the LINC are analog-to-digital conversion circuits which receive the signal and convert it to a signed 11-bit binary number in the range ±377, leaving the result in the accumulator. Thus, for example, a voltage of 0 on one of the high frequency lines will be converted to 0 when sampled with a SAM instruction, and the number 0 will be left in the accumulator. Voltages on the high frequency lines greater than or equal to +1v will, when sampled, cause $+377_8$ to be left in the accumulator. Voltages less than or equal to −1v will cause −377 to be left in the accumulator.

| Memory Address | Memory Buffer | Effect |
|---|---|---|
| → p | SAM n  100 + 20i + n | Conversion of voltage on line n → C(ACC). |

The value of this facility, which makes it possible to evaluate data while they are being generated, can easily be seen. The sample instruction is frequently used with the display instruction in this context.

To illustrate use of this instruction, look first at a simple example of a sample and display program. The following sequence of instructions samples the voltage on input line 10, and displays continuously a plot of the corresponding digital values. It provides the viewer with a continuous picture of the analog signal on that line. The sample values left in the accumulator are used directly as the vertical coordinates. In this example, input 10 is sampled.

*See chart III in appendix 2.

| Memory Address | Memory Buffer | | Effect |
|---|---|---|---|
| 17 | [0, H] | [-] | For channel selection and H coordinate. |
| → 1000 | SET i 17 | 0077 | Set register 17 to begin H coordinate at H = 0; channel 0. |
| 1001 | 1777 | 1777 | |
| 1002 | →SAM 10 | 0130 | Sample input 10, leaving its value in the ACC as the V coordinate. |
| 1003 | DIS i 17 | 0177 | Index the H coordinate and display. |
| 1004 | JMP 1002 | 7002 | Return and repeat endlessly. |

Example 24 Simple Sample and Display

Note that since here a continuous display is wanted, it is not necessary to reset register 17 to any specific horizontal coordinate.

A second example illustrates one of the uses of the potentiometers. This program plots the contents of a $512_{10}$ word segment of memory registers 0-1777. Location of the segment is selected by rotating knob 5, whose value is used to determine the address at which to begin the display. As the viewer rotates the knob, the display effectively moves back and forth across the memory.

| Memory Address | Memory Buffer | | Effect |
|---|---|---|---|
| 12 | [X] | [-] | |
| 13 | [1, H] | [-] | For channel selection, H coordinate, and counter. |
| → 20 | →SET i 13 | 0073 | Set register 13 to select channel 1 and to begin displaying at H = 0. |
| 21 | 4777 | 4777 | |
| 22 | SAM 5 | 0105 | Sample knob 5, add 400 to make the value positive, rotate left 1 place to produce an address for display, and store in register 12. |
| 23 | ADA i | 1120 | |
| 24 | 400 | 0400 | |
| 25 | ROL 1 | 0241 | |
| 26 | STC 12 | 4012 | |
| 27 | →LDA i 12 | 1032 | Index the address of the vertical coordinate, and put the coordinate into the ACC. Position it for display, index the H coordinate and display. |
| 30 | SCR 3 | 0343 | |
| 31 | DIS i 13 | 0173 | |
| 32 | XSK 13 | 0213 | Check to see whether $512_{10}$ points have been displayed. (X(13) = 1777?). |
| 33 | JMP 27 | 6027 | If not, return to display next point. |
| 34 | JMP 20← | 6020 | If so, return to reset counter and get new address from knob 5. |

Example 25 Moving Window Display under Knob Control

At locations 23-25, a memory address is computed for the first vertical coordinate by adding 400 to the sample value. This leaves the value in the range +1 to +777; it is then rotated left 1 place to produce an initial address in the range 2-1776 for the display.

A final example illustrates the technique of accumulating a frequency distribution of sampled signal amplitudes appearing on line 12, and displaying it simultaneously as a histogram. The distribution is compiled in a table at locations 1401-1777, and the sample values themselves form the addresses for table entry. Registers 1401-1777 are initially set to −377 so that the histogram will be from the bottom of the scope.

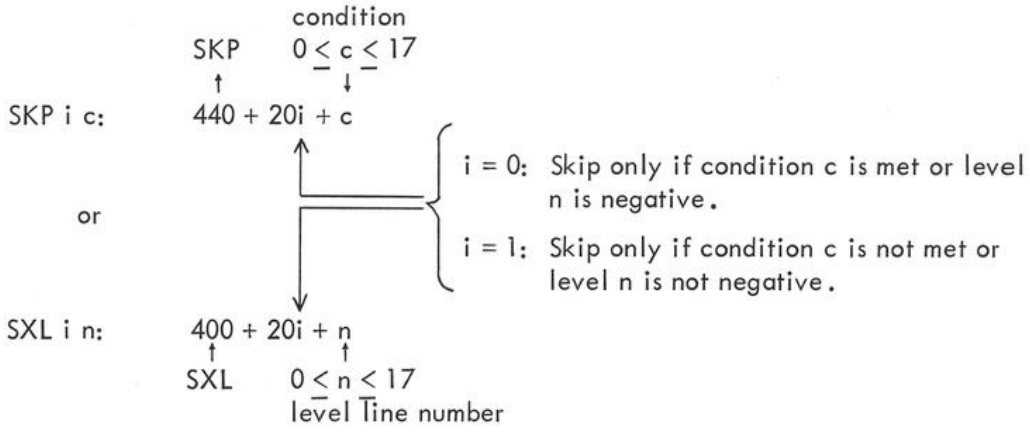
Note, at locations 104 and 105, because of using memory registers 1401-1777, the same index register (register 2) may be interpreted both as address (location 104) and counter (location 105). A separate counter is not needed because the final address (1777) will serve also as the basis of the skip decision for the XSK instruction. The same is true at location 124 and 134.

| Memory Address | Memory Buffer | | Effect |
|---|---|---|---|
| 2 | [X] | [-] | Address of vertical coordinates. |
| 3 | [0, H] | [-] | Channel selection and H coordinate. |
| ⋮ | ⋮ | ⋮ | |
| 100 | SET i 2 | 0062 | Initial routine to set registers 1401-1777 to −377. |
| 101 | 1400 | 1400 | |
| 102 | LDA i | 1020 | |
| 103 | −377 | 7400 | |
| 104 | STA i 2 | 1062 | |
| 105 | XSK 2 | 0202 | |
| 106 | JMP 104 | 6104 | |
| 107 | SET i 2 | 0062 | Set register 2 to initial address minus one of vertical coordinates. |
| 110 | 1400 | 1400 | |
| 111 | SET i 3 | 0063 | Set register 3 to select channel 0 and begin display at H=201. |
| 112 | 200 | 0200 | |
| 113 | SAM 12 | 0112 | Sample input line 12. |
| 114 | SCR 1 | 0341 | Add 1400+200 to the sample value to form an address for recording the event and store. |
| 115 | ADA i | 1120 | |
| 116 | 1600 | 1600 | |
| 117 | STC 123 | 4123 | |
| 120 | LAD i | 1020 | Add 1 to the contents of the register just located by the sample value to record the event. |
| 121 | 1 | 0001 | |
| 122 | ADM | 1140 | |
| 123 | [-] | [-] | |
| 124 | LDA i 2 | 1022 | Index register 2 and put a histogram value in the accumulator. |
| 125 | DIS i 3 | 0163 | Index the H coordinate and display. |
| 126 | DIS 3 | 0143 | Display without indexing. |
| 127 | ADA i | 1120 | Fill in the bar by decreasing the vertical coordinate by 1 and continuing the display until a point is displayed at V=−377. |
| 130 | −1 | 7776 | |
| 131 | SAE i | 1460 | |
| 132 | −400 | 7377 | |
| 133 | JMP 126 | 6126 | |
| 134 | XSK 2 | 0202 | When bar is finished, check to see whether 377 values have been displayed. (X(2) = 1777?). |
| 135 | JMP 113 | 6113 | If not, return to get next sample. |
| 136 | JMP 107 | 6107 | If so, return to reset vertical coordinate address, H coordinate, and repeat. |

Example 26 Histogram Display of Sampled Data

## 2-12 THE SKIP CLASS INSTRUCTIONS

Instructions belonging to the skip class test various conditions of the accumulator, the keyboard, the tapes, and the external level lines of the data terminal module. Coding for these instructions includes the condition or level line to be checked and an option to skip or not skip when the condition is met or the external level is negative.




In these instructions the i-bit can be used to invert the skip decision. When i = 0, the computer skips the next register in the instruction sequence when the condition is met or external level is negative. However, when i = 1, the computer skips when the condition is not met or the external level is not negative. Otherwise the computer always goes to the next register in the sequence.

The four situations which may arise are summarized in the following table. The skip class instruction is assumed to be in register p.

Branching in Skip Class Instructions

| i | Condition met or level negative? | Location of next instruction |
|---|---|---|
| 0 | yes | p + 2 (skip) |
| 0 | no | p + 1 |
| 1 | yes | p + 1 |
| 1 | no | p + 2 (skip) |

SKP i c instructions test 16 conditions, which, because of their variety, are described with different 3-letter expressions. Thus the AZE i instruction already presented is the same as SKP i 10. Another instruction, APO i, synonymous with SKP i 11, checks to see whether the accumulator is positive (bit 0 = 0):

LZE, SNS, KST
Case: i = 0

| Memory Address | Memory Buffer | | Effect |
|---|---|---|---|
| p | APO | 400+11 | If C(bit 11 of ACC) = 0, go to p + 2 for the next instruction; if C(bit 11 of ACC) = 1, go to p + 1. |
| p + 1 | - | - | |
| p + 2 | - | - | |

Case: i = 1

| Memory Address | Memory Buffer | | Effect |
|---|---|---|---|
| p | APO i | 400+20+11 | If C(bit 11 of ACC) = 1, go to p + 2 for the next instruction; if C(bit 11 of ACC) = 0, go to p + 1. |
| p + 1 | - | - | |
| p + 2 | - | - | |

Other SKP variations check whether C(L) = 0, (LZE i, code 452 + 20i, which is synonymous with SKP i 12) or whether one of the 6 sense switches on the console is up (SNS i 0, SNS i 1, ..., SNS i 5, synonymous with SKP i 0, SKP i 1, ..., SKP i 5). (The sense switches are numbered from left to right, 0-5.)

The SXL i n instruction (skip on negative external level) checks for the presence of a – 3v level on external level line n, $0 \leq n \leq 13$, at the data terminal module. It is often used with the operate instruction, discussed in the next section, to help synchronize the LINC with external equipment.

The skip instruction KST i (key struck), code 415 + 20i, checks whether a keyboard key has been struck. KST i is synonymous with SXL i 15.

To illustrate the use of these instructions the following program counts the signal peaks above a certain threshold, $100_8$, for a set of $1000_8$ samples appearing on input line 13. The number of peaks exceeding the threshold will be left in the accumulator.

| Memory Address | Memory Buffer | | Effect |
|---|---|---|---|
| 7 | [-n] | [-] | Counter for 1000 samples. |
| 10 | [n] | [-] | Counter for number above $100_8$. |
| ⋮ | ⋮ | ⋮ | |
| → 1500 | SET i 7 | 0067 | Set register 7 to count 1000 samples. |
| 1501 | -1000 | 6777 | |
| 1502 | SET i 10 | 0070 | Clear register 10 to count peaks. |
| 1503 | 0 | 0000 | |
| 1504 | →SAM 13 | 0113 | Sample input line 13 and subtract 100 from the sample value. |
| 1505 | ADA i | 1160 | |
| 1506 | -100 | 7677 | |
| 1507 | APO i | 0471 | Is the accumulator positive? |
| 1510 | XSK i 10 | 0230 | If so, the value was above 100; add 1 to the counter. If not, skip the instruction in location 1510. |
| 1511 | XSK i 7← | 0227 | Index register 7 and test. |
| 1512 | JMP 1504 | 7504 | If 1000 samples have not been taken, return. |
| 1513 | LDA← | 1000 | If 1000 samples have been taken, put the number of those above 100 into the accumulator and halt. |
| 1514 | 10 | 0010 | |
| 1515 | HLT | 0000 | |

Example 27 Counting Samples Exceeding a Threshold

Another program samples and displays continuously the input from line 14 until a letter, i.e., a key whose code value is higher than $23_8$, is struck on the keyboard.

| Memory Address | Memory Buffer | | Effect |
|---|---|---|---|
| 1 | [1, H] | [-] | Channel selection and H coordinate. |
| → 100 | SET i 1 | 0061 | Set register 1 to select channel 1 and begin display at H = 1. |
| 101 | 4000 | 4000 | |
| 102 | → SAM 14 | 0114 | Sample line 14 and display its value. |
| 103 | DIS i 1 | 0161 | |
| 104 | KST | 0415 | Has a key been struck? |
| 105 | JMP 102 | 6102 | If not, return and continue sampling and displaying. |
| 106 | KBD← | 0515 | If so, read the key code into the accumulator and subtract $23_8$ from its code value. |
| 107 | ADA i | 1120 | |
| 110 | −23 | 7754 | |
| 111 | APO | 0451 | Is ACC positive? |
| 112 | JMP 102 | 6102 | If not, the value was less than $23_8$. Return and continue sampling. |
| 113 | HLT← | 0000 | If so, the value was 24 or greater; halt. |

Example 28 Simple Sample and Display with Keyboard Control

Note that the KBD instruction at location 106 is executed only when a key has been struck (because of KST at location 104) and therefore does not need to direct the computer to pause.

## 2-13 SUBROUTINE TECHNIQUES

Before describing the remaining instructions, some mention should be made of the technique of writing subroutines. Frequently a program has to execute the same set of instructions at several different places in the program sequence. In this case it is an inefficient use of memory registers to write out the same set of instructions each time it is needed. It is more desirable to write the instructions once as a separate, or "sub," routine to which the program can jump whenever these instructions are to be executed. Once the instructions in the subroutine have been executed, the subroutine should return control (jump back) to the main program.

For example, suppose that in two different places in a program we must execute the same set of arithmetic operations. Visualize the general structure of such a program as follows:

2-13.1 Main Program

| Memory Address | Memory Buffer |
|---|---|
| Start → 100 | Main |
| | Program |
| | Instructions |
| 150 | JMP 1000 → Jump out to subroutine |
| 151 | Continue ← Return from subroutine |
| | Main |
| | Program |
| | Instructions |
| 200 | JMP 1000 → Jump out to subroutine |
| 201 | Continue ← Return from subroutine |

Subroutine

| Memory Address | Memory Buffer |
|---|---|
| Enter Subroutine → 1000 | Subroutine Instructions ↓ } Arithmetic Operations |
| 1020 | JMP MP → Return to main program |

It appears from this example that jumping to the subroutine from the main program (at locations 150 and 200) is straightforward. The subroutine must be able to return control to the main program, however, reentering it at a different place each time the subroutine is finished. That is, the JMP instruction at location 1020 must be changed so that the first time the subroutine is used it will return to the main program via a JMP 151 and the second time via a JMP 201.

It will be remembered that every time the computer executes a JMP instruction (other than JMP 0) at any location p, the instruction JMP p + 1 replaces the contents of register 0 (see page 11). Thus, when JMP 1000 is executed at location 150, a JMP 151 is automatically stored in register 0, saving the return point for the subroutine. The subroutine might retrieve this information in the following way:

OPR, EXC

2-13.2 Subroutine

| Memory Address | Memory Buffer | Effect |
|---|---|---|
| Enter Subroutine → 1000 | LDA | C(0) → C(ACC); i.e., JMP p + 1 → C(ACC). |
| 1001 | 0 | |
| 1002 | STC 1020 | C(ACC) → C(1020). |
| ⋮ | ⋮ | Execute arithmetic operations. |
| 1020 | [JMP p + 1] ← | Return to main program. |

A simple JMP 0 in location 1020 clearly suffices when the subroutine does not, during its execution, destroy the contents of register 0. In this case, the instructions in locations 1000-1002 would be unnecessary.

A problem arises in the above example when the subroutine is not free to use the accumulator to retrieve the return point. Another method, using the SET instruction, is possible when there is an available β register.

| Memory Address | Memory Buffer | Effect |
|---|---|---|
| Enter Subroutine → 1000 | SET 10 | C(0) → C(10; i.e., JMP p + 1 is saved in a free β register. |
| 1001 | 0 | Execute arithmetic operations; the accumulator has not been disturbed. |
| ⋮ | ⋮ | |
| 1020 | JMP 10 | Return to main program by jumping to register 10. |

## 2-14 PROCESSOR INTERCOMMUNICATION

### 2-14.1 Control Transfer Between Processors

There are two cases to consider in transferring control between processors. The first and simplest case is where no dispatching is required: the processor not in use is required to execute only one subprogram. The second case requires selection among several subprograms or subroutines for execution in the alternate processor mode. In such a case, modification of the alternate processor's program counter is necessary.

OPR, EXC

An example of the first, nondispatching case is that of a PDP-8 program using the LINC processor strictly for display. Another example would be a LINC program relying on the PDP-8 processor to service only one peripheral device, such as 138E/139E ADC and multiplexer control (not on interrupt). The second, more general, case is well exemplified by the program of operation, PROGOFOP, a collection of subroutines to service many of the special LINC features. The appropriate subroutine is called into play for each particular condition to be serviced.

Let us examine the nondispatching case first. No changes need be made to the program counter register. Transfer of control back and forth is straightforward. Since the PDP-8 processor is in data break when the LINC processor is operational, a simple (LINC) HLT instruction stops LINC operation and transfers control back to the PDP-8. Program operation resumes in the PDP-8 at the location following the one in which control was transferred to the LINC. Similarly, to transfer control from the PDP-8 to the LINC, the instruction sequence, CLA, TAD (12), ICON, is executed. The LINC subprogram resumes where it was interrupted by HLT. The programmer can arrange jumps and halts so that a CLA, TAD, ICON sequence is an effective call to a LINC processor subroutine. In the same way a HLT can be effectively a call to a PDP-8 subroutine. An example follows below.

The most efficient means by which the LINC scope is used in a PDP-8 program takes advantage of the shared memory between the two processors. Control is transferred to a short LINC program to display a list of points. The LINC processor transfers control back to the 8-processor upon encountering a halt instruction (HLT).

The PDP-8 program is given in detail below:

| Memory Address | Memory Buffer | Effect |
|---|---|---|
| MAIN, | ..... | |
| | ..... | |
| | CLA | |
| | TAD (12) | /12 OCTAL, TRANSFERS |
| | | /CONTROL TO LINC PROCESSOR |
| EXIT, | ICON | /AT LOCATION IN LINC P |
| | NOP | /REGISTER (LAST PDP-8 INST.) |
| RETURN, | ..... | /CONTROL IS RETURNED HERE |
| | ..... | |

The LINC program is given in detail below:

| Memory Address | | Memory Buffer | Effect |
|---|---|---|---|
| 20 | DISPL, | LDA i 2 | /LOADS LINC AC WITH CONTENTS /OF LOC SPECIFIED BY XR2 |
| | | DIS i 3 | /DISPLAYS: X,Y /Y=(AC), (XR3)=(XR3)+1, X= /(XR3) |
| | | XSK i 4 | /SEE IF ENTIRE LIST HAS BEEN /DISPLAYED |
| | | JMP 20 | /MORE DISPLAYING |
| | SETUP, | SET i 2 | /RESET DISPLAY POINTER TO: |
| | | LIST 1 | /LOCATION PREVIOUS TO LIST /OF POINTS |
| | | SET i 4 | /RESET COUNTER TO: |
| | | POINTS | /NUMBER OF POINTS TO BE /DISPLAYED |
| | | SET i 3 | /RESET X-AXIS CONTROL |
| | | -1 | |
| | | HLT | /RETURN CONTROL TO PDP-8 |
| | | JMP 20 | /NEXT CALL WILL RETURN TO /THIS LOCATION |

LINC INITIALIZATION (DONE ONCE ONLY)

(A PDP-8 SUBPROGRAM)

| | | |
|---|---|---|
| INITL, | CLA | |
| | TAD (SETUP | |
| | ISSP | /8 AC TO LINC PROGRAM COUNTER |
| | CLA | |
| | TAD (10) | /ENABLE LINC SECTION |
| | ICON | |
| | TAD (2 | /12 IN AC, GET READY TO /TRANSFER CONTROL. SEE LINC-8 /USERS HANDBOOK |
| | ICON | /GO TO LINC PROGRAM AT /"SETUP" |
| | NOP | /SETUP IS COMPLETE (CONTROL /RETURNED HERE) |
| | ..... | /OTHER INITIALIZATIONS |
| | ..... | |
| | JMP MAIN | |

In the second case, control must be transferred to the appropriate other-processor subroutine. This can be done when going from the PDP-8 to the LINC by setting the LINC P register with an ISSP instruction before starting the LINC (with a CLA, TAD, ICON sequence). The LINC program then starts at the location specified in the P register. Control may be transferred from the LINC main program to a PDP-8 subroutine by means of the operate (OPR) and execute (EXC) classes of LINC instructions. LINC execution of these instructions transfers control to PROGOFOP, the "program of operation." PROGOFOP determines that a call to the 8-processor has taken place. PROGOFOP has retrieved the instruction from the LINC processor and retains it in the PDP-8 accumulator. By examining the AC the user may determine which subroutine to call. He must change a few locations in PROGOFOP to accomplish this. These changes along with the other programming required to implement such a subroutine are described below.

PROGOFOP sections which need to be modified for user-defined OPR and EXC instructions follow:

| Memory Address | Memory Buffer | Effect |
|---|---|---|
| PROGOFOP NOW READS: | | |
| EXECUT, | TAD LINSTR | /LINC INSTR TO AC |
| | NOP | |
| ALTERATION TO PROGOFOP: | | |
| EXECUT, | TAD LINSTR | |
| | JMS EXCTAB | /LINC INSTR TO AC<br>/JMP TO DISPATCHING<br>/ROUTINE |

The user would change the "NOP" to JMP EXCTAB, (jump to EXC class dispatching subroutine).

JMS is a jump to subroutine (in PDP-8 coding). The jump is made to EXCTAB + 1, and the PDP-8 program counter is stored in location EXCTAB. To return to PROGOFOP the user should jump to the location held in EXCTAB, that is JMP I EXCTAB. PROGOFOP recalls the LINC processor at the location following the EXC. A program which transfers control to the appropriate EXC subroutine is shown below (this is called dispatching).

| Memory Address | Memory Buffer | Effect |
|---|---|---|
| EXCTAB, | 0 | |
| | AND (37) | /MASK FOR n of EXCn |
| | TAD (JMP I EXCGOT) | |
| | DCA JUMPEX | /AC HOLDS JMP I EXCGOT+n |
| JUMPEX, | 0 | /WILL HOLD A JUMP |
| | | /INSTRUCTION |
| EXCGOT, | EXC0 | /LOCATIONS OF |
| | EXC 1 | /EXC SUBROUTINES |
| | EXC 2 | |
| | --- | |
| | --- | |
| | --- | |
| EXCn, | | /A SAMPLE EXC SUBROUTINE |
| RETURN, | JMP I EXCTAB | /RETURN TO LINC PROGRAM |

The OPR class is handled in a similar fashion:

PROGOFOP reads:

OPERATE, NOP

The user would change the "NOP" to "JMS, OPRDO" (jump to operate dispatching subroutine).

| Memory Address | Memory Buffer | Effect |
|---|---|---|
| OPRDO, | 0 | /HOLDS LOC IN PROGOFOP |
| | | /TO WHICH RETURN SHOULD BE MADE |
| | AND (37 | /MASK FOR n of EXCn |
| | TAD (JMP I OPSORT | /AC HOLDS JMP INSTRUCTION |
| | DCA JUMP | /"JUMP" HOLDS JMP I OPSORT+n |
| JUMP, | 0 | /WILL DO A JUMP TO |
| | | /PROPER LOC IN TABLE |
| OPRSORT, | OPR0 | /LOCATIONS OF OPR SUBROUTINES |
| | OPR1 | |
| | ----- | |
| | ----- | |

An operate subroutine would then be written in normal PDP-8 coding with control being returned through PROGOFOP to the LINC program.

```
OPRn,   ------
        ------
        ------
        JMP I OPRDO        /RETURN TO LINC PROGRAM
```

The following OPR instructions are already defined in PROGOFOP.

```
OPR 13      /EFFECTIVELY JMS TO A
            /8-SUBROUTINE AT
            /LOCATION SPECIFIED
            /BY (NON-ZERO) LINC ACCUMULATOR
OPR 14      /TYPE OUT ASCII CHARACTER
            /IN LINC ACCUMULATOR
OPR 15      /READ KEYBOARD
OPR 16      /READ RIGHT SWITCHES
OPR 17      /READ LEFT SWITCHES
```

A common subroutine calling sequence holds the locations of the arguments directly after the calling location. What this implies is that PDP-8 subroutines must be able to read the LINC program counter and that LINC subroutines must be able to read the PDP-8 program counter. In the first case, parameters can be accessed via the ISSP, IBAC sequence. Modification of the LINC program counter can be made while in the 8 mode by the ISSP instruction. This P register modification is necessary to prevent returning control to the LINC at the middle of a list of parameters.

For parameter transmission from a PDP-8 program to a LINC subroutine, the LINC routine must be able to determine the contents of the PDP-8 program counter. In order to do this a sequence of instructions must be executed before control is transferred to the LINC. The sequence of instructions is as follows:

| Memory Address | Memory Buffer | Effect |
|---|---|---|
| GOLINC, | JMS PCSTOR | /GET PROGRAM COUNTER |
| PCSTOR, | 0 | /WILL HOLD PC |
| | TAD PCSTOR | /GET PC INTO AC |
| | TAD (5 | /(AC) + 5 IS LOCATION OF |
| | | /ICON. AC HOLDS ICON LOCATION |
| | DCA XR1 | /STORE AC IN XR1 |
| | | /XR1 IS ALPHA REGISTER 1 |
| | | /OF THE LINC PROCESSOR |
| | TAD (12 | /TRANSFER CONTROL TO LINC |
| | ICON | |
| | A | /PARAMETER |
| | B | /PARAMETERS |
| | C | /PARAMETERS |
| | .... | |

The LINC alpha register 1 holds the PDP-8 location at which control was relinquished. Thus, a LINC processor instruction LDA i 1 gets the location of first parameter of the subroutine into the accumulator. The sequence LDA i 1, STC 2, LDA 2 gets the parameter itself. Succeeding parameters may be loaded into the accumulator through use of the same instruction sequence since index register 1 is incremented each time the sequence is executed. Control can be returned to the proper PDP-8 location via the OPR 13 instruction. Alpha register 1 holds the location of the last parameter. Control should then be returned to the PDP-8 at the location following the last parameter.

2-14.2 Example of use of the OPR 13 instruction (LINC Program).

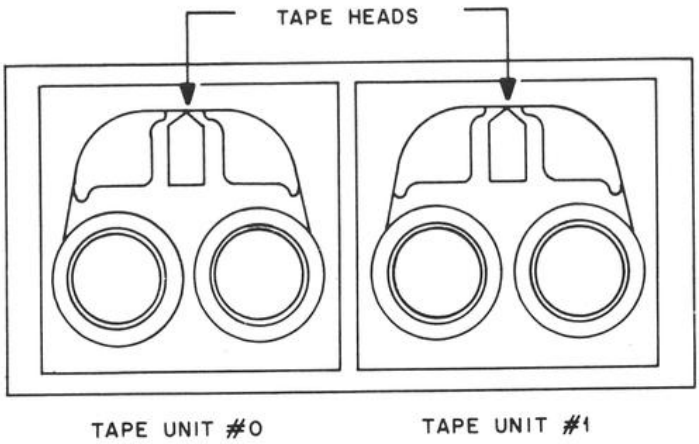
| Memory Address | Memory Buffer | Effect |
|---|---|---|
| | LDA i | /THE AC HOLDS PDP-8 LOCATION |
| | SUBR8 | /TO WHICH CONTROL IS TRANSFERRED |
| GOTO8 | OPR 13 | /GO TO PDP-8 PROGRAM |
| RESUME, | ..... | /RESUME LINC PROGRAM |
| | ..... | |
| | ..... | |
| 8 Subroutine | | |
| SUBR8, | 0 | /HOLDS LOCATION IN PROGOFOP |
| | | /TO WHICH RETURN SHOULD BE |
| | | /MADE FOR RESTARTING THE |
| | | /LINC |
| | ..... | |
| | ..... | |
| | ..... | |
| | JMP I SUBR | /PROGOFOP WILL RETURN |
| | | /CONTROL TO THE LINC AT THE |
| | | /LOCATION FOLLOWING THE OPR 13 |

## 2-15 MAGNETIC TAPE INSTRUCTIONS

The last class of instructions, for magnetic tape, requires some discussion of LINC tape units and tape format. The LINC uses small reel (3-3/4 inch diameter) magnetic tapes for storing programs and data. There are two tape units on a single panel:

RDE

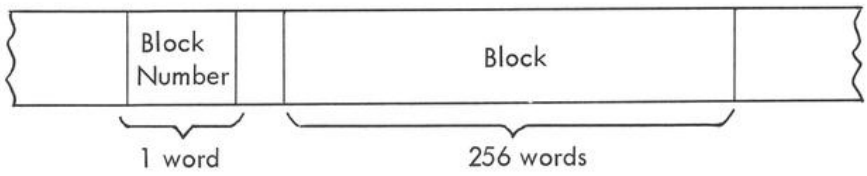
LINC MAGNETIC TAPES




Any magnetic tape instruction may refer to either the tape on unit 0 or the tape on unit 1; which unit to use is specified by the instruction itself; only one unit, however, is every used at one time.

In the original LINC, handling of magnetic tape and its instructions was done entirely by the computer hardware. However, in the LINC-8 system, the tape and its instructions is handled by software, a program which permanently resides in PDP-8 core memory area to handle magnetic tape, as well as other input/output and special feature functions. This program is called PROGOFOP (program of operation). PROGOFOP interprets some of the LINC's instructions as well as handling most input/output for the LINC. This has no effect on the eventual result of the LINC instructions; it merely means that hardware functions have been replaced with software equivalents.

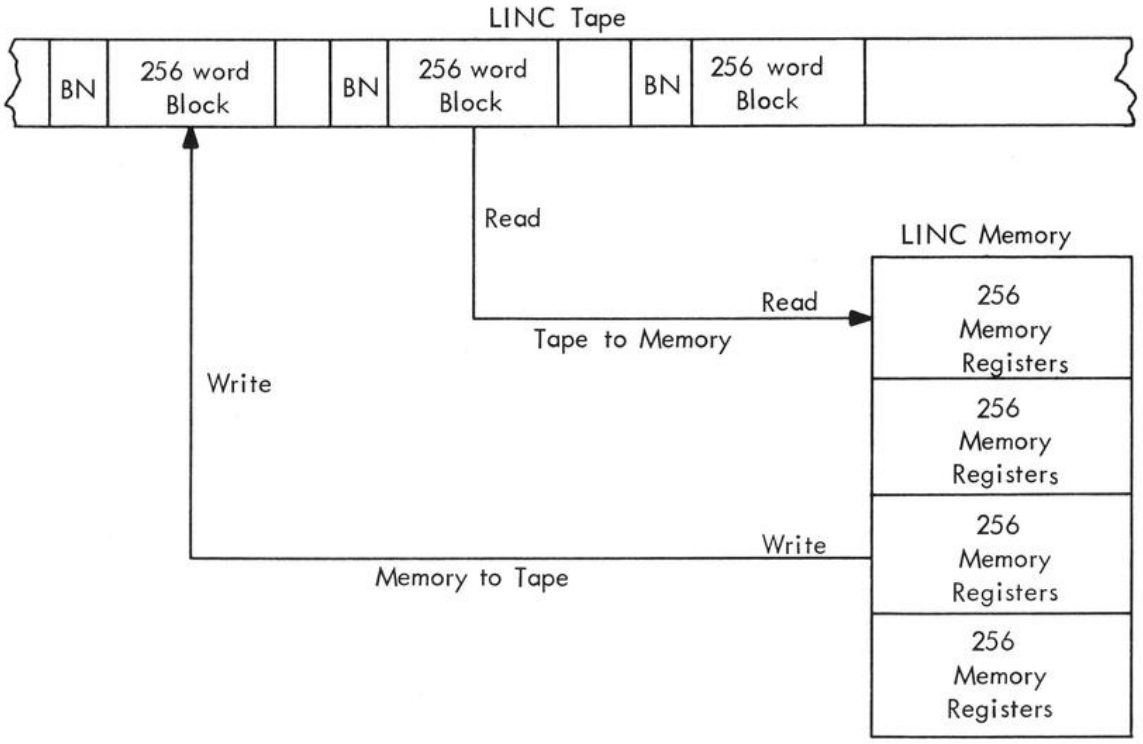
A LINC tape can hold 131, 072, 12-bit words of information, or the equivalent of $128_{10}$ full LINC memories. It is divided into $512_{10}$ smaller segments known as blocks, each of which contains $256_{10}$ 12-bit words, a size equal to one-quarter of LINC memory. Blocks are identified on any tape by block numbers, $0\text{-}777_8$; magentic tape instructions specify which block to use by referring to its block number. A block number (BN) on the tape permanently occupies a 12-bit space preceding the 256 words of the block itself:




There are other special words on the tape, serving other functions, which complete the tape format. Before describing these, however, look more specifically at one of the magnetic tape instructions, RDE i u (read tape).

## 2-15.1 Block Transfers and Checking

Read tape is one of six magnetic tape instructions which copy information either from the tape into the LINC memory (reading), or from the memory onto the tape (writing). These are generally called block transfer instructions because they transfer one or more blocks of information between the tape and the memory:




All magnetic tape instructions are double register instructions. RDE, typical of a block transfer instruction, is written:

| Memory Address | Memory Buffer | |
|---|---|---|
| p | RDE i u | 702 + 20i + 10u |
| p + 1 | QN (3 bits) \| BN (9 bits) | 1000QN + BN |

The first register of the instruction has two special bits. The u-bit (bit 8) selects the tape unit: when u = 0, the tape on unit 0 is used; when u = 1, the tape on unit 1 is used. Magnetic tape instructions require that the tape on the selected unit move at a speed of approximately 60 ips. Therefore, if the tape is not moving when the computer encounters a magnetic tape instruction, tape motion is started automatically and the computer waits until the tape has reached the required speed before continuing with the instruction.

The i-bit (bit 7) specifies the motion of the tape after the instruction is executed. If i = 0, the tape will stop; if i = 1, it will continue to move at 60 ips. It is sometimes more efficient to let the tape continue to move, as, perhaps, to execute several magnetic tape instructions in succession. If it stops, it is necessary to wait for it to start again at the beginning of the next tape instruction. Examples of this will be given later.

In the second register of the RDE instruction, the rightmost 9 bits hold the requested block number, BN; that is, they tell the computer which block on the tape to read into the memory. The left 3 bits hold the quarter number, QN, which refers to the memory. QN specifies which quarter of memory to use in the transfer. The quarters of the LINC memory are numbered 0-7,* and refer to the memory registers as follows:

| Quarter Number | Memory Registers (octal) |
|---|---|
| 0 | 0 - 377 |
| 1 | 400 - 777 |
| 2 | 1000 - 1377 |
| 3 | 1400 - 1777 |
| 4 | 2000 - 2377 |
| 5 | 2400 - 2777 |
| 6 | 3000 - 3377 |
| 7 | 3400 - 3777 |

Suppose, for example, the programmer wishes to transfer data stored on tape into memory registers 1000-1377. The data is in block 267 and the tape mounted on unit 1:

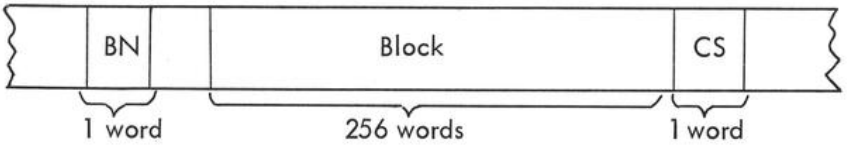
| Memory Address | Memory Buffer | | Effect |
|---|---|---|---|
| → 200 | RDE u | 0712 | Select unit 1; C(block 267) → C(quarter 2). |
| 201 | 2 \| 267 | 1000x2+267 | |

This instruction will start to move the tape on unit 1 if it is not already moving. It then reads block 267 on that tape into quarter 2 of memory and stops the tape when the transfer is completed. The computer will go to location 202 for the next LINC instruction. After the transfer the information in block 267 is still on the tape; only memory registers 1000-1377 and the accumulator are affected. Conversely, writing affects only the tape and the accumulator: the memory is left unchanged.

*See appendix 1.

WRI

Another special word on the tape, located immediately following the block, is called the checksum, CS:




The checksum, a feature common to many tape systems, checks the accuracy of the transfer of information to and from the tape. On a LINC tape, the checksum is the complement of the sum of the 256 words in the block. Such a number is formed during the execution of another block transfer instruction, WRI i u (write tape). This instruction writes the contents of the specified memory quarter in the specified block of the selected tape:

| Memory Address | Memory Buffer | |
|---|---|---|
| p | WRI i u | 706 + 20i + 10u |
| p + 1 | QN\|BN | 1000QN + BN |

During the transfer the words being written on the tape are added together without end-around carry in the accumulator. This sum is then complemented and written in the CS space following the block on the tape. After the operation the checksum is left in the accumulator and the computer goes to p + 2 for the next LINC instruction. QN, BN, i, and u are all interpreted as for RDE.

One means of checking the accuracy of the transfer is to form a new sum and compare it to the checksum on the tape. This happens during RDE: the 256 words from the block on the tape are added together without end-around carry in the accumulator while they are being transferred to the memory. This uncomplemented sum is called the data sum. The checksum from the tape is then added to this data sum and the result, called the transfer check, is left in the accumulator. If the information has been transferred correctly, the data sum will be the complement of the checksum, and the transfer check will equal −0 (7777): the block "checks." Thus, by examining the accumulator after an RDE instruction, the programmer can tell if the block was transferred correctly. The following sequence of instructions does this and reads block 500 again if it does not check:

RDC

| Memory Address | Memory Buffer | | Effect |
|---|---|---|---|
| → 300 | →RDE | 0702 | Read block 500, unit 0, into quarter 3. Leave the transfer check in the accumulator and stop the tape. |
| 301 | 3\|500 | 3500 | |
| 302 | SAE i | 1460 | Skip to location 305 if C(ACC) = 777, i.e., if the block checks. If C(ACC) ≠ 7777, return to read the block again. |
| 303 | 7777 | 7777 | |
| 304 | JMP 300 | 6300 | |
| 305 | - ←- | - | |
| ⋮ | ⋮ | ⋮ | |

The remaining block transfer instructions check transfers automatically. RDC i u (read and check), does in one instruction exactly what the above sequence of instructions does. That is, it reads the specified block of the selected tape into the specified quarter of memory and forms the transfer check in the accumulator. If the transfer check does not equal 7777, the instruction is repeated (the block is reread, etc.). When the block is read correctly, 7777 is left in the accumulator and the computer goes on to the next LINC instruction at p + 2. The RDC instruction is written:

| Memory Address | Memory Buffer | |
|---|---|---|
| p | RDC i u | 700+20i+10u |
| p + 1 | QN\|BN | 1000QN+BN |

One of the most frequent uses of instructions which read the tape is to put LINC programs stored on tape into the memory. Suppose the programmer is given a tape, for example, which has in block 300 a program he wants to run. The program is $100_8$ registers long, starting in register 1250. He can mount the tape on either unit and then set and execute either RDE or RDC in the LEFT and RIGHT SWITCHES. If he uses RDE, he should look at the ACCUMULATOR lights after the transfer to make sure the transfer check = 7777. When double register instructions are set in the toggle switches, the first word is set in the LEFT SWITCHES, and the second in the RIGHT SWITCHES. If the tape is on unit 1, to use RDC the toggle switches should be set as follows:

CHK

| Console Location | Contents | |
|---|---|---|
| LEFT SWITCHES | RDC u | 0710 |
| RIGHT SWITCHES | 2\|300 | 2300 |

QN = 2 because the program in block 300 must be stored in memory registers 1250-1347, which are located in quarter 2. Raising the DO lever will cause the LINC to read the block into the proper quarter and check it. Start at 1250 from the console, using the RIGHT SWITCHES.

The remaining block transfer instructions will be described later.

A non-transfer instruction, CHK i u (check tape), makes it possible to check a block without destroying information in the memory. This instruction does exactly what RDE does, except that the information is not transferred into the memory; that is, it reads the specified block into the accumulator only, forms the data sum, adds it to the checksum from the tape, and leaves the result, the transfer check, in the accumulator. Since this is a non-transfer instruction, QN is ignored by the computer. Otherwise this instruction is written as are the other instructions:

| Memory Address | Memory Buffer | |
|---|---|---|
| p | CHK i u | 707+20i+10u |
| p + 1 | BN | BN |

The following program checks sequentially all the blocks on the tape on unit 0. The program starts at location 200. If a block does not check, the program puts its block number into the accumulator and halts at location 221. To continue checking, reenter the program at location 207. The program will halt at location 216 when it has checked the entire tape.

WRC

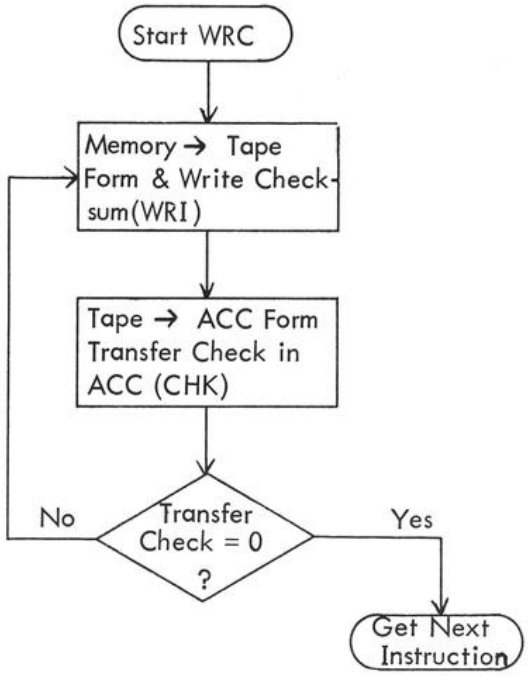
| Memory Address | Memory Buffer | | Effect |
|---|---|---|---|
| Start → 200 | CLR | 0011 | Store 0 in register 203 as first block number. |
| 201 | STC 203 | 4203 | |
| 202 | →CHK i | 0727 | Check the block specified in register 203; transfer check → C(ACC); the tape continues to move. |
| 203 | [BN]← | [-] | |
| 204 | SAE i | 1460 | If the transfer check = −0, skip to location 207. |
| 205 | 7777 | 7777 | |
| 206 | JMP 217 | 6217 | If the block does not check, jump to location 217. |
| Reenter ⇒ 207 | LDA i | 1020 | Add 1 to the block number in register 203, and leave the sum in the accumulator. |
| 210 | 1 | 0001 | |
| 211 | ADM | 1140 | |
| 212 | 203 | 0203 | |
| 213 | SAE i | 1460 | If all the blocks have been checked, skip to location 216. Otherwise return to check next block. |
| 214 | 1000 | 1000 | |
| 215 | JMP 202 | 6202 | |
| 216 | HLT← | 0000 | |
| 217 | →LDA | 1000 | Load the block number of the block which failed into the accumulator, and halt. |
| 220 | 203 | 0203 | |
| 221 | HLT | 0000 | |

Example 29 Simple Check of an Entire Tape

A block transfer instruction, WRC i u, (write and check), combines the operations of the instructions WRI and CHK, and, like read-and-check, repeats the entire process if the check fails. That is, WRC writes the contents of the specified memory quarter in the specified block, forms the checksum in the accumulator and writes the checksum onto the tape. It then checks the block just written. If the resulting transfer check does not equal −0, the block is rewritten and rechecked. When the block checks, 7777 is left in the accumulator and the computer goes on to the next LINC instruction at p + 2. WRC is written:

| Memory Address | Memory Buffer | |
|---|---|---|
| p | WRC i u | 704 + 20i + 10u |
| p + 1 | QN\|BN | 1000QN + BN |

This write-and-check process may be diagrammed:




The following sequence illustrates the use of some of the block transfer instructions. Since the LINC memory is small, a program must frequently be divided into sections which will fit into tape blocks, and the sections read into the memory as needed. This example saves (writes) the contents of quarter 2 of memory (registers 1000-1377) on the tape. It then reads a program section from the tape into quarters 1, 2, 3 (register 400-1777) and jumps to location 400 to begin the new section of the program. Assume that the tape is on unit 0. Memory quarter 2 will be saved in block 50; the program to be read from the tape is in blocks 201-203:

| Memory Address | Memory Buffer | | Effect |
|---|---|---|---|
| → 100 | WRC i | 0724 | C(quarter 2) → C(block 50); transfer is checked, and the tape continues to move. |
| 101 | 2\|50 | 2050 | |
| 102 | RDC i | 0702 | C(block 201) → C(quarter 1), and C(block 202) → C(quarter 2); transfers are checked and the tape continues to move. |
| 103 | 1\|201 | 1201 | |
| 104 | RDC i | 0720 | |
| 105 | 2\|202 | 2202 | |
| 106 | RDC | 0720 | C(block 203) → C(quarter 3); transfer is checked and the tape stops. |
| 107 | 3\|203 | 3202 | |
| 110 | JMP 400 | 6400 | Jump to the new section. |
| ⋮ | ⋮ | ⋮ | |
| 400 | →[-] | [-] | |

Example 30 Dividing Large Programs Between Tape and Memory

At the end of the above sequence, the contents of memory registers 400-1777 and tape block 50 have been altered; quarter 0 of memory, in which the sequence itself is held, is unaffected.

Another program repeatedly fills quarter 3 with samples from input line 14 and writes the data in consecutive blocks on tape beginning at block 200. The number of blocks of data to collect and save is specified by the setting of the RIGHT SWITCHES. When the requested number has been written, the program saves itself in block 177 and halts. The tape is on unit 1.

| Memory Address | Memory Buffer | | Effect |
|---|---|---|---|
| 10 | [X] | [-] | Memory address for storing samples. |
| 11 | [-n] | [-] | Counter. |
| ⋮ | ⋮ | ⋮ | |
| → 1000 | RSW | 0516 | C(RIGHT SWITCHES) → C(ACC). Complement the number and store in register 11. |
| 1001 | COM | 0017 | |
| 1002 | STC 11 | 4011 | |
| 1003 | →SET i 10 | 0070 | Set register 10 to store samples beginning at 1400. |
| 1004 | 1377 | 1377 | |
| 1005 | →SAM 14 | 0114 | Sample input line 14, store value and repeat until $400_8$ samples have been taken. |
| 1006 | STA i 10 | 1070 | |
| 1007 | XSK 10 | 0210 | |
| 1010 | JMP 1005 | 7005 | |
| 1011 | WRC u← | 0714 | When quarter 3 is full, write it on tape and check the tape stops. |
| 1012 | [3 200] | [-] | |
| 1013 | LDA i | 1020 | Add 1 to the BN in register 1012. |
| 1014 | 1 | 0001 | |
| 1015 | ADM | 1140 | |
| 1016 | 1012 | 1012 | |
| 1017 | XSK i 11 | 0231 | Index the counter and skip if the requested number has been collected. |
| 1020 | JMP 1003 | 7003 | If not, return. |
| 1021 | WRC u← | 0714 | If so, write this program in block 177, check the transfer and stop the tape. |
| 1022 | 2\|177 | 2177 | Halt the computer. |
| 1023 | HLT | 0000 | Halt the computer. |

Example 31 Collecting Data and Storing on Tape

Since the program saves itself when finished, the operator can continue to collect data at a later time by reading block 177 into quarter 2, and starting at 1000. Since the BN in location 1012 will have been saved, the data will continue to be stored in consecutive blocks.

## 2-15.2 Group Transfers

Two other block transfer instructions, similar to RDC and WRC, permit a program to transfer as many as 8 blocks of information with one instruction. These are called the group transfer instructions;

they transfer information between consecutive quarters of the memory and a group of consecutive blocks on the tape. Suppose, for example, that we want to read 3 blocks from the tape into memory quarters 1, 2, and 3. The 3 tape blocks are 51, 52, and 53. Using the instruction RCG i u (read and check group), write:

| Memory Address | Memory Buffer | |
|---|---|---|
| p | RCG i u | 701 + 20i + 10u |
| p + 1 | 2\|51 | 2051 |

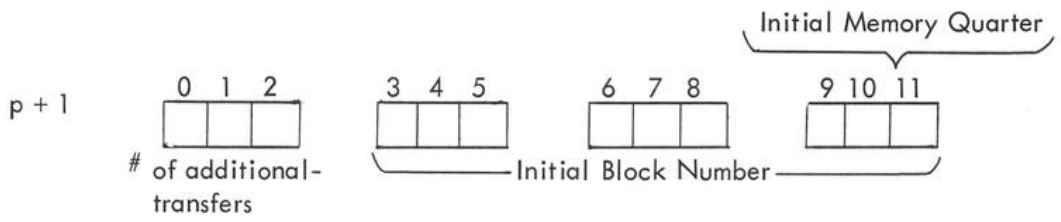
The first register specifies the instruction, the tape unit, and the tape motion as usual. The second register, however, is interpreted somewhat differently. It uses BN to select the first block of the group. In addition, the rightmost 3 bits of BN specify also the first memory quarter of the group. That is, block 51 will be read into memory quarter 1, (block 127 would be read into memory quarter 7, etc.). The leftmost 3 bits (usually QN are used to specify the number of additional blocks to transfer. In the above example, block 51 is read into quarter 1, and 2 additional blocks are transferred: block 52 into quarter 2 and block 53 into quarter 3.

The format for WCG i u (write and check group) is the same as for RCG:

| Memory Address | Memory Buffer | |
|---|---|---|
| p | WCG i u | 705 + 20i + 10u |
| p + 1 | 3\|300 | 3300 |

The computer interprets the above example as: write and check quarter 0 in block 300, and make 3 additional consecutive transfers: quarter 1 into block 301, quarter 2 into block 302, and quarter 3 into block 303. When the leftmost 3 bits are 0, i.e., 0 additional transfers, the WCG instruction is like the WRC instruction in that only 1 block is transferred.

The second word of a group transfer instruction may be diagramed:



RCG and WCG always operate on consecutive memory quarters and tape blocks. Specifying 3 additional transfers when the initial block is, say, 336, will transfer information between tape blocks 336, 337, 340, and 341, and memory quarters 6, 7, 0, and 1; that is, quarter 0 succeeds quarter 7. The tranfers are always checked; when a transfer does not check, the instruction is repeated starting with the block that failed. With WCG, all the blocks and their checksums are first written, and then all are checked. If any block fails to check, the blocks are rewritten beginning with the block that failed, and then all blocks are checked again. As with RDC and WRC, group transfer instructions leave −0 in the accumulator and go to p + 2 for the next LINC instruction.

Using RCG instead of RDC, the program example on page 78 can be written more effeciently:

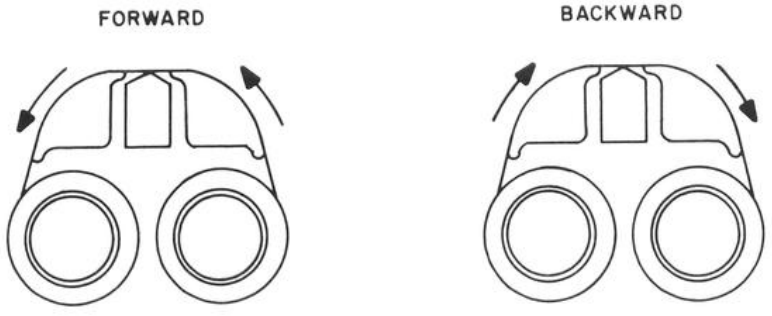
| Memory Address | Memory Buffer | | Effect |
|---|---|---|---|
| → 100 | WRC i | 0724 | C(quarter 2) → C(block 50); transfer is checked and tape continues to move. |
| 101 | 2\|50 | 2050 | |
| 102 | RCG | 0701 | Read blocks 201-203 into quarters 1-3; check the transfers and stop the tape. |
| 103 | 2\|201 | 2201 | |
| 104 | JMP 400 | 6400 | Jump to the new section. |

Example 32 Tape and Memory Exchange with Group Transfer

## 2-15.3 Tape Motion and the Move Toward Block Instruction

When the computer is searching the tape for a required block, it looks at each block number in turn until it finds the correct one. Since the tape may be positioned anywhere when the search is begun, it must be able to move either forward or backward to find the block.

Forward means moving from the low block numbers to the high numbers; physically the tape moves onto the lefthand reel.

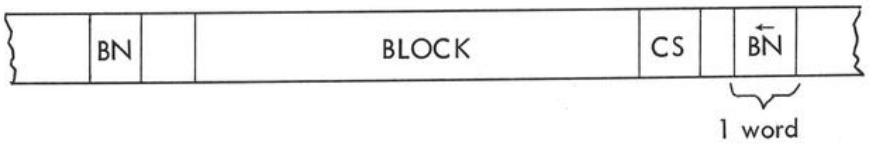


Backward means moving from the high numbers to the low; the tape moves onto the righthand reel.

When searching for a requested block, the computer decides whether the tape must move forward or backward by subtracting each block number it finds from the requested number, and using the sign of the result to determine the direction of motion. If the difference is positive, the search continues in the forward direction; if negative, it continues in the backward direction. This may, of course, mean that the tape has to reverse direction in order to find the required block.

Suppose, for example, that the computer is instructed to read block 50, and that the tape is presently moving forward just below block 75. The next block number found will be 75. The result of subtracting 75 from 50 is −25, which indicates not only that the tape is 25 blocks away from block 50, but also that block 50 is below the present tape position. The tape will reverse its direction and go backward.

To facilitate searching in the backward direction a special word called a backward block number, $\overleftarrow{BN}$, follows the checksum for each block:




When searching in the forward direction, the computer looks at forward block numbers, BN; when searching in the backward direction, it looks at backward block numbers, $\overleftarrow{BN}$. In either direction, each block number found is subtracted in turn from the requested number, and the direction reverses as necessary, until the result of the subtraction is −0 in the forward direction. Transfers and checks are made only in the forward direction.

Thus, in the above example, the tape will continue to move in the backward direction until the result of the subtraction is positive, i.e., until the BN for block 49 is found and subtracted from 50, indicating that the tape is now below block 50. The direction will be reversed, the computer will find 50 as the next forward block number, BN, and the transfer will be made because −0 is the result of the subtraction and the tape is moving forward.

For all magnetic tape instructions, if the tape is not moving when the instruction is encountered, the computer starts the tape in the forward direction and waits until it is moving at the required speed before reading a forward block number, BN, and reversing direction if necessary. If the tape is in motion, however, (including coasting to a stop), the computer does not change direction until block number comparison requires it.

For all tape transfer or check instructions with i = 1, the tape continues to move forward after the instruction is executed.

MTB

For all magnetic tape instructions stops are made in the backward direction. For transfer or check instructions this means that the tape always reverses before stopping. Furthermore, the tape then stops below the last block involved in the instruction, so that when the tape is restarted, this block will be the first one found. This reduces delay in programs which make repeated references to the same block.

The last magnetic tape instruction illustrates some to the tape motion characteristics. MTB i u (move toward block) is written:

| Memory Address | Memory Buffer | |
|---|---|---|
| p | MTB i u | 703 + 20i + 10u |
| p + 1 | BN | BN |

As in the other magnetic tape instructions, the u-bit selects the tape unit. The tape motion bit (the i-bit) and the second register, however, are interpreted somewhat differently. MTB directs the LINC to subtract the next block number it finds on the tape from the number specified in the second word of the instruction, and leave the result in the accumulator. QN is ignored during execution of MTB. For example, if the block number in the second register of the instruction is 0, and the tape is just below block 20 and moving forward, then −20, or 7757, will be left in the accumulator. The MTB instruction can thus be used to find out where the tape is at any particular time.

When i = 0, the tape is stopped as usual after the instruction is executed. When i = 1, however, the tape is left moving toward the specified block. The result of the subtraction is left in the accumulator, and the tape direction is reversed if necessary as the computer goes on to the next instruction. MBT i does not actually find the block; it merely orients the tape motion toward it.

The initial direction of motion and possible reversal are determined for MTB just as they are for all other magnetic tape instructions, as described above. Note, however, that since MTB i makes no further corrections to the direction of motion, the specified block may eventually be passed.

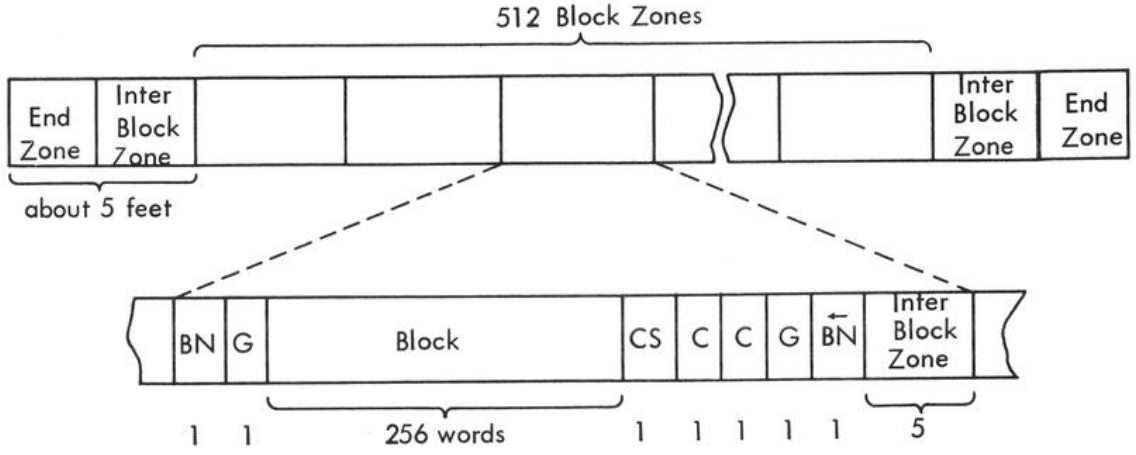
The move-toward-block instruction serves not only to identify tape position, but also to save time. If, for example, a program must read block 700, and then, at some later time, write in block 50, it is efficient to have the tape move toward block 50 in the interim while the program continues to run:

| Memory Address | Memory Buffer | | Effect |
|---|---|---|---|
| → 100 | RDC i | 0720 | C(block 700) → C(quarter 3); tape moves forward. |
| 101 | 3\|700 | 3700 | |
| 102 | MTB i | 0723 | C(103)-next BN → C(ACC); tape reverses and moves backward toward block 50. |
| 103 | 50 | 0050 | Tape continues to move backward while program continues. |
| ⋮ | ⋮ | ⋮ | |
| 300 | WRI | 0706 | C(quarter 0)→ C(block 50); tape stops. |
| 301 | 50 | 0050 | |

In this example it would be inefficient to stop the tape (i = 0) with the RDC instruction at location 100 or to let it continue to move forward until block 50 is called for. Although the number left in the accumulator after executing the MTB at location 102 may not be of interest, the MTB does reverse the tape. Then, when block 50 is called for, the delay in finding it will not be so long.

## 2-15.4 Tape Format

Certain other facts about the tape format should be mentioned. Other special words on the tape are shown:




At each end of the tape is an area called end zone which provides physical protection for the rest of the tape. When a tape which has been left moving as the result of executing a tape instruction with i = 1 reaches an end zone, the tape stops automatically. (This prevents the tape from being pulled off the reel.) Words marked C and G above do not generally concern the programmer except insofar as

IBZ

they affect tape timing. The computer uses words marked C to insure that the tape writers are turned off following a write instruction. Words marked G, called guard words, protect the forward and backward block numbers when the write current is turned on and off.

Inter-block zones are spaces between block areas which can be sensed by the skip class instruction, IBZ i, when either tape is moving either forward or backward. The purpose of such sensing is to make programmed block searching more efficient. For example, suppose that somewhere in a program block 500 must be read into quarter 2; assume it does not matter when as long it is before the program gets to the instructions beginning at location 650. The following illustration uses a subroutine to check the position of the tape and execute the read instruction if the tape is within 2 blocks of block 500. If the tape is not in an inter-block zone, the main program continues without having to wait for a block number to appear. For purposes of simplicity, assume that the tape (on unit 0) is moving. The program begins at location 400 and the subroutine at location 20.

Note that the following example works only if the tape is stopped by the RDC instruction in register 32. If the tape is not stopped here, subsequent jumps to the subroutine may continue to find the tape at an inter-block zone (locations 20-22) and block 500 may be read repeatedly. The test with the APO instruction at location 646, which signifies if the transfer has been made or not, is necessary to guarantee that the transfer will occur before location 650. At this point, if the transfer has not been made, the JMP 32 at location 647 will be executed.

| Memory Address | Memory Buffer | | Effect |
|---|---|---|---|
| 20 | IBZ | 0453 | Enter subroutine and sense tape position. |
| 21 | JMP 0 | 6000 | Return if tape is not at an inter-block zone. |
| 22 | MTB i | 0723 | If it is, subtract BN or $\overleftarrow{BN}$ from 500. Tape continues to move toward block 500. |
| 23 | 500 | 0500 | |
| 24 | APO | 0451 | Is result positive? |
| 25 | COM | 0017 | If negative, complement it. |
| 26 | ADA i | 1120 | Add −2 to see if tape is within 2 blocks of block 500. |
| 27 | −2 | 7775 | |
| 30 | APO i | 0471 | Is result positive? |
| 31 | JMP 0 | 6000 | If result is positive, return to main program. |
| 32 | RDC | 0700 | If negative, tape is within 2 blocks of block 500. Make the transfer and stop the tape. |
| 33 | 2 500 | 2500 | |
| 34 | STC 645 | 4645 | Store the transfer check = −0 in location 645 to indicate transfer has been made, and return. |
| 35 | JMP 0 | 6000 | |
| ⋮ | ⋮ | ⋮ | |
| → 400 | CLR | 0011 | Store positive 0 in location 645 to indicate transfer has not been made. |
| 401 | STC 645 | 4645 | |
| 402 | JMP 20 | 6020 | Jump to subroutine at these points; return to p + 1 and continue with main program. |
| ⋮ | ↓ | ↓ | |
| 500 | JMP 20 | 6020 | |
| ⋮ | ↓ | ↓ | |
| 600 | JMP 20 | 6020 | |
| ⋮ | ↓ | ↓ | |
| 644 | LDA i | 1020 | Put test number (either 0000 or 7777) into accumulator. |
| 645 | [-] | [-] | |
| 646 | APO i | 0471 | Skip to location 650 if the transfer has been made; (C(ACC) = 7777). |
| 647 | JMP 32 | 6032 | If not, jump to subroutine to make transfer, and return to location 650. |
| 650 | | | |
| ⋮ | ⋮ | ⋮ | |

Example 33 Block Search Subroutine

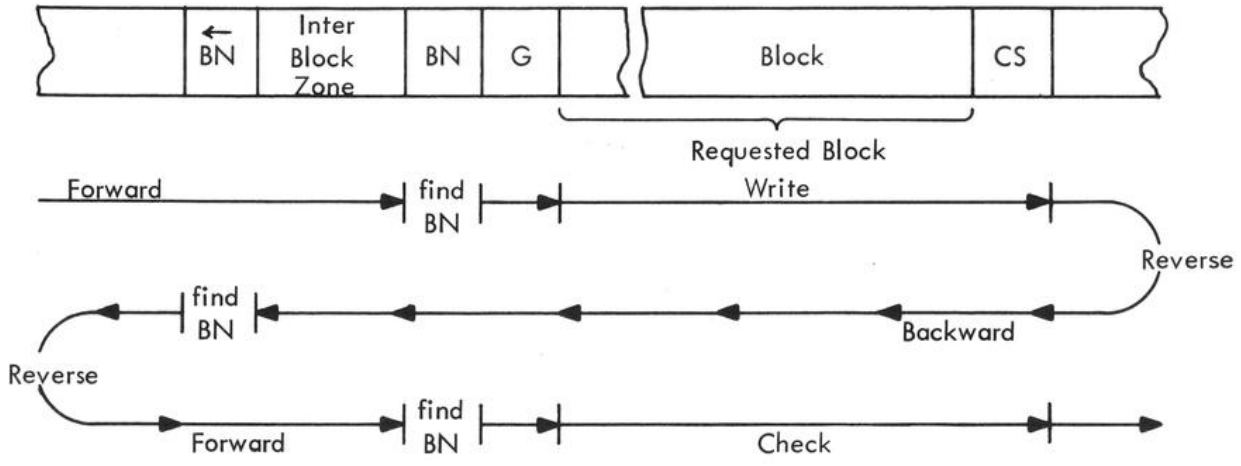
## 2-15.5 Tape Motion Timing

When a tape is moving at a rate of 60 ips, it takes approximately 43 msec to move from one forward block number to the next, or 160 μsec per word. The following table summarizes some of the timing factors:

| LINC TAPE MOTION TIME | |
|---|---|
| START (from no motion to 60 ips) | ~0.1 sec |
| STOP (from 60 ips to no motion) | ~0.3 sec |
| REVERSE DIRECTION (from 60 ips to 60 ips in opposite direction) | ~0.1 sec |
| CHANGE UNIT (from no motion to 60 ips on new unit) | ~0.1 sec |
| BN to BN (at 60 ips) | ~ 43 msec |
| END ZONE to END ZONE (at 60 ips) | ~23 sec |

Some methods of using the tape instructions efficiently become obvious from the above table. Generally speaking, tape instructions should be organized around a minimum number of stops and a minimum amount of tape travel time. When dealing with only one tape unit, it is usually efficient to use consecutive or nearly consecutive blocks in order to reduce the travel time between blocks.

It is also efficient to request lower-numbered blocks before higher-numbered blocks, avoiding unnecessary reversals. The write-and-check instruction, requiring two reversals, is thus costly. It first must find and write in the block in the forward direction; the tape must reverse and go backward until it is below the block, and then reverse a second time and go forward to find and check the block:



Because of these reversals it is sometimes more efficient to use two tape instructions, WRI followed by CHK, than to use WRC. This is true, for example, when more than one block must be written and checked. For example, write quarters 1, 2, and 3 in blocks 100, 101, and 102, and check the transfers: using WRC, this would take a minimum of six reversals. The following sequence requires a minimum of two reversals:

| Memory Address | Memory Buffer | | Effect |
|---|---|---|---|
| → 20 | →LDA | 1000 | Put the BN of the first block to be checked in register 32. |
| 21 | 24 | 0024 | |
| 22 | STC 32 | 4032 | |
| 23 | WRI i | 0726 | Write 3 consecutive blocks on the tape on unit 0 and leave the tape moving forward after each transfer. |
| 24 | 1\|100 | 1100 | |
| 25 | WRI i | 0726 | |
| 26 | 2\|101 | 2101 | |
| 27 | WRI i | 0726 | |
| 30 | 3\|102 | 3102 | |
| 31 | →CHK i | 0720 | Check the blocks, beginning with block 100. |
| 32 | [BN] | [-] | |
| 33 | SAE i | 1460 | If a block does not check, repeat entire process. |
| 34 | 7777 | 7777 | |
| 35 | JMP 20 | 6020 | |
| 36 | LDA i← | 1020 | Add 1 to the BN in register 32. If the result ≠ 1 103, not all have been checked. Return and check the next block. |
| 37 | 1 | 0001 | |
| 40 | ADM | 1140 | |
| 41 | 32 | 0032 | |
| 42 | SAE i | 1460 | |
| 43 | 1\|103 | 1103 | |
| 44 | JMP 31 | 6031 | |
| 45 | MTB← | 0703 | When all have checked, execute move-toward-block to stop the tape, and halt. |
| 46 | 0 | 0000 | |
| 47 | HLT | 0000 | |

Example 34 Write and Check with Fewest Reversals

In this example the two reversals will occur the first time the CHK instruction at location 31 is executed. Other reversals may be necessary when the computer initially searches for block 100, and when a block does not check, but careful handling of the tape instructions can re uce some of these delays. It should be noted that there are 9 words on the tape between any CS and the next BN in the forward direction. When the tape is moving at speed, it takes 1440 μsec to move over these 9 words. Thus the program has time to execute several instructions between consecutive blocks, i.e., before the next BN appears. In the above example, then, there is no danger that the next block will be passed while the instructions at locations 33-44 are being executed.

# CHAPTER 3
# GUIDE

## 3-1 GENERAL

GUIDE is a system of routines which controls a file of binary programs stored on LINC-tape. By using the keyboard, an operator may obtain from LINC-tape any program in the file by its 6-character name, cause it to be read into the computer memory, and then execute it as a program. Using GUIDE, an operator may put a binary program located anywhere on either tape on the tape drives into the binary file on either drive. It is also possible to remove from a file a program which is no longer desired. GUIDE will, upon command, display to the operator an index of the binary programs currently in its file so that the operator may determine if a desired program is on the tape. GUIDE also provides direct communication with the LAP system. GUIDE occupies blocks 400 to 477 on the tape. Blocks 410 to 477 are used for the storage of user's programs.

## 3-2 GENERAL OPERATING PROCEDURE

To set the basic GUIDE system into memory:

1. Mount tape with GUIDE system on it on tape unit 0
2. Set 0700 into the LEFT SWITCHES
3. Set 3400 into the RIGHT SWITCHES
4. Operate DO
5. After tape stops moving, operate START RS

The basic GUIDE system is now in memory and on the display the words "EXECUTE THE PROGRAM ???????" are presented.

## 3-3 BASIC SYSTEM COMMANDS

a. INDIS - This is a command to GUIDE to display to the operator the index of programs in the file area. The following information is displayed: 6-character title of each program, the first block number used to store the program, the number of blocks (or quarters) the program occupies, and the starting address of the program. Only a portion of the index will be displayed at any given time. To advance to the next higher portion (higher block numbers), type F on keyboard. To move back to a lower portion (lower block numbers), type B on keyboard. Typing EOL (1 key on keyboard) returns GUIDE to "EXECUTE..." display.

b. REWIND - This is a command to GUIDE to rewind the tape on the specified drive. REWIND or REWIND 0 rewind tape drive 0 and the computer halts. REWIND 1 causes tape drive 1 to rewind and returns to the "EXECUTE..." display.

c. LAPGO - This command causes GUIDE to read the LAP system into memory and start LAP with a cleared working area.

d. LAPRTN - This command causes GUIDE to read the LAP system into memory and start LAP; the working area is the same as when LAP was last used.

e. CAST - Use of this command causes GUIDE to copy onto the tape on tape drive 1 the LAP and GUIDE systems from tape drive 0. The tape on unit 1 must, however, have the mark and timing tracks already written on it.

f. FILEBI - Use of this command allows the operator to file any binary program from either tape onto the other tape so that the program may be accessed by GUIDE and placed in memory. After the program has been filed, its statistics are entered in the index of binary programs, displayed by INDIS, for reference.

g. DELETE - When this command is used, GUIDE deletes a specified program from the binary file area.

## 3-4 USE OF BASIC COMMANDS

The following assumes the basic GUIDE system is in memory and running, and that the presentation "EXECUTE..." is on the display. Information typed to GUIDE is normally terminated with EOL (1 key).

a. INDIS

1. Type INDIS (EOL).
2. Type F to advance display to next four entries in index.
3. Type B to retreat display to last four entries in index.
4. Type EOL to return to basic display.

b. REWIND

1. Type REWIND (EOL), REWIND 0 (EOL), or REWIND 1 (EOL) to rewind specified unit.
2. If unit 1 is rewound, basic GUIDE display is presented.
3. If unit 0 is rewound, LINC halts and gong chimes.

c. LAPGO

Type LAPGO (EOL). LAP is taken from unit 0 and uses the working area of unit 0.

d. LAPRTN

Type LAPRTN (EOL). LAP is taken from unit 0, and uses the working area of unit 0.

e. CAST

1. Type CAST (EOL).
2. Mount a marked tape on unit 1.
3. Type 0 (EOL) or 1 (EOL) in answer to question asked on display. Question concerns the index on unit 1.

f. FILEBI

1. Type FILEBI (EOL).
2. Type origin of binary program, either name or block number of first block occupied by program, then (EOL).
3. Type origin reel, then (EOL).
4. Type destination reel, then (EOL).
5. If block number was typed, assign a 6 character name to program, type it in, then type (EOL).
6. Type starting address (EOL).
7. Type number of blocks of tape (or quarters of LINC memory) program occupies (EOL).

g. DELETE

1. Type DELETE (EOL).
2. Type name of program, then (EOL).
3. Type unit number where program is located, then (EOL).

## 3-5 LOADING A USER'S PROGRAM INTO MEMORY

a. Get basic GUIDE system into memory and running.

b. Type INDIS (EOL) to search index to determine if desired program is on the tape. Return to "EXECUTE..." display after title has been found in index.

c. Type 6-character title, then (EOL). GUIDE loads program into memory and starts it at the address indicated in the index as the program's starting address.

# CHAPTER 4
# LAP 4

## 4-1 GENERAL

LAP4 (LINC Assembly Program 4) is a system which aids the programmer in creation and in manipulation of the symbolic text of source programs, and converts the symbolic text into binary so that it may be executed as a program. The symbolic text is often called a manuscript.

LAP4 occupies blocks 270-377 on a reel of LINC-tape. Blocks 270-327 are used for the storage of the system itself, and blocks 330-377 are used as working area for LAP4. It is here that the manuscript is written during generation of symbolic text as well as where the binary of the converted program is temporarily stored (the binary should be filed by GUIDE for permanent storage).

The LAP4 system is operated from the keyboard and it is from here that the symbolic text is inputted and commands are given to the system.

## 4-2 GENERAL OPERATING PROCEDURE

To get the LAP4 system into memory:

1. Read the basic GUIDE system into memory and start it running
2. Type LAPGO (EOL) or LAPRTN (EOL)

The LAP4 system is then read into memory. LAPGO causes LAP4 to be read into memory with the working area cleared of all manuscript material (no symbolic text) and the number 0001 displayed on the scope to indicate the first line of symbolic text. LAPRTN causes LAP4 to be read into memory with the working area in the same condition as when LAP4 was last used (manuscript is still intact if LAP4 was exited properly) and the first free line number displayed on the scope. Normally only the current line number and its contents are displayed.

## 4-3 BASIC SYSTEM (META) COMMANDS

a. REMOVE - Use of this command causes LAP4 to remove from the working area the specified line (or lines) of text thereby deleting it (or them). All succeeding lines are renumbered to maintain continuity of line numbers.

b. INSERT - This command causes all text typed after the issuing of this command, until the END command is given, to be inserted before the specified line. All succeeding lines are renumbered to maintain continuity of line numbers.

c. PACK - The insertion and deletion of lines of text in the manuscript working area causes physical gaps in the text stored; thus the text is stored inefficiently. To remove these gaps, the command PACK is given which causes the text to be physically repositioned, fills the gaps, and makes room for more lines of text to be stored in the working area.

d. DISPLAY - Use of this command causes consecutive lines of text to be displayed on the scope. The operator may display from 1 to $100_8$ lines of text at one time; however, no editing can take place while the display command is being executed. Typing F advances the display to the next higher (numerical) set of lines of text; typing B retreats the display to the next lower set of lines of text. Typing F when at the highest set of lines or typing B when at the lowest set of lines has no effect. Typing (EOL) causes LAP 4 to return to the normal input mode.

e. SAVE MANUSCRIPT - This is a command to LAP4 to write the manuscript in the LAP4 working area into the specified blocks anywhere on either reel of LINC-tape. An unpacked manuscript is packed by LAP 4 before it is saved.

f. ADD MANUSCRIPT - This command adds a manuscript to the manuscript currently in the LAP4 working area. The added manuscript may be located anywhere on either tape as long as its location and size is known.

g. CONVERT - The convert command causes LAP 4 to convert the manuscript in the LAP4 working area into its binary equivalent. The resultant binary is stored in blocks 330-333. Information for quarter 0 is in block 330, quarter 1 in block 331, etc. If multiply-defined or undefined symbolic addresses (tags) are encountered, LAP4 displays these on the scope to inform the operator that such errors exist.

h. CONVERT MANUSCRIPTS - To convert manuscripts not in the working area, give this command. Up to eight manuscripts may be converted by this command; however, all manuscripts must be on tape unit 0. The result of the conversion will be in blocks 330-333 on unit 0. The manuscript in the working area is neither affected or converted by this command.

i. COPY - This command allows the operator to copy any number of blocks of information from anywhere on tape to anywhere on tape. The information moved may be manuscripts, binary programs, or anything else written on tape. Care must be

taken when copying overlapping blocks to prevent destruction of data since only three blocks will be moved at a time. More than three blocks can be specified for any copy, but copy moves it in three block segments.

j. START LAP 4 - To restart LAP 4 with the working area cleared, give this command.

k. START GUIDE - To transfer control of the computer to GUIDE to do something with GUIDE, this command saves the manuscript in the working area and then reads the basic GUIDE system into memory and starts it running. The operator can return to LAP 4 from GUIDE with the saved manuscript in the working area by executing the GUIDE command LAPRTN.

l. MANUSCRIPT CONTROL - This command allows the user to manipulate manuscripts into and out of manuscript files and the LAP 4 working area. After giving this command, the operator is given his choice of the several functions which LAP 4 can do for him. He can examine the index of any of the eight manuscript files on each tape unit so that he may determine if a desired manuscript is filed there. He may add to the working area any manuscript in any file, or put into any file the manuscript currently in the LAP 4 working area. He may remove from any file manuscripts which he no longer wants filed. He may clear a file of all manuscripts. After doing all of the above the operator may then return to normal input with the LAP 4 working area intact, or, if he prefers, may transfer control of the computer to GUIDE.

## 4-4 USE OF BASIC (META) COMMANDS

The following assumes that the current line is displayed on the scope and that it contains no text. If anything else is displayed, it is necessary to return to the normal input mode by the appropriate manner before the command is given. Commands issued to LAP 4 are terminated by META, as compared to lines of text which are terminated by EOL. Both are 1 key.

a. REMOVE

1. Type RE LN, n (META) or RE LN – LN + n (META) where LN is a line number (octal) and n is the number of lines (octal) to be deleted.

2. If the RE LN – LN + n format is used, LAP4 will remove from line LN to LN + n – 1 (n lines of text). The higher line number will not be removed.

3. To remove all lines after a given line, type the first line number to be removed and then any value of n equal to or greater than the number of lines to be removed.

4. Attempts to remove non-existent lines causes LAP 4 to respond with NO.

5. LAP 4 returns to normal input mode after the specified lines have been removed.

b. INSERT

1. Type IN LN (META) where LN represents the number of the line before which the insertion should be made. All text typed is then inserted before the specified line until the termination command is given, at which time LAP 4 returns to the normal input mode.

2. Type EN (META) on a separate line to terminate the insertion of text. LAP 4 then returns to the normal input mode with a new line number displayed on the scope.

3. LAP 4 allows the operator to insert up to $110_8$ lines on one INSERT command. Attempts to exceed this cause LAP 4 to automatically terminate that insertion. However, an INSERT command can be given again immediately if desired.

4. LAP 4 responds with NO if certain operator errors are made during insertion, such as: inserting before a non-existent line, or inserting no lines after giving the INSERT command. LAP 4 questions META commands other than EN (META) (end) issued during an insertion.

c. PACK

Type PA (META). LAP 4 then packs the manuscript in the working area, filling in any physical gaps which may exist in the manuscript. LAP 4 returns to normal input mode when packing is completed.

d. DISPLAY

1. Type DI LN, S (META), where LN is the number of the first line to be displayed and S is the size (number of lines) of the display. The text is displayed S lines at a time, the first display starting with line number LN.

2. If LN is not specified, the number 0001 is assumed. If S is not specified, the number $10_8$ is assumed. Thus, typing D1 (META) causes 10 lines starting with line 0001 to be displayed.

3. Type F to advance display to next higher S lines segment.

4. Type B to retreat display to next lower S lines segment.

5. Type EOL to return to normal input mode.

6. Lines of text are displayed along with their line numbers.

7. LAP 4 packs an unpacked manuscript before displaying it.

8. Type NS to change size of display (number of lines) once the display of text has begun.

9. Type LN L to change first line number of display once the display of text has begun.

e. SAVE MANUSCRIPT

1. Type SM (META). LAP 4 then displays to the operator the number of blocks of text to be moved and asks for the destination of the text.

2. Type the destination block number, terminating with (EOL).

3. Type unit number, terminating with (EOL).

4. LAP 4 packs an unpacked manuscript before saving it.

5. LAP 4 returns to normal input mode when this command is completed.

f. ADD MANUSCRIPT

1. Type AM BN, UN (META) where BN represents the first block occupied by the desired manuscript and UN indicates the unit (number of tape drive) where the manuscript is located.

2. LAP 4 returns to normal input mode after executing this command.

3. It is advisable to pack the working area after adding manuscripts.

4. LAP 4 responds with a NO if there is no manuscript at the indicated location. Typing any key allows LAP4 to recover.

g. CONVERT

1. Type CV (META). LAP 4 converts the manuscript in the working area of LAP4 and stores the binary in blocks 330-333. Manuscripts can be no longer than $3777_8$ lines.

2. If no errors occur. LAP 4 returns to the normal input mode.

3. The LAP4 system is able to detect certain types of errors in the manuscript, mainly multiply-defined symbols and undefined symbols. If these errors occur, LAP4 displays the error(s) and the symbols which caused them. Typing F advances the error display to the next higher set, while typing B retreats the error display to the next lower set. Multiply-defined symbols are displayed separately from undefined symbols. Typing (EOL) returns LAP 4 to the normal input mode.

h. CONVERT MANUSCRIPTS

1. Type CM (META)

2. Type in the block numbers of the first block of each manuscript to be converted. Separate each entry with a space. Terminate the string with (EOL). To delete an entry type (DEL) before the (EOL).

3. All manuscripts must be on unit 0.

4. No single manuscript may be longer than $2000_8$ lines.

5. Manuscripts are converted in the order typed.

6. Up to eight manuscripts may be converted with each CM.

7. Binary version of manuscripts is in blocks 330-333 as with CV.

8. Multiply-defined symbols and undefined symbols are displayed the same way as with CV.

9. If no manuscript is found at the indicated block, LAP4 responds with NO. Striking any key returns LAP4 to the normal input routine.

i. COPY

1. Type CP (META).

2. Type number of blocks to copy, then (EOL).

3. Type first block number of origin, then (EOL).

4. Type unit number of origin, then (EOL).

5. Type first block number of destination, then (EOL).

6. Type unit number of destination, then (EOL).

7. LAP4 then copies the specified blocks, 3 at a time. Care must be taken when origin and destination blocks overlap.

8. LAP4 returns to normal input mode when copying is completed.

9. No more than $777_8$ blocks may be copied.

10. Individual entries (b), c), d), e), f)) may be deleted by typing (DEL) before final (EOL).

11. If an illegal (non-numeric) entry is made, LAP4 asks all its questions again.

j. START LAP4

Type LA (META). LAP4 is restarted with the working area clear and the number 0001 (line 0001) displayed. LAP4 will be in the normal input mode.

k. START GUIDE

Type GU (META). Control of the computer is transferred to the GUIDE system. The manuscript in the working area of LAP4 is saved for restoration by the GUIDE command LAPRTN.

## 1. MANUSCRIPT CONTROL

NOTE: For most of the options presented to the operator, use of this META command causes LAP4 to examine sense switch 0. If SSW0 is 0, LAP4 performs the specified function on file 2, unit 0, which is commonly called the standard file. If SSW0 is 1, LAP 4 performs the specified operation on the specified file on either tape unit. The files are numbered 0 to 7 and start at blocks 000, 100, 200, etc., respectively. All files except file 2 are 100 blocks long. File 2 is 70 blocks long since LAP4 begins at block 270.

1. Type MC (META). LAP4 then displays a series of options from which the operator may choose the function he desires LAP4 to perform.

2. Type the desired option number, then (EOL).

3. Type in the answers to each question LAP4 asks, terminating each with (EOL). Typical questions concern manuscript name, file number, unit number, block number. The final (EOL) causes LAP 4 to execute the function.

4. Except for going to LAP4 or GUIDE, after function is complete, LAP 4 returns to MC option display.

5. If a manuscript file is being displayed, typing F advances the display, typing B retreats the display, and typing EOL returns LAP4 to the MC option display.

## 4-5 LAP4 LANGUAGE

### a. CHARACTER SET

1. All 26 upper case letters of the alphabet.

2. The octal number set (0-7).

3. The following special or punctuation characters

| | | | |
|---|---|---|---|
| a. | ⊟ | (origin) | specifies an origin |
| b. | # | (number sign) | specifies a symbolic tag |
| c. | i | (lower case I) | specifies bit 7 on a 1 |
| d. | u | (lower case U) | specifies bit 8 on a 1 |
| e. | \| | (vertical bar) | separates QN and BN in magnetic tape instructions |

| | | |
|---|---|---|
| f. p | (lower case P) | specifies the current location |
| g. + | (plus) | add values of syllables in 1's complement |
| h. − | (minus) | subtract values of syllables in 1's complement |
| i. = | (equal) | parameter assignment |
| j. [ | (left bracket) | initiates a comment |
| k. (EOL) | | terminate statement |
| l. (CASE) | | change case of keyboard for upper case characters |
| m. (META) | | terminates command to LAP4 |
| n. (SPACE) | | separation character for ease of reading text |
| o. (DEL) | | delete all information to previous line terminator |

b. PROGRAMMING RULES

1. The following elements must be alone on a line of text.

   a. Origins

   b. Comments

   c. Parameter assignments (use of =)

2. Tags (a symbol used to represent a memory location)

   a. Must begin with #.

   b. Must be two characters of the format number-letter (e.g., #4E).
   Only octal numbers allowed, and capital letters.

   c. No tag delimiter is required.

   d. No space may occur in the tag.

   e. Tags defined by parameter assignment may not be used with a #.

3. Spaces

   a. Not permitted on origin line except before the character ⊟.

   b. Not permitted before a [.

   c. Not permitted before a #.

   d. Not permitted anywhere on a parameter assignment.

   e. Not permitted in the middle of the digits of a number.

   f. Not permitted in the middle of an address calculation.

   g. Not permitted in the middle of a symbolic mnemonic.

   h. May be used to separate tag, operation, index, address, vertical bar fields of a line.

   i. Not required anywhere on a line.

4. Symbolic Mnemonics

   a. Only the defined symbolic mnemonics may be used as instructions.

5. Parameter Assignments

   a. Defined by equal sign (=).

   b. Number-letter combination must appear on left of =.

   c. Octal assignment must be on right of =.

   d. Octal assignment may not be signed.

   e. Must appear alone on a line.

   f. No spaces allowed on the line.

6. Address Calculation

   a. No spaces permitted in the address calculation.

   b. Symbolic or relative addressing allowed with any combination of number-letter combinations or p or octal numbers (e.g., JMP p-1, ADD 500, ADA i 3Z, STC 4P-1, 6E + 3, RDC i u).

   c. All undefined number-letter combinations are assigned the value 0000.

   d. All multiply-defined number-letter combinations are assigned the last value specified.

7. Example of a symbolic program and its octal equivalent.

```
                0001    [THIS PROGRAM
                0002    [DISPLAYS THE
                0003    [CONTENTS OF
                0004    [THE RIGHT
                0005    [SWITCHES AS
                0006    [A DECIMAL
                0007    [NUMBER AT
                0010    [THE CENTER
                0011    [OF THE SCOPE
                0012    ⊟354
                0013    [TABLE OF
                0014    [CONSTANTS
                0015    [FOR DISPLAY
                0016    [OF DIGITS
0354            0017    #5E 4136        4136
0355            0020        3641        3641
0356            0021        2101        2101
0357            0022        0177        0177
0360            0023        4523        4523
0361            0024        2151        2151
0362            0025        4122        4122
0363            0026        2651        2651
0364            0027        2414        2414
0365            0030        0477        0477
0366            0031        5172        5172
0367            0032        4651        4651
```

```
0370    0033          1506              1506
0371    0034          4225              4225
0372    0035          4443              4443
0373    0036          6050              6050
0374    0037          5126              5126
0375    0040          2651              2651
0376    0041          5120              5120
0377    0042          3651              3651
        0043    [THESE
        0044    [INSTRUCTIONS
        0045    [SET INDEX
        0046    [REGISTERS
        0047    [FOR 4
        0050    [CHARACTERS,
        0051    [HORIZONTAL
        0052    [COORDINATE
        0053    [OF 320, A
        0054    [POINTER TO
        0055    [6E, AND A
        0056    [COUNTER TO 0
0400    0057    #5H SET i 2P      0062
0401    0060        -4            7773
0402    0061        SET i 1P      0061
0403    0062        320           0320
0404    0063        SET i 3P      0063
0405    0064        6E            0453
0406    0065        SET i 4P      0064
0407    0066        0             0000
0410    0067        RSW           0516
0411    0070        STC 6Z        4452
        0071    [THESE
        0072    [INSTRUCTIONS
        0073    [DETERMINE
        0074    [HOW MANY
        0075    [1000, 100
        0076    [10, 1,
        0077    [DECIMAL,
        0100    [ARE IN THE
        0101    [NUMBER, ONE
        0102    [AT A TIME
0412    0103    #6B CLR           0011
0413    0104        ADA 3P        1103
0414    0105        COM           0017
0415    0106        ADA i         1120
0416    0107        1             0001
0417    0110        LAM           1200
0420    0111        6Z            0452
0421    0112        XSK i 4P      0224
0422    0113        LZE           0452
0423    0114        JMP P-11      6412
0424    0115        LDA 3P        1003
0425    0116        LAM           1200
0426    0117        6Z            0452
        0120    [THESE
        0121    [INSTRUCTIONS
        0122    [SET UP THE
        0123    [TABLE ENTRY
        0124    [POINT TO
        0125    [DISPLAY THE
        0126    [CORRECT
        0127    [DIGIT
```

```
0427  0130        LDA i              1020
0430  0131        -1                 7776
0431  0132  #6C ADD 4P               2004
0432  0133        ROL 1              0241
0433  0134        ADA i              1120
0434  0135        5E                 0354
0435  0136        STC 5P             4005
      0137  [THESE
      0140  [INSTRUCTIONS
      0141  [DISPLAY THE
      0142  [DIGIT AT 0 V
      0143  [AND 320+ H
0436  0144        DSC 5P             1745
0437  0145        DSC i 5P           1765
      0146  [THESE
      0147  [INSTRUCTIONS
      0150  [SET UP FOR
      0151  [THE NEXT
      0152  [DIGIT
      0153  [DISPLAY
0440  0154        LDA i              1020
0441  0155        4                  0004
0442  0156        ADM                1140
0443  0157        1P                 0001
0444  0160        SET i 4P           0064
0445  0161        0                  0000
0446  0162        XSK i 3P           0223
      0163  [THIS XSK
      0164  [TESTS TO SEE
      0165  [IF 4 DIGITS
      0166  [WERE
      0167  [DISPLAYED
0447  0170        XSK i 2P           0222
0450  0171        JMP 6B             6412
0451  0172        JMP 5H             6400
0452  0173  #6Z 0                   0000
      0174  [TABLE OF
      0175  [OCTAL VALUES
      0176  [OF 1000,
      0177  [100, 10, 1
0453  0200  #6E 1750                1750
0454  0201        144                0144
0455  0202        12                 0012
0456  0203        1                  0001
      0204  [DEFINITION
      0205  [OF
      0206  [PARAMETERS
      0207        1P=1
      0210        2P=2
      0211        3P=3
      0212        4P=4
      0213        5P=5
```

# APPENDIX 1
# GLOSSARY

## A1-1 WORD DEFINITIONS

**Address** — A unique 11-bit binary number assigned to each 12-bit binary word (core storage location in LINC-8 memory; allowable range for addresses is $(0000\text{-}3777)_8$.

**Assembler** — A program which translates program statements in a symbolic language closely resembling machine language into machine language.

**β** — Refers to bits 8-11 of certain LINC-8 instructions which may reference the β-registers (addresses 0001-0017).

**BN** — Abbreviation for block number; see Block.

**Binary** — Used to refer to the aggregate of the machine language instructions generated by the conversion (assembly) of a manuscript by LAP4.

**Block** — A numbered section of a marked LINC tape capable of retaining $400_8$ 12-bit binary words; blocks are numbered consecutively from $(000\text{-}777)_8$.

**Case** — The upper leftmost key on the LINC keyboard, used in input to this LINC-8 utility system to cause the system to treat the next struck character as upper case.

**Comment** — In LAP4, an MS line beginning with the comment character (↑), used by the programmer to illustrate his MS, but ignored by LAP4 during conversion.

GLOSSARY (continued)

| | |
|---|---|
| Compiler | A program which translates program statements in a symbolic language closely resembling English or mathematics into machine language. |
| Control Block | See MS control block or file control block. |
| Control Console | The LINC-8 panel which contains the toggle switches, pushbuttons, levers, rotary switches, and indicator lights; operation of the LINC-8 utility system is initiated via the control console. |
| Conversion | The assembly process whereby LAP4 translates a program written in a symbolic language into machine language; MS is converted into binary. |
| Core Storage | The LINC-8 memory. |
| Delete | To remove a line of MS or an answer to a displayed question in this LINC-8 Utility system, use the del key. |
| EOL | Abbreviation for end of line; the key used to indicate to the utility system the end of a MS line or the end of an answer to a displayed question. |
| Equality | An MS line in LAP4 used to assign an absolute numerical value to a tag. |
| File | Either the file of binary programs maintained by GUIDE or a file of MS created and maintained under the control of the MC meta command in LAP4. |

## GLOSSARY (continued)

| | |
|---|---|
| File Control Block | The first block in an MS file; used in LAP4 by the MC meta command to retain titles, block numbers, etc., of filed MS. |
| Full-size Character | A character displayed on the scope via a 4 x 6 grid pattern, the grid spacing being 4 units between points. |
| GUIDE | The GUIDE to binary programs; one of the two systems which comprise this LINC-8 utility system; used for the filing and execution of binary programs. |
| Half-size Character | A character displayed on the scope via a 4 x 6 point grid pattern, the grid spacing being 2 units between points. |
| i | The i-bit; bit 7 of certain LINC-8 instructions. |
| Index | Either the index to the GUIDE file of binary programs or the index to an MS file. |
| Keyboard Codes | The 6-bit codes for the characters on the LINC keyboard; generated in the accumulator upon the execution of a KBD instruction after a key has been struck. |
| LAP4 | LINC-8 Assembly Program 4; one of the two systems which comprise this LINC-8 utility system; used for the creation, conversion, and filing of MS. |
| LN | Abbreviation for line number; see line. |
| Line | A string of characters (keyboard codes) in a LAP4 manuscript, last character of which is EOL (or META). |
| MS | Abbreviation for manuscript; see manuscript. |

GLOSSARY (continued)

| | |
|---|---|
| MS Control Block | The first block of every LAP4 MS, created during regular input of an MS by LAP4 in the working area; contains information about number of lines, number of tape blocks occupied, etc. |
| MS Line | A line retained by LAP4 as a permanent part of an MS; i.e., program lines, equalities, origins and comments; as opposed to meta commands. |
| Machine Language | The directly machine-interpretable, i.e., binary, form of the LINC-8 instructions. |
| Manuscript | A series of one or more program lines, equalities, origins, and comments typed into the LAP4 system and stored on tape. |
| Marking | The process whereby a virgin tape is readied for use on the LINC-8. |
| Meta Command | A line not retained by LAP4 as part of an MS; a direct, immediately executed command to LAP4. |
| Mnemonics | Three-character acronyms or abbreviations for the LINC-8 instructions. |
| Object Program | The binary generated by conversion of an MS. |
| Order Code | The LINC-8 instruction repertoire. |
| Origin | An MS line used to locate sections of a program in core storage at absolute addresses. |

GLOSSARY (continued)

| | |
|---|---|
| p | Keyboard character; interpreted by LAP4, on a program line, as referring to the present location; i.e., the address of the location in which the binary for the current line will reside. |
| Packing | The process whereby gaps in MS left by the operation of the meta commands RE, IN, AM, or MC are removed. |
| Palimpsest | A parchment which has been re-used, the earlier writing having been erased. |
| Pass | In LAP4, a scan of an MS from beginning to end during conversion. |
| Program | A series of instructions to the LINC-8. |
| Program Line | An MS line which will cause binary to be generated; i.e., will occupy a location in core storage upon conversion. |
| Q | Abbreviation for quarter; see quarter. |
| QN | Abbreviation for quarter number; see quarter. |
| Q&A | Abbreviation for the Questions and Answers subroutine, used for displays by the GUIDE system commands, convenience programs, and the LAP4 meta commands CP and MC. |
| Quarter | One fourth of a $1024_{10}$ word LINC-8 memory bank; consists of $400_8$ contiguous 12-bit words. |
| Regular Input | The section of LAP4 which accepts input from the keyboard of MC lines and meta commands. |

## GLOSSARY (continued)

| | |
|---|---|
| S | Abbreviation for size; refers to the number of lines of MS displayed on the scope by the DI meta command. |
| SSW | Abbreviation for SENSE switch. |
| Scope | The standard LINC-8 display scope. |
| Source Program | MS. |
| Subroutine | A program written to perform some special function; may be entered from another program, to which it will return control upon completion of its operation. |
| Symbolic Address | A number, letter combination (tag) used to reference a core location, the absolute value of which is assigned by LAP4 during conversion. |
| Symbolic Operation Code | Mnemonics. |
| System Tape | A tape which contains the LAP4 and GUIDE systems. |
| Tag | A number, letter combination used as a symbolic address by LAP4. |
| Tape Block | See Block. |
| UN | Abbreviation for unit number; see Unit. |
| Unit | LINC-8 tape unit 0 (left) or 1 (right). |

GLOSSARY (continued)

| | |
|---|---|
| Utility System | A programming system for the LINC-8 composed of two communicating systems, LAP4 and GUIDE. |
| Working Area | That section of a system tape used by the LAP4 system for storing MS and the binary converted from MS; occupies blocks 330 and ff. |

## A1-2 SYMBOL DEFINITIONS

### A1-2.1 Registers

| Symbol | Function |
|---|---|
| A | Accumulator |
| B | Memory buffer |
| C | Control |
| L | Link bit |
| P | Program counter |
| R | Output of relays |
| S | Memory address |
| Z | Odd jobs |

## A1-2.2 Other Symbols

| Symbol | Definition |
|---|---|
| $A_j$ | Bit j of register A. |
| $A_{j-k}$ | Bits j-k, inclusive, of A. |
| i | Bit 7 of the instruction word or of the contents of C. |
| u | Bit 8 of the instruction word or of the contents of C. |
| n | Bits 8-11 of the instruction word, when these bits are not used to refer to one of the first 16 memory locations as index registers. |
| $\beta$ | Bits 8-11 of the instruction word, in those instructions which may use these bits to specify the address of an index register. |
| p | The address of the memory location from which the first word of the current instruction was obtained. |
| X | Bits 2-11 of a twelve bit word. |
| $X(\beta)$ | Bits 2-11 of the contents of index register $\beta$. |
| X(p+1) | Bits 2-11 of the contents of the memory location whose address is p+1. |
| h | A bit which is used to specify which half of the operand word is used by a half-word instruction. |
| $h(\beta)$ | Bit 0 of the contents of index register $\beta$. |
| h(p+1) | Bit 0 of the contents of the memory location whose address is p+1. |
| $X(\beta)_{ndx}$ | 1 + $X(\beta)$, using 10-bit 2's complement addition. |
| $X(\beta)_{hndx}$ | $X(\beta)_{hndx} = X(\beta)$ if $h(\beta) = 0$<br>$X(\beta)_{hndx} = X(\beta)_{ndx}$ if $h(\beta) = 1$. |
| Y | The address of the operand of an instruction, 11 bits in length. |
| Y(p+1) | Bits 1-11 of the contents of the memory location whose address is p+1. |
| $Y(\beta)$ | Bits 1-11 of the contents of index register $\beta$. |

# APPENDIX 2
# CHARTS

## A2-1 CHART I CLASSES OF LINC INSTRUCTIONS AND THEIR CODES

### Miscellaneous

| | |
|---|---|
| *HLT | 0000 |
| | 0001 |
| | 0002 |
| | 0003 |
| | 0004 |
| ZTA | 0005 |
| | 0006 |
| | 0007 |
| | 0010 |
| *CLR | 0011 |
| | 0012 |
| (MARK) | 0013 |
| *ATR | 0014 |
| *RTA | 0015 |
| *NOP | 0016 |
| *COM | 0017 |

### Alpha n

| | |
|---|---|
| *SET (i) α | 0040 |
| *SAM (i) n | 0100 |
| *DIS (i) α | 0140 |
| *XSK (i) α | 0200 |

### Shift

| | |
|---|---|
| *ROL (i) n | 0240 |
| *ROR (i) n | 0300 |
| *SCR (i) n | 0340 |

### Skip On Level

| | |
|---|---|
| *SZL (i) n | 0400 |
| | 0401 |
| | 0402 |
| | 0403 |
| | 0404 |
| | 0405 |
| | 0406 |
| | 0407 |
| | 0410 |
| | 0411 |
| | 0412 |
| | 0413 |
| | 0414 |
| *KST (i) | 0415 |
| | 0416 |
| | 0417 |

*These mnemonics are defined to LAP4

### Skip On Condition

| | |
|---|---|
| *SNS (i) n | 0440 |
| | 0441 |
| | 0442 |
| | 0443 |
| | 0444 |
| | 0445 |
| | 0446 |
| | 0447 |
| *AZE (i) | 0450 |
| *APO (i) | 0451 |
| *LZE (i) | 0452 |
| *IBZ (i) | 0453 |
| FLO (i) | 0454 |
| ZZZ (i) | 0455 |
| | 0456 |
| | 0457 |

### Magnetic Tape

| | |
|---|---|
| *RDC (i) (u) | 0700 |
| *RCG (i) (u) | 0701 |
| *RDE (i) (u) | 0702 |
| *MTB (i) (u) | 0703 |
| *WRC (i) (u) | 0704 |
| *WCG (i) (u) | 0705 |
| *WRI (i) (u) | 0706 |
| *CHK (i) (u) | 0707 |

### Execute PDP-8

| | |
|---|---|
| EXC | 0740 |

### Operate

| | |
|---|---|
| *OPR (i) n | 0500 |
| | 0501 |
| | 0502 |
| | 0503 |
| | 0504 |
| | 0505 |
| | 0506 |
| | 0507 |
| | 0510 |
| | 0511 |
| | 0512 |
| (PDP) | 0513 |
| (TYP) | 0514 |
| *KBD (i) | 0515 |
| *RSW | 0516 |
| *LSW | 0517 |

### Unused Codes

0540

### Memory Bank Selection

| | |
|---|---|
| LMB | 0600 |
| UMB | 0640 |

### Full Address

| | |
|---|---|
| *ADD X | 2000 |
| *STC X | 4000 |
| *JMP X | 6000 |

*These mnemonics are defined to LAP4

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | Index (or Beta) | | | | |
| *LDA (i) β | 1000 | Half Word | *LDH (i) β | 1300 | | *BCL (i) β | 1540 |
| *STA (i) β | 1040 | | *STH (i) β | 1340 | | *BSE (i) β | 1600 |
| *ADA (i) β | 1100 | | *SHD (i) β | 1400 | | *BCO (i) β | 1640 |
| *ADM (i) β | 1140 | | *SAE (i) β | 1440 | | | 1700 |
| *LAM (i) β | 1200 | | *SRO (i) β | 1500 | | *DSC (i) β | 1740 |
| *MUL (i) β | 1240 | | | | | | |

*These mnemonics are defined to LAP4.

A2-2 CHART II ASR 33 LINC CODE

| ASR 33 | | LINC | | ASR 33 | | LINC | |
|---|---|---|---|---|---|---|---|
| ASCII Code | Symbol | LINC Code | Symbol | ASCII Code | Symbol | LINC Code | Symbol |
| 260 | 0 | 00 | 0 | 304 | D | 27 | D |
| 261 | 1 | 01 | 1 | 304 | E | 30 | E |
| 262 | 2 | 02 | 2 | 306 | F | 31 | F |
| 263 | 3 | 03 | 3 | 307 | G | 32 | G |
| 264 | 4 | 04 | 4 | 310 | H | 33 | H |
| 265 | 5 | 05 | 5 | 311 | I | 34 | I |
| 266 | 6 | 06 | 6 | 312 | J | 35 | J |
| 267 | 7 | 07 | 7 | 313 | K | 36 | K |
| 270 | 8 | 10 | 8 | 314 | L | 37 | L |
| 271 | 9 | 11 | 9 | 315 | M | 40 | M |
| 212/215 | LF/CR | 12 | META/EOL | 316 | N | 41 | N |
| 377 | RUBOUT | 13 | delete | 317 | O | 42 | O |
| 240 | SPACE | 14 | SPACE | 320 | P | 43 | P |
| 275/246 | =/& | 15 | = / i | 321 | Q | 44 | Q |
| 300/247 | @/' | 16 | u/p | 322 | R | 45 | R |
| 254/255 | ,/- | 17 | ,/- | 323 | S | 46 | S |
| 256/253 | ./+ | 20 | ./+ | 324 | T | 47 | T |
| 244/257 | $// | 21 | θ/\| | 325 | U | 50 | U |
| 333/243 | [/# | 22 | [/# | 326 | V | 51 | V |
| 375 | ALTMODE | 23 | CASE | 327 | W | 52 | W |
| 301 | A | 24 | A | 330 | X | 53 | X |
| 302 | B | 25 | B | 331 | Y | 54 | Y |
| 303 | C | 26 | C | 332 | Z | 55 | Z |

## A2-3 CHART III PATTERN WORDS FOR CHARACTER DISPLAY

A table of 24-bit patterns for 4 x 6 display, using the DSC instruction, of all characters on the LINC keyboard. The table is ordered numerically as the characters are coded on the keyboard. Table entries for non-displayable characters are 0.

| | | | | | |
|---|---|---|---|---|---|
| 0 | 4136 | A | 4477 | U | 0177 |
| | 3641 | | 7744 | | 7701 |
| 1 | 2101 | B | 5177 | V | 0176 |
| | 0177 | | 2651 | | 7402 |
| 2 | 4523 | C | 4136 | W | 0677 |
| | 2151 | | 2241 | | 7701 |
| 3 | 4122 | D | 4177 | X | 1463 |
| | 2651 | | 3641 | | 6314 |
| 4 | 2414 | E | 4577 | Y | 0770 |
| | 0477 | | 4145 | | 7007 |
| 5 | 5172 | F | 4477 | Z | 4543 |
| | 0651 | | 4044 | | 6151 |
| 6 | 1506 | G | 4136 | = | 1212 |
| | 4225 | | 2645 | | 1212 |
| 7 | 4443 | H | 1077 | u | 0107 |
| | 6050 | | 7710 | | 0107 |
| 8 | 5126 | I | 7741 | , | 0500 |
| | 2651 | | 0041 | | 0006 |
| 9 | 5120 | J | 4142 | . | 0001 |
| | 3651 | | 4076 | | 0000 |
| EOL | 0000 | K | 1077 | ⊟ | 4577 |
| | 0000 | | 4324 | | 7745 |
| del | 0000 | L | 0177 | [ | 4177 |
| | 0000 | | 0301 | | 0000 |
| SPACE | 0000 | M | 3077 | | |
| | 0000 | | 7730 | | |
| i | 0101 | N | 3077 | | |
| | 0126 | | 7706 | | |
| p | 3700 | O | 4177 | | |
| | 3424 | | 7741 | | |
| - | 0404 | P | 4477 | | |
| | 0404 | | 3044 | | |
| + | 0404 | Q | 4276 | | |
| | 0437 | | 0376 | | |
| \| | 0000 | R | 4477 | | |
| | 0077 | | 3146 | | |
| # | 3614 | S | 5121 | | |
| | 1436 | | 4651 | | |
| CASE | 0000 | T | 4040 | | |
| | 0000 | | 4077 | | |

## A2-4 UTILITY SYSTEM TAPE ALLOCATION

| Block | Allocation |
|---|---|
| 000 - 012 | Loaders |
| 013 - 267 | Available to user (may be used for MS files) |
| 270 - 327 | LAP4 system |
| 330 - 377* | LAP4 working area |
| 400 - 407 | GUIDE system |
| 410 - 477 | GUIDE file area |
| 500 - 777 | Available to user (may be used for MS files) |

## A2-5 GUIDE TAPE ALLOCATION

| Block | Allocation |
|---|---|
| 400 | Input control |
| 401 | Display index (INDIS) |
| 402 | Index of binary program file |
| 403 | Questions and answers subroutine |
| 404 | File a binary program (FILEBI) |
| 405 | Create a system tape (CAST) |
| 406 | File a binary program (FILEBI) |
| 407 | Delete a filed program (DELETE) |
| 410 - 477 | Binary programs |

*LAP4 does not test for an upper limit on the length of a manuscript. Manuscripts of maximum length ($4000_8$ lines) in unpacked form might exceed the working area assigned above.

## A2-6 LAP4 TAPE ALLOCATION

| Block | Allocation |
|---|---|
| 270 | |
| 271 | |
| 272 | |
| 273 | MANUSCRIPT CONTROL meta command |
| 274 | |
| 275 | |
| 276 | |
| 277 | |
| 300 | Regular input |
| 301 | COPY meta command |
| 302 | Temporary Storage |
| 303 | |
| 304 | CV and CM meta commands |
| 305 | |
| 306 | |
| 307 | Reserved block |
| 310 | SAVE MANUSCRIPT meta command |
| 311 | DISPLAY meta command |
| 312 | |
| 313 | PACK meta command |
| 314 | INSERT meta command |
| 315 | |
| 316 | ADD MANUSCRIPT meta command |
| 317 | REMOVE meta command |
| 320 | Pass III |
| 321 | Regular input |
| 322 | |
| 323 | Temporary storage |
| 324 | Temporary storage for INSERTED lines |
| 325 | |
| 326 | Pass I for conversion |
| 327 | Pass II for conversion |
| 330 | |
| 331 | Binary program after conversion |
| 332 | |
| 333 | |
| 334 | Temporary storage |
| 335 | Manuscript control block |
| 336 and ff. | Manuscript |

## A2-7 LAP4 META COMMANDS

| Command | Required Format | Information Requested During Operation | Comments |
|---|---|---|---|
| REMOVE | RE LN,n<br>RE LN-LN+n | none | *Removes n lines of MS beginning with line LN |
| INSERT<br>END | IN LN<br>EN | none | *Allows insertions of lines prior to line LN; insertion terminated by EN |
| PACK | PA | none | *Removes gaps in MS left by RE, IN, AM, and MC (option 2) |
| DISPLAY | DI LN,S | none | *Displays MS; F: forward; B: backward; L after octal nos.: LN; S after octal nos.: S |
| SAVE MANUSCRIPT | SM | unit number;<br>initial block number | *Saves MS in any designated block on either unit |
| ADD MANUSCRIPT | AM BN,UN | none | Adds MS to working area from any block on either unit |
| CONVERT | CV | none | *Converts MS |
| CONVERT MANUSCRIPTS | CM | initial block number(s) on unit 0 of each MS to be converted | Converts manuscripts residing anywhere on unit 0 tape |
| COPY | CP | number of blocks to be copied; UN and initial BN of places from and to which copy will be made | Copies up to $777_8$ blocks from either tape to either tape |
| START LAP | LA | none | Starts LAP4 system |
| START GUIDE | GU | none | Starts GUIDE system |
| MANUSCRIPT CONTROL | MC | option number 0-4; various additional information requested by options 1-4 | Allows manipulation of MS files; SSW0 down - standard MS file; up - file requested |

LN - line number
n - number of lines
s - size of display

BN - block number
UN - unit number

*Operates only on MS in working area

## A2-8 GUIDE SYSTEM COMMANDS

| Command | Information Requested During Operation | Comments |
|---|---|---|
| INDIS | none | Displays index |
| REWIND | none | Rewinds tapes |
| LAPGO | none | Starts LAP4 |
| LAPRTN | none | Returns to LAP4 |
| CAST | create basic index or retain old index | Creates a systems tape |
| FILEBI | name or block number of program; if necessary, short title, starting location and number of blocks, units to and from; return option 0-1 | Files a binary program by name or block number from and onto either unit |
| DELETE | name and unit number; return options 0-2 | Deletes a filed program by name from either unit |

## A2-9 SUMMARY OF ANSWERING PROCEDURE FOR Q & A

| Status of Display | Result when Key Struck | | | |
|---|---|---|---|---|
| | del | EOL | CASE | All others |
| no questions | inoperative | proceed* | display fades from scope until next character struck; any next character treated as upper case | inoperative |
| no entries in current question | answers to all previous questions deleted | current question filled completely with blanks (14); | display fades from scope until next character struck; any next character treated as upper case | struck character appears on scope in place of one question mark |
| partial entry in current question, question marks remaining | answer to current question deleted | remaining question marks filled with blanks (14); proceed* | display fades from scope until next character struck; any next character treated as upper case | struck character appears on scope in place of one question mark |
| complete entry in current question, no question marks remaining (EOL not yet struck) | answer to current question deleted | proceed* | display fades from scope until next character struck; any next character treated as upper case | inoperative |

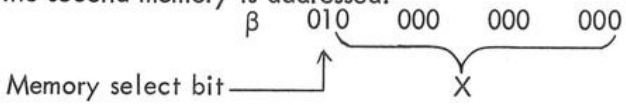
*Proceed either back to program or to next question, whichever applies.

# APPENDIX 3
# EXTENDED MEMORY PROGRAMMING

## A3-1 DOUBLE MEMORY

The LINC has been presented as having a single 12-bit, $1024_{10}$ word memory. A second addressable memory provides $2048_{10}$, or $4000_8$ 12-bit words. This second memory is addressable for data storage and retrieval; it can not, however, be used to hold running programs.

Bit 1 of a register containing a memory address, e.g., a β register, is designated as the memory select bit. When this bit is 1, the second memory is addressed:




The addresses for the second memory may then be thought of as 2000 + X, where $0 \leq X \leq 1777$, as usual.

More simply perhaps, it is referred to as memory registers $2000\text{-}3777_8$. While this scheme makes the memory addresses of the two memories continuous, they can not always be treated as such by the programmer. The instruction location register, having only 10 bits, prohibits using the second memory to hold running programs; the next sequential instruction location after 1777 is always 0. Moreover, the full-address class instructions can address only registers 0-1777.

All other memory reference instructions have available a memory select bit, and can address either memory. The instruction

| | |
|---|---|
| p | LDA |
| p+1 | 2133 |

will load the accumulator with the contents of register 2133, i.e., register 133 of the second memory. It must be remembered, however, that all instructions which index the first 16 registers (index class, half-word class, XSK, and DIS) index 10 bits only, and thus index from 1777 to 0 without affecting the memory select bit. Therefore, by setting bit 1 the programmer can index through either memory he chooses, but he cannot index from one memory to the other, e.g.:

| Memory Address | Memory Contents | |
|---|---|---|
| 3 | [2000 + X] | [-] |
| ⋮ | ⋮ | ⋮ |
| →40 | SET i 3 | 0063 |
| 41 | 3777 | 3777 |
| 42 | →LDA i 3 | 1023 |
| 43 | JMP 42 | 6042 |

In this example register 3 will contain the succession of values: 3777, 2000, 2001, ..., 3777, 2000, etc., repeatedly scanning the second memory. In order for the first execution of the LDA instruction at location 42 to index register 3 to 2000, register 3 must be set initially to 3777, i.e., X(3) = 1777 and memory select bits = 1.

For many purposes this indexing scheme presents no disadvantages. Often, however, one would like to use both memories, for example to collect a large number of data samples. The following program fills memory registers 400-3777 with sample values of the signal on input line 10. The sample-and-store part of the program is written as a subroutine (locations 31-40), and the sample rate is controlled by a OPR i n instruction:

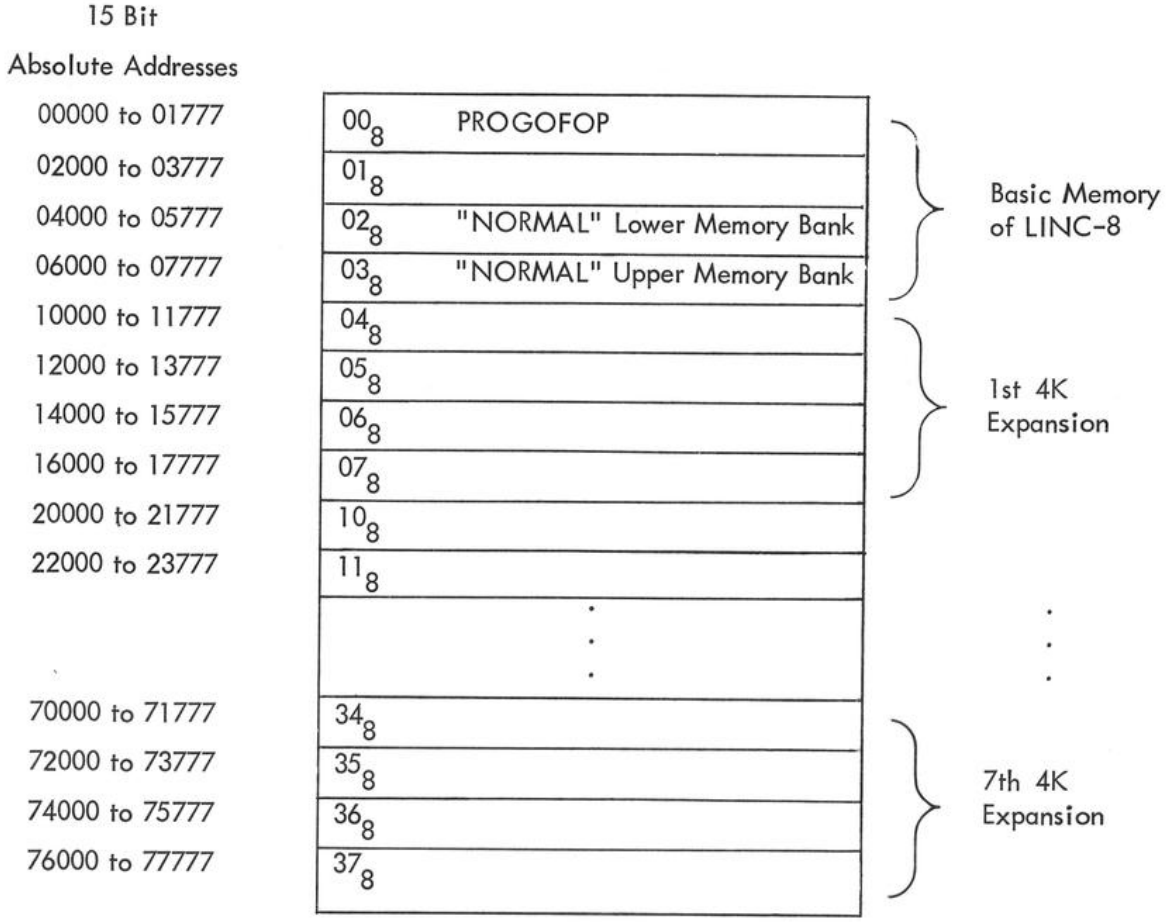
| Memory Address | Memory Contents | | Effect |
|---|---|---|---|
| 7 | [-] | [-] | For memory address. |
| 10 | [JMP X] | [-] | For return point. |
| : | : | : | |
| → 20 | SET i 7 | 0067 | Set 7 to initial address minus 1 and jump to subroutine. |
| 21 | 377 | 0377 | |
| 22 | JMP 31 | 6031 | |
| 23 | SET i 7 | 0067 | Return from subroutine; set 7 to initial address minus 1 for second memory, and jump to subroutine. |
| 24 | 3777 | 3777 | |
| 25 | JMP 31 | 6031 | |
| 26 | WCG | 0705 | Return from subroutine; write memory quarters 1 through 7 in blocks 31-37 and halt. |
| 27 | 6\|31 | 6031 | |
| 30 | HLT | 0000 | |
| 31 | SET 10 | 0050 | Enter subroutine and save return point in register 10. |
| 32 | 0 | 0000 | |
| 33 | OPR i 1 | 0521 | Pause until restart signal appears on external level line 1. |
| 34 | SAM 10 | 0110 | Sample input on line 10 and store. |
| 35 | STA i 7 | 1067 | |
| 36 | XSK 7 | 0207 | If X(7) = 1777, return to get next sample. |
| 37 | JMP 33 | 6033 | |
| 40 | JMP 10 | 6010 | When X(7) = 1777, return to main program via register 10. |

Example 35 Indexing Across Memory Boundaries

## A3-2 CHANGING MEMORY BANKS

In actuality there are more than $2048_{10}$ words in the LINC-8 computer. The basic LINC-8 contains $4096_{10}$ words with the capacity of expansion up to $32{,}768_{10}$ words. From the LINC point of view,

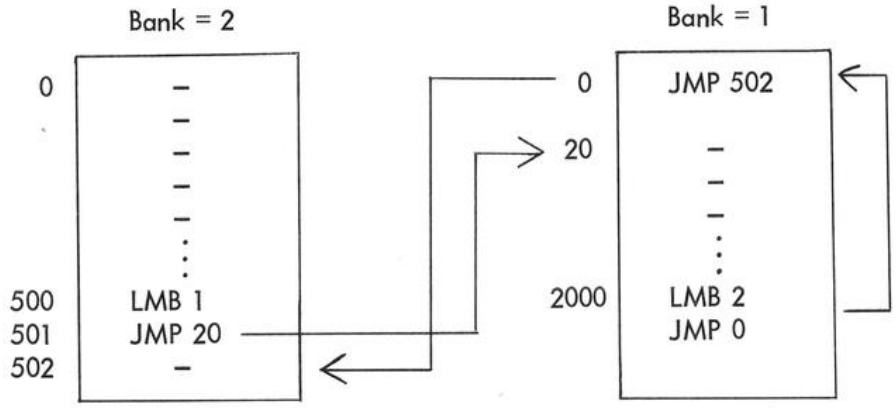
it is best to envision this 32K as 32 1K segments numbered from $00_8$ to $37_8$ (we will call these 1K segments memory banks in this discussion). In memory bank 0, PROGOFOP resides. Normally, LINC programs reside in memory bank 2 (instructions) and 3 (data--see previous discussion). It is possible, however, to change this if such a condition is found desirable.




Diagramatic Representation of LINC-8 Memory

An example of a desirable condition would be when a program requires more than $1024_{10}$ data words to be in computer memory at one time. A decision is made to store the data in memory bank 1 and memory bank 3. The program will occupy memory bank 2 (where it normally is) and the double memory programming technique discussed previously will be used to access this data. The accessing of the data in bank 3 is no problem as bank 3 is the "normal upper memory bank." However, when it is desired to access the data in bank 1 the "number" of the upper memory bank must be changed. This is accomplished by executing the instruction UMB N (640 + N), change upper memory bank to N. N must be nonzero to change the upper memory bank selector. After this instruction is executed, all references to "upper memory" will be to this memory bank (1 in our example) until it is changed by another UMB N.

Another example of when it would be desirable to change memory banks would be if a program were too large to occupy one memory bank and it was desirable to store a frequently used subroutine in a different memory bank. Again, let us use memory bank 1, this time to hold the subroutine. The main program is in memory bank 2; the data, in bank 3. To change memory banks and transfer the control of a program, the instruction LMB N (600 + N), change lower memory bank to N ($N \neq 0$), is given. Upon executing the next JMP X ($X \neq 0$) control will be transferred to location X of bank N (1 in our example) and JMP p+1 will be stored in location 0 of the new memory bank. To exit bank 1 to the original program in bank 2, a LMB 2 instruction is given, followed by a JMP 0.




The upper memory bank number is not affected by the LMB instruction (nor is it directly affected by the JMP X).

# APPENDIX 4
# INSTRUCTIONS

## A4-1 MSC CLASS INSTRUCTIONS

| HLT | 0000 | 3 μsec | HLT |
|---|---|---|---|
| Halt. The computer halts. The RUN light on the console is extinguished. The gong chimes if it is turned on. The computer can be restarted only from the console. | | | |

| ZTA | 0005 | 3 μsec | ZTA |
|---|---|---|---|
| Z to A. The contents of Z, bits 0 to 10, replace the contents of A, bits 1 to 11. Bit 0 of A is set to 0. Bit 11 of Z is ignored. The contents of Z are not changed. | | | |

| CLR | 0011 | 3 μsec | CLR |
|---|---|---|---|
| Clear. Clear A, L, and Z. | | | |

| ATR | 0014 | 3 μsec | ATR |
|---|---|---|---|
| A to R. The contents of the right half of A ($A_6$-$A_{11}$) replace the contents of R. The contents of A are not changed. | | | |

| RTA | 0015 | 3 μsec | RTA |
|---|---|---|---|
| R to A. The contents of R replace the contents of the right half of A. The left half of A is cleared. The contents of R are not changed. | | | |

| NOP | 0016 | 3 μsec | NOP |
|---|---|---|---|
| No operation. This instruction provides a delay of 3 μsec before proceeding to the next instruction. It does nothing. | | | |

| COM | 0017 | 3 μsec | COM |
|---|---|---|---|
| Complement. Complement the contents of A. | | | |

## A4-2 SKIP CLASS INSTRUCTIONS

Address of Next Instruction to be Executed Following Skip Class Instruction

| i | Condition | Address of Next Instruction |
|---|---|---|
| 0 | met | p + 2 |
| 0 | not met | p + 1 |
| 1 | met | p + 1 |
| 1 | not met | p + 2 |

| SNS i n | 0440 + 20i + n | 3 μsec | SNS |
|---|---|---|---|
| Sense switch. Check if sense switch n ($0 \leq n \leq 5$) is up. | | | |

| AZE i | 0450 + 20i | 3 μsec | AZE |
|---|---|---|---|
| A zero. Check if A contains either 0000 or 7777. | | | |

| APO i | 0451 + 20i | 3 μsec | APO |
|---|---|---|---|
| A positive. Check if $A_0$ (the sign bit of A) is 0. | | | |

| LZE i | 0452 + 20i | 3 μsec | LZE |
|---|---|---|---|
| L zero. Check if L is 0. | | | |

| IBZ i | 0453 + 20i | 3 μsec | IBZ |
|---|---|---|---|
| Inter-block zone. Check if either tape unit is up to speed and reading an inter-block zone mark. | | | |

| FLO i | 0454 + 20i | 3 μsec | FLO |
|---|---|---|---|
| Overflow. Check if overflow flip-flop is in the one state. FLOFF (overflow flip-flop) will be set on ADD, ADA, ADM, or LAM if an overflow occurred. Overflow exists when the sum of two positive numbers is negative or the sum of two negative numbers is positive. | | | |

| ZZZ i | 0455 + 20 i | 3 μsec | ZZZ |
|---|---|---|---|
| Z bit 11 on a zero. Check if bit 11 of Z is a 0. | | | |

| SXL i n | 0400 + 20i + n | 3 μsec | SXL |
|---|---|---|---|
| Skip on external level. Check if external level input line n ($0 \leq n \leq 13_8$ is in its negative state. (−3v) | | | |

| KST i | 0415 + 20i | 3 μsec | KST |
|---|---|---|---|
| Key struck. Check if a key has been struck and is in the locked position. | | | |

## A4-3 SHIFT CLASS INSTRUCTIONS

| ROL i n | 0240 + 20i + n | see table below | ROL |
|---|---|---|---|
| Rotate left. Shift the contents of A n places to the left. If i=0, bit $A_0$ is shifted into bit $A_{11}$, and the contents of L are unchanged. If i=1, bit $A_0$ is shifted into L and L is shifted into bit $A_{11}$. No effect on Z. | | | |

| ROR i n | 0300 + 20i + n | see table below | ROR |
|---|---|---|---|
| Rotate right. Shift the contents of A n places to the right. If i=0, bit $A_{11}$ is shifted into bit $A_0$, and $Z_0$ and the contents of L are unchanged. If i=1, bit $A_{11}$ is shifted into L and L is shifted into bit $A_0$, $A_{11}$ is shifted into $Z_0$. $Z_{11}$ is always lost. | | | |

| SCR i n | 0340 + 20i + n | see table below | SCR |
|---|---|---|---|
| Scale right. Shift the contents of A n places to the right with bit $A_0$ never changing. If i=0, the information shifted out of $A_{11}$ is shifted into $Z_0$ and L is not changed. If i=1, bit $A_{11}$ is shifted into L and the information shifted out of L is lost. $A_{11}$ is shifted into $Z_0$. $Z_{11}$ is always lost. | | | |

A4-3.1 Execution Times for Shift Class Instructions

| n | t | n | t |
|---|---|---|---|
| 0 | 3 μsec | 10 | 9.0 μsec |
| 1 | 3 μsec | 11 | 9.0 μsec |
| 2 | 4.5 μsec | 12 | 10.5 μsec |
| 3 | 4.5 μsec | 13 | 10.5 μsec |
| 4 | 6.0 μsec | 14 | 12.0 μsec |
| 5 | 6.0 μsec | 15 | 12.0 μsec |
| 6 | 7.5 μsec | 16 | 13.5 μsec |
| 7 | 7.5 μsec | 17 | 13.5 μsec |

## A4-4 FULL ADDRESS CLASS INSTRUCTIONS

| ADD X | 2000+X | 3 μsec | ADD |
|---|---|---|---|

Add. Add the contents of memory register X to the contents of A, and leave the resulting sum in A. The addition is 1's complement binary addition (i.e., with end-around carry). The contents of memory register X are not changed.

| STC X | 4000+X | 3 μsec | STC |
|---|---|---|---|

Store and clear A. The contents of A are copied into memory register X and A is then cleared.

| JMP X | 6000+X | see note | JMP |
|---|---|---|---|

Jump to X. If $X \neq 0$, 6000 + p + 1 is stored into memory register 0. Regardless of the value of X, the next instruction is taken from the memory location whose address is X.

NOTE: If $X \neq 0$, this instruction is executed in 3 μsec. If X = 0, the instruction is executed in 1.5 μsec.

## A4-5 INDEX CLASS INSTRUCTIONS

Addressing and set up time in index class instructions.

| i | β | Y | t | Indexing |
|---|---|---|---|---|
| 0 | 0 | Y (p+1) | 4.5 μsec | ---------- |
| 1 | 0 | p+1 | 3.0 μsec | ---------- |
| 0 | $1 \leq \beta \leq 17$ | Y(β) | 4.5 μsec | ---------- |
| 1 | $1 \leq \beta \leq 17$ | $Y(\beta)_{ndx}$ | 4.5 μsec | $X(\beta) \rightarrow_{ndx} X(\beta)$ |

| LDA i β | 1000 + 20i + β | t μsec | LDA |
|---|---|---|---|
| Load A. Copy the contents of memory register Y into A. The contents of memory register Y are not changed. | | | |

| STA i β | 1040 + 20i + β | t μsec + 1.5 μsec | STA |
|---|---|---|---|
| Store A. Copy the contents of A into memory register Y. The contents of A are not changed. | | | |

| ADA i β | 1100 + 20i + β | t μsec | ADA |
|---|---|---|---|
| Add to A. Add the contents of memory register Y to the contents of A and leave the resulting sum in A. The addition is 1's complement binary addition (i.e., with end-around carry). The contents of memory Y are not changed. | | | |

| ADM i β | 1140 + 20i + β | t + 3.0 μsec | ADM |
|---|---|---|---|
| Add to memory. Add the contents of A to the contents of memory register Y and leave the sum both in A and in memory register Y. The addition is 1's complement binary addition (i.e., with end-around carry). The contents of L are not changed. | | | |

| LAM i β | 1200 + 20i + β | t + 3.0 μsec | LAM |
|---|---|---|---|
| Link-add to memory. First, add the contents of L (0 or 1) to the contents of A and leave the sum in A. The addition is 2's complement 12-bit binary addition with the end-carry replacing the original contents of L. If there is no end-carry, L is cleared. Next, add the contents of memory register Y to the contents of A and leave the sum in A and in memory register Y. If there is an end-carry, set L to 1. | | | |

| MUL i β | 1240 + 20i + β | t + 30.0 μsec | MUL |
|---|---|---|---|
| Multiply. Multiply the contents of A by the contents of memory register Y and leave half of the product in A. The contents of A and of memory register Y are treated as 1's complement binary numbers with bit 0 serving as a sign bit. Their full product contains 22 bits plus a sign bit. If bit 0 (the h bit) of β is 0, the multiplication is carried out as an integer multiplication and the least significant 11 bits of the product are left in the least significant 11 bit positions in A. The left-most bit of A contains the sign of the product. If the h bit is 1, the multiplication is carried out as a fraction multiplication and the most significant 11 bits of the product are left in the least significant 11 bit positions in A. The left-most bit of A contains the sign of the product. The least significant bits of the product are in Z bits 0 to 10. The sign of the product is also left in L in both cases. The contents of Y are unchanged. | | | |

| SAE i β | 1440 + 20i + β | t + 1.5 μsec | SAE |
|---|---|---|---|
| Skip if A equals. If the contents of A exactly match the contents of memory register Y, take the next instruction from memory location p+2. Otherwise take the next instruction from memory location p+1. The contents of A and of memory register Y are not changed. | | | |

| SRO i β | 1500 + 20i + β | t + 1.5 μsec | SRO |
|---|---|---|---|
| Skip and rotate. Rotate the contents of memory register Y one place to the right. If, after the rotate, bit 0 of the contents of memory register Y is a 0, take the next instruction from memory location p+2. Otherwise, take the next instruction from memory location p+1. | | | |

| BCL i β | 1540 + 20i + β | t μsec | BCL |
|---|---|---|---|
| Bit clear. For each bit position of memory register Y that contains a 1, clear the corresponding bit position in A. The contents of memory register Y are not changed. | | | |

| BSE i β | 1600 + 20i + β | t μsec + 1.5 μsec | BSE |
|---|---|---|---|
| Bit set. For each bit position of memory register Y that contains a 1, set the corresponding bit position in A to a 1. The contents of memory register Y are not changed. | | | |

| BCO i β | 1640 + 20i + β | t μsec | BCO |
|---|---|---|---|
| Bit complement. For each bit position of memory register Y that contains a 1, complement the corresponding bit position in A. The contents of memory register Y are not changed. | | | |

| DSC i β | 1740 + 20i + β | 75-140 μsec | DSC |
|---|---|---|---|

Display character. Intensify points in a 2 x 6 grid on the display scope with the pattern displayed controlled by the contents of memory register Y. Each bit position of memory register Y controls the intensification of one of the twelve points in the grid. The diagram below specifies the bit position controlling each grid point.

| | |
|---|---|
| 6 | 0 |
| 7 | 1 |
| 8 | 2 |
| 9 | 3 |
| 10 | 4 |
| 11 | 5 |

Spacing between points is +4 in both the horizontal and vertical directions. The contents of memory register 1, augmented by +4, controls the horizontal position of the lefthand edge of the grid. The contents of the accumulator, with bits 7-11 set to 0, controls the vertical position of the grid's lower edge. Bit 0 of memory register 1 selects one of two display channels for intensification. At the end of the instruction, the accumulator has been augmented by $30_8$ and register 1 has been augmented by $10_8$. Memory register Y is left unchanged. The Z register is used by this instruction.

## A4-6 HALF WORD CLASS INSTRUCTIONS

Addressing and set up time in half-word class instructions

| i | β | Y | h | t | Indexing |
|---|---|---|---|---|---|
| 0 | 0 | Y(p+1) | h(p+1) | 4.5 usec | ---------- |
| 1 | 0 | p+1 | 0 | 3.0 usec | ---------- |
| 0 | $1 < \beta < 17$ | Y(β) | h(β) | 4.5 usec | ---------- |
| 1 | $1 \leq \beta \leq 17$ | $Y(\beta)_{hndx}$ | $\overline{h(\beta)}$ | 4.5 usec | See note below |

If h=0, the left half of memory register Y is the operand. If h=1, the right half of memory register half of memory register Y is the operand.

| LDH i β | 1300 + 20i + β | t μsec | LDH |
|---|---|---|---|
| Load half. Copy the contents of the designated half of memory register Y into the right half of A. The left half of A is cleared. The contents of memory register Y are not changed. | | | |

| STH i β | 1340 + 20i + β | t μsec + 1.5 μsec | STH |
|---|---|---|---|
| Store half. Copy the contents of the right half of A into the designated half of memory register Y. The contents of A and of the remaining half of memory register Y are not changed. | | | |

| SHD i β | 1400 + 20i + β | t μsec + 1.5 μsec | SHD |
|---|---|---|---|
| Skip if half differs. Compare the contents of the right half of A with the contents of the specified half of memory register Y. If they do not match exactly, take the next instruction from memory address p+2. If they do match, take the next instruction from memory address p+1. The contents of A and of memory register Y are not changed. | | | |

NOTE: $X(\beta)_{hndx} \rightarrow X(\beta)$ and $\overline{h(\beta)} \rightarrow h(\beta)$

## A4-7 ALPHA CLASS INSTRUCTIONS

| SET i α | 40 + 20i + α | i=1: 4.5 μsec, i=0: 6.0 μsec | SET |
|---|---|---|---|
| Set. Set the contents of memory register α equal to the contents of memory register Y. The contents of memory register Y are not changed. | | | |

| DIS i α | 140 + 20i + α | 18 μsec | DIS |
|---|---|---|---|
| Display. Index the contents of memory register α if i=1. Intensify a point on the scope whose horizontal position is specified by bits 3-11 of memory register α and whose vertical position is specified by bits 3-11 of A. Bit 0 of memory register α selects one of two display channels for intensification.<br><br>The leftmost point which can be displayed corresponds to the horizontal coordinate 000 octal, and the rightmost point to 777. The lowest point which can be displayed corresponds to the vertical coordinate −377, and the highest to +377. Bits 0 through 2 of memory register α and of A do not effect the position of the point which is intensified. | | | |

| XSK i α | 200 + 20i + α | 4.5 μsec | XSK |
|---|---|---|---|
| Index and skip. Index the contents of memory register α if i=1. If contents of (α) equals 1777, take the next instruction from p+2. Otherwise, take the next instruction from p+1. | | | |

## A4-8 SAMPLE CLASS INSTRUCTIONS

| SAM i n | 100 + 20i + n | 19.5 μsec | SAM |
|---|---|---|---|
| Sample. Sample the signal on one of 16 input channels selected by n. Leave its binary value, seven bits plus sign bit, in the least significant bit positions of the accumulator. The sign bit is extended through bit 0. $0 \leq n \leq 7$ selects one of the potentiometers on the display scope: $10 \leq n \leq 17$ selects one of the analog inputs in the terminal box. The i bit has no effect unless equipped with additional analog channels; then i selects the second 16 channels. | | | |

## A4-9 MAGNETIC TAPE CLASS INSTRUCTIONS

These instructions are used to transfer information between the internal core memory of the LINC and digital tapes on either of the two LINC tape transports. Two additional instructions in this class permit moving tapes and checking the integrity of information stored on tapes without modifying the contents of the tapes or the core memory.

LINC tapes are subdivided permanently into blocks by an initial process of marking which records a fixed pattern on certain portions of the tape. This pattern includes fixed block addresses which permit references to information stored on tape by means of a block number. Information is always transferred and checked in units of complete blocks which are specified by their block addresses. Each block includes a checksum for verifying the integrity of information transfers to and from the tape.

A standard LINC tape contains $1000_8$ ($512_{10}$) addressable blocks numbered consecutively from 000-777. Each block on a standard tape contains $400_8$ ($256_{10}$) words corresponding to the contents of one quarter of a LINC core memory module. The first core memory address whose contents are transferred to or from a given block is the first address within the specified memory quarter. Quarter 0 begins with address 0, quarter 1 with address 400, and so on up to quarter 7, which begins with address 3400. The length of blocks as well as the format of the block addresses can be varied by using a non-standard tape which has been marked using a non-standard marking program. Once a given tape is marked, however, the format is fixed for that tape unless it is completely erased and remarked.

### A4-9.1 Tape Motion and Searching

Tape motion and searching are essentially identical for all magnetic tape instructions. The tape system automatically searches for and finds the specified block on tape and then completes the desired operation. If the tape transport selected by a magnetic tape class instruction is not in motion at the time it is selected, it starts to move the tape in the forward direction (i.e., toward larger block addresses) until the next block address is read. If the tape unit selected is already in a state of motion, the initial direction of motion continues until the next block address is read. At this time the desired block number is compared with the block address just read and the tape is subsequently moved in the correct direction to approach the desired block. When the desired block address is reached and the tape is moving in the forward direction, the specified tape operation is carried out.

At the conclusion of a tape instruction the i-bit of the tape instruction determines whether the motion of the tape is to continue or not. If the i-bit is a 1, tape motion continues in the direction it was moving at the completion of the instruction (usually forward). If the i-bit is a 0, the tape is moved backwards for a fraction of a second and is then stopped. In the event that the tape has been left in motion by a 1 in the i-bit position, the motion continues until either the end of the tape is reached, the computer is halted, or the next magnetic tape instruction is executed. Only one tape transport can be in motion under computer control at a given time.

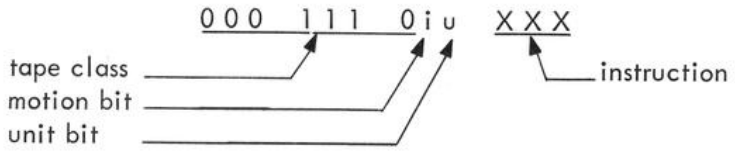
## A4-9.2 Transfer Check

The term data sum refers to the sum (2's complement) of all the data words in a block. A data sum is automatically calculated in the accumulator whenever a block of tape is written, and its complement, called the checksum, is written in a special place at the end of the block. When a block of tape is read, a new data sum is automatically calculated in the accumulator and the checksum from tape is added to it. If the result, called the transfer sum, is not −0, there has been a transfer error. Usage of this feature varies from instruction to instruction and is, therefore, described in detail for each.

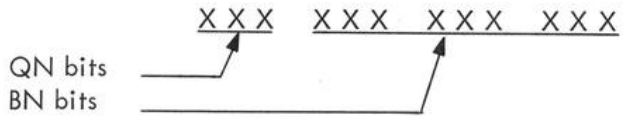
## A4-9.3 Instruction Format

All magnetic tape class instructions are two-word instructions. The first word specifies the instruction, selects the left or right tape unit, and determines the motion of the tape following the completion of the instruction. The second word specifies the memory quarter(s) and the tape block address(es) used in the transfer.

First word:




Second word:




Motion bit: See A4-9.1

Unit bit: If u=0, the lefthand unit is selected.
If u=1, the righthand unit is selected.

QN & BN bits: (for instructions RDC, RDE, WRC, WRI)

The QN bits specify the quarter of memory involved.
The BN bits specify the block of tape involved.

QN & BN bits: (for instructions RCG and WCG)

The QN bits specify the number of consecutive block-quarter transfers to occur, after the first block-quarter transfer.

The BN bits specify the first block of tape involved in the transfer. Bits 0-2 of the BN bits specify the first quarter of memory involved in the transfer.

QN & BN bits: (for instructions MTB and CHK)

The QN bits have no meaning. In the CHK instruction, the BN bits specify the block involved. The use of the BN bits in the MTB instruction is explained under that instruction.

| RDC i u | 700 + 20i + 10u | RDC |
|---|---|---|
| Read and check. The specified block is read into the specified quarter of memory and the accumulator is examined for −0. If it is −0, the computer goes on to the next instruction. If not, the block is re-read until either a −0 results or the computer is halted from the console. | | |

| RCG i u | 701 + 20i + 10u | RCG |
|---|---|---|
| Read and check group. The specified blocks are read into the specified quarters of memory. After each block is read, the accumulator is examined for −0. If it's not, the block is re-read until either a −0 results or the computer is halted from the console. When all blocks have been read successfully, the computer goes on to the next instruction. | | |

| RDE i u | 702 + 20i + 10u | RDE |
|---|---|---|
| Read tape. The specified block of tape is read into the specified quarter of memory. The transfer sum is left in the accumulator and the computer goes on to the next instruction. | | |

| MTB i u | 703 + 20i + 10u | MTB |
|---|---|---|
| Move towards block. The first block address read from tape is subtracted from the block number specified by the BN bits of the instruction and the difference is left in the accumulator. If i=1, the tape is left in motion at the end of the instruction. Tape motion will be in the forward direction if the accumulator is positive and in the backwards direction if the accumulator is negative (including −0). If i=0, tape motion stops at the end of the instruction. | | |

| WRC i u | 704 + 20i + 10u | WRC |
|---|---|---|
| Write and check. The specified quarter of memory is written in the specified block of tape. The tape then reverses, re-finds the specified block, reads it, and examines the accumulator for −0. If it is −0, the computer goes on to the next instruction. If not, the block is re-written and re-read until either −0 results or the computer is halted from the console. Memory is not changed by this instruction. | | |

| WCG i u | 705 + 20i + 10u | WCG |
|---|---|---|
| Write and check group. The specified quarters of memory are written into the specified blocks of tape. After all the blocks are written, the tape reverses, re-finds the first block, and reads all of the blocks just written. The accumulator is examined for −0 after each block is read, and if it is not −0, the whole instruction is repeated, beginning with the block that failed. When all blocks have been successfully written, the computer goes on to the next instruction. Memory is not changed by this instruction. | | |

| WRI i u | 706 + 20i + 10u | WRI |
|---|---|---|
| Write tape. The specified quarter of memory is written in the specified block of tape. The checksum is left in the accumulator and the computer goes on to the next instruction. Memory is not changed by this instruction. | | |

| CHK i u | 707 + 20i + 10u | CHK |
|---|---|---|
| Check tape. The specified block of tape is read and the transfer sum is left in the accumulator. Memory is not changed by this instruction. | | |

## A4-10 OPERATE CLASS INSTRUCTIONS

| OPR i n | 500 + 20i + n | OPR |
|---|---|---|
| Operate channel n. These instructions form a powerful, though complex, set of input-output commands whose functions are partially controlled by signals from external equipment. They are executed by the PDP-8 portion of the LINC-8. | | |

| OPR i 13 | 513 + 20i | < 200 μsec | (PDP) |
|---|---|---|---|
| Transfer control to PDP-8 mode by executing a PDP-8 JMS instruction to the absolute address specified in the Linc A register. This address is taken to be in the first 4K segment of LINC-8 memory. i Bit has no effect. | | | |

| OPR 14 (TYP) | 514 | <200 μsec | TYP |
|---|---|---|---|
| Print the ASCII Character in bits 4-11 of the Linc A register. If Teletype Printer is free, put character in printer buffer and immediately return to LINC programming. If printer is not finished, then pause until character can be put in printer buffer. | | | |

| OPR i 15 | 515 + 20i | <200 μsec | KBD |
|---|---|---|---|
| Read keyboard. If a key has been struck, the code number corresponding to the key is read into A, and the instruction is completed without pausing whether i=0 or i=1. If no key has been struck and i=1, the computer pauses until a key is struck, then reads the key code into A, and continues to the next instruction. If no key has been struck and i=0, there is no pause and the computer goes to the next instruction with A cleared. | | | |

| OPR i 16 | 516 + 20i | <200 μsec | RSW |
|---|---|---|---|
| Right switches. The contents of the RIGHT SWITCHES on the console are read into A. There is no pause. The i bit has no effect. | | | |

| OPR i 17 | 517 + 20i | <200 μsec | LSW |
|---|---|---|---|
| Left switches. The contents of the LEFT SWITCHES on the console are read into A. There is no pause. The i bit has no effect. | | | |

## INDEX OF PROGRAMMING EXAMPLES

## PAGE INDEX OF LINC-8 INSTRUCTIONS