PROGRAMMING THE LINC

SECOND EDITION

Computer Systems Laboratory
Washington University
St. Louis, Missouri

PROGRAMMING THE LINC

Second Edition

Mary Allen Wilkes and Wesley A. Clark

Computer Systems Laboratory
Washington University
724 South Euclid Avenue
St. Louis, Missouri 63110

This work was supported by the Division of Research Facilities and Resources of the National Institutes of Health under grant FR-218-01-03, and, in cooperation with the Bio-Sciences Office of the National Aeronautics and Space Administration, under NIH contract PH43-63-540. "Programming the LINC" originally appeared as Section 2 of LINC Volume 16, Programming and Use I, Computer Research Laboratory, Washington University, St. Louis, Missouri, June 1965.

First edition: June 1965
Reissue: August 1967
Second edition: January 1969

For whom the gong perhaps chimes

PROGRAMMING THE LINC

Contents

Contents

Index of Programming Examples

Page Index of LINC Instructions

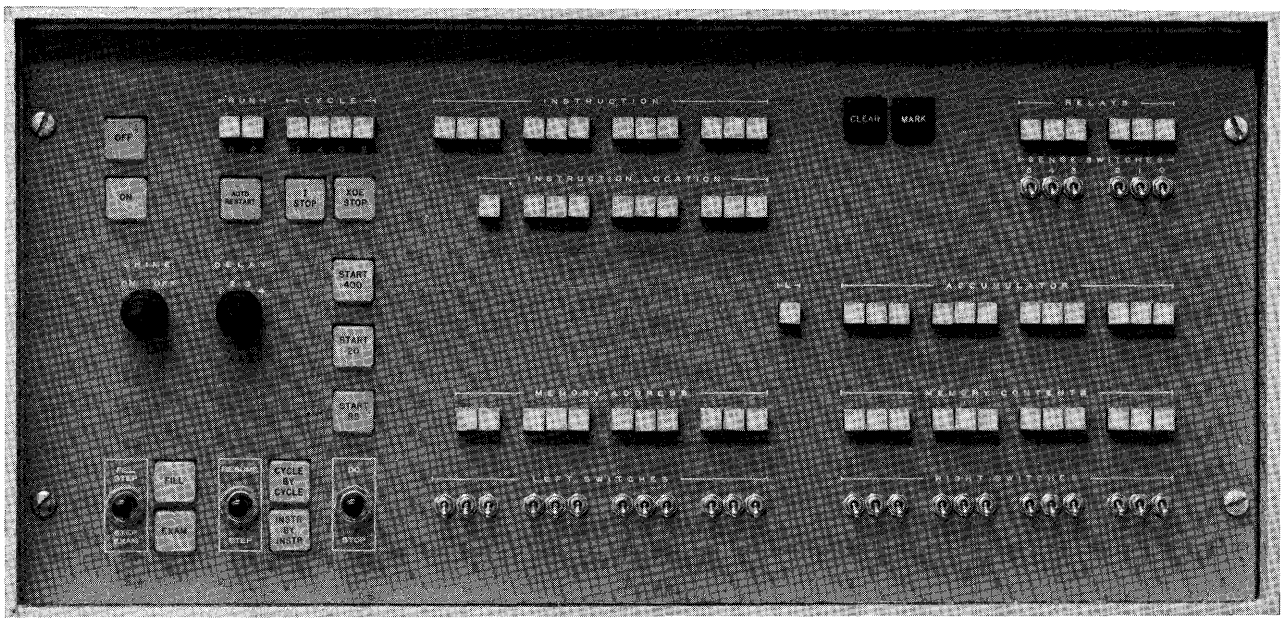
PROGRAMMING THE LINC

1. Introduction

The LINC (Laboratory Instrument Computer) is a stored-program binary-coded digital computer designed to operate in the laboratory environment as a research tool. The following description is intended to serve as a general introduction to basic programming concepts and techniques, and specifically as an introduction to LINC programming.

The "classic" LINC,[1] the basis of this document, has found variation in manufacture in the form of the LINC-8 and the micro-LINC. Other variations may yet appear. The fundamental programming techniques, however, are the same for all varieties, and references to "the LINC" in the following can generally be read without respect to variant. A summary on LINC Variants is provided in Appendix IV. It especially affects Chapter 16, and all questions of instruction execution times.

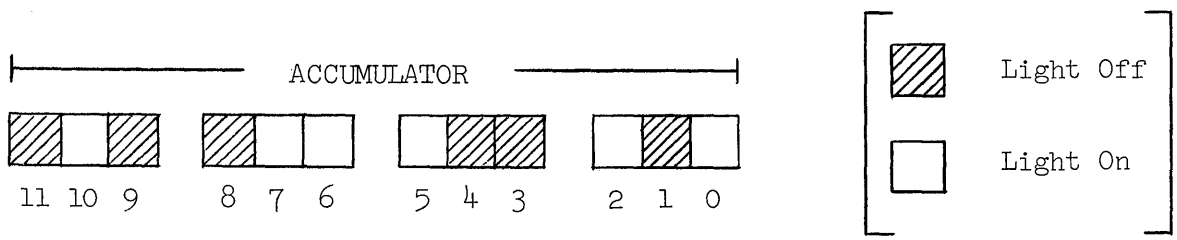
Like most digital computers, the LINC operates by manipulating binary numbers held in various registers (storage devices for numbers), under the control of a program of instructions which are themselves coded as binary numbers and stored in other registers. LINC instructions generally fall into types 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 is therefore recommended when class characteristics are described. Furthermore, not all LINC instructions are described here in detail, specifically those resulting from modifications to the computer as covered in Appendix III. Therefore, this document should be read in conjunction with the LINC Order Code Summary, Appendices II and III-6.



The best way to begin is to consider only a few of the registers and switches which are shown on the LINC Control Console:[2] the ACCUMULATOR (ACC) which is a register of 12 lights, the LINK BIT (L), the LEFT and RIGHT SWITCHES, which are rows of 12 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.

2. Number Systems

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



then the binary number stored in the Accumulator is

$$C(ACC) = 010\ \ 011\ \ 100\ \ 101 \quad \text{(binary)}$$

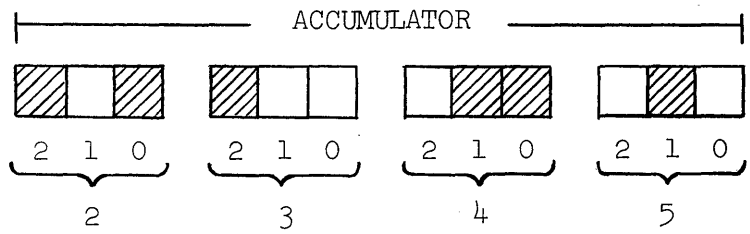
which has the decimal value

$$\begin{aligned} C(ACC) &= 2^{10} + 2^{7} + 2^{6} + 2^{5} + 2^{2} + 2^{0} \\ &= 1024 + 128 + 64 + 32 + 4 + 1 \\ &= 1253 \quad \text{(decimal)} \end{aligned}$$

We can also view this as an <u>octal</u> number by considering each group of three bits in turn. In this example, grouping and factoring proceed as follows:

$$\begin{aligned} C(ACC) &= (2^{10}) + (2^{7}+2^{6}) + (2^{5}) + (2^{2}+2^{0}) \\ &= (2^{1})\cdot 2^{9} + (2^{1}+2^{0})\cdot 2^{6} + (2^{2})\cdot 2^{3} + (2^{2}+2^{0})\cdot 2^{0} \\ &= (2)\cdot 8^{3} + (3)\cdot 8^{2} + (4)\cdot 8^{1} + (5)\cdot 8^{0} \\ &= 2 \qquad 3 \qquad 4 \qquad 5 \\ &= 2345 \quad \text{(octal)} \end{aligned}$$

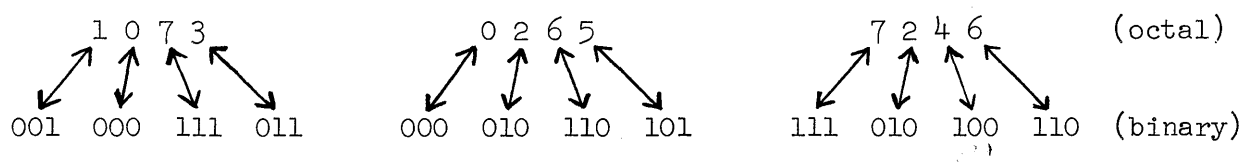
To put this more simply, each octal digit can be treated as an independent 3-bit binary number whose value, (0, 1, ..., 7), can be obtained from the weights 2^2, 2^1, and 2^0:



= 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.

The 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 <u>signed number</u>. One of the bits must be reserved for the sign of the number. The left-most bit is therefore identified as the SIGN BIT (0 for +, 1 for -). To change the sign of a binary number, we <u>complement</u> the number (replace all ZEROS by ONES and vice-versa).[3] 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.

We say that the pair of binary numbers 101111110011 and 010000001100 are ones' complements of each other, (in octal these are 5763 and 2014), and will 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 the number zero:

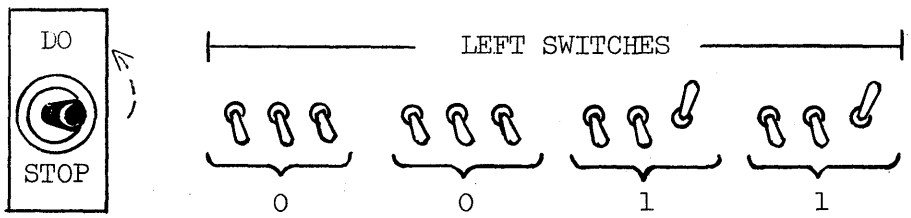
000 000 000 000 = +0
111 111 111 111 = -0

Note finally that the sum of any binary number and its complement is always a zero of the second kind, "minus zero," in this system.

3. Simple Instructions

The LINC 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, will be 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 will then halt. For example, if we set the Left Switches to the code number for the instruction "CLEAR," which happens to be 0011 (octal), and then momentarily raise the DO lever, the Accumulator lights will all go out and so will the Link Bit light, so that C(ACC) = 0, and C(L) = 0. In setting a switch, "up" corresponds to "one."



Left Switches set to 0011 (octal), the code number for "CLEAR."

COM

ATR

RTA

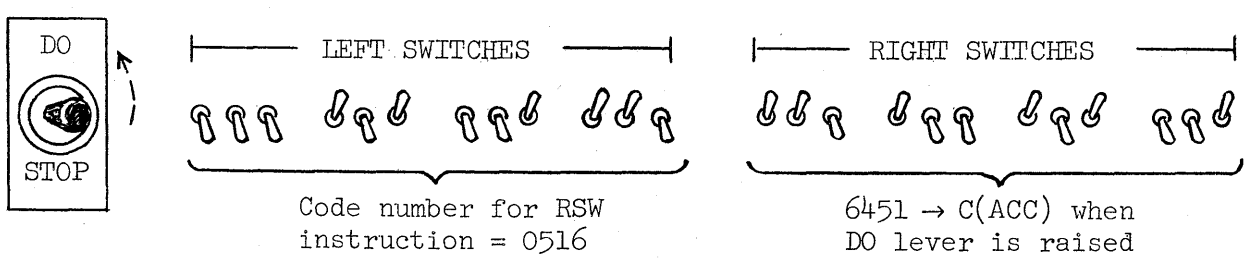
RSW

Tersely: If C(Left Switches) = 0011 (octal), then DO has the effect $0 \rightarrow C(ACC)$ and $0 \rightarrow C(L)$. (Read "zero 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, COMPLEMENT (or COM), with the code number 0017 (octal), 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.")

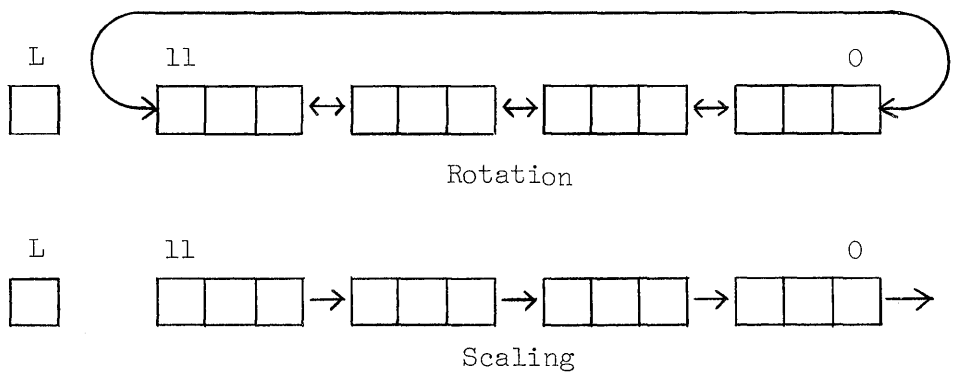
Two other instructions of this class transfer information between the Accumulator and the Relay Register. The Relay Register, displayed on the upper right corner of the Control Console, operates 6 relays which can be used to control or run external equipment. An instruction with the code 0014 (octal), called ACCUMULATOR TO RELAY, ATR, directs the LINC to copy the contents of the right <u>half</u> of the Accumulator, i.e., the right-most 6 bits, into the Relay Register. The Accumulator itself is not changed when the instruction is executed. Another instruction, called RELAY TO ACCUMULATOR, RTA, with the octal code 0015, causes the LINC to clear the Accumulator and then copy 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 is left cleared (i.e., containing zeros).

Another instruction called RIGHT SWITCHES, RSW, with the code number 0516 (octal), directs the LINC to copy the contents of the Right Switches into the Accumulator. By setting the Left Switches to 0516, the Right Switches to whatever value we want to put in the Accumulator, and then momentarily raising the DO lever, we can change the contents of the Accumulator to any new value we like. The drawing shows how the switches should be set to put the number 6451 (octal) into the Accumulator:



4. Shifting

After a number has been put into the Accumulator it can be repositioned or "shifted," to the right or left. There are two ways of shifting: rotation, in which the end-elements of the Accumulator are connected together so as 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

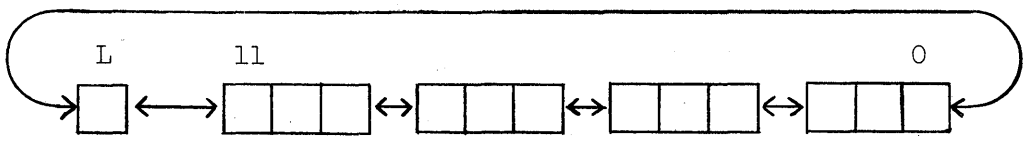
Note that, in scaling, bits are lost to the right, which amounts to an error of "rounding off"; the original sign is preserved in the Sign Bit and replicated in the bit positions to the right of the Sign Bit. This has the effect of reducing the size of the number by powers of two (analogous to moving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

Shifting

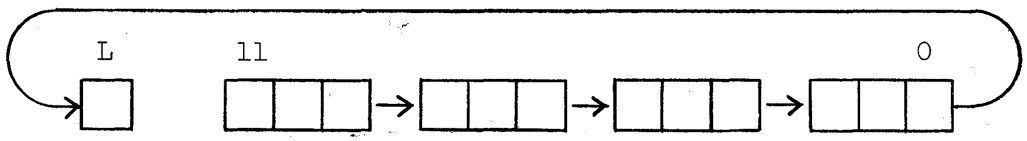
ROR
ROL
SCR

The LINC has three instructions, called the Shift Class instructions, which shift the contents of the Accumulator; these are: ROTATE RIGHT, ROTATE LEFT, and SCALE RIGHT. Unlike the simple instructions we have 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, we write "ROL n," which means "rotate the contents of the Accumulator n places to the left," where n can be any number from 0 through 17 (octal).

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 0 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. We use the full expression <u>ROL i n</u>, which has the octal coding:

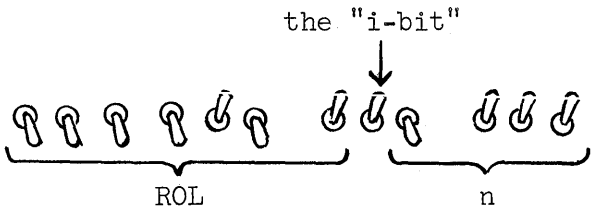
$$\text{ROL } i\ n \qquad 0240 + 20i + n$$

$i = 0$: ACC only
$i = 1$: Link $\longleftrightarrow$ ACC

number of places to shift
($n = 0, 1, \ldots, 17$)

so that, for example, ROTATE ACC LEFT 3 PLACES has the code number 0243, and ROTATE ACC WITH LINK LEFT 7 PLACES has the code number 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

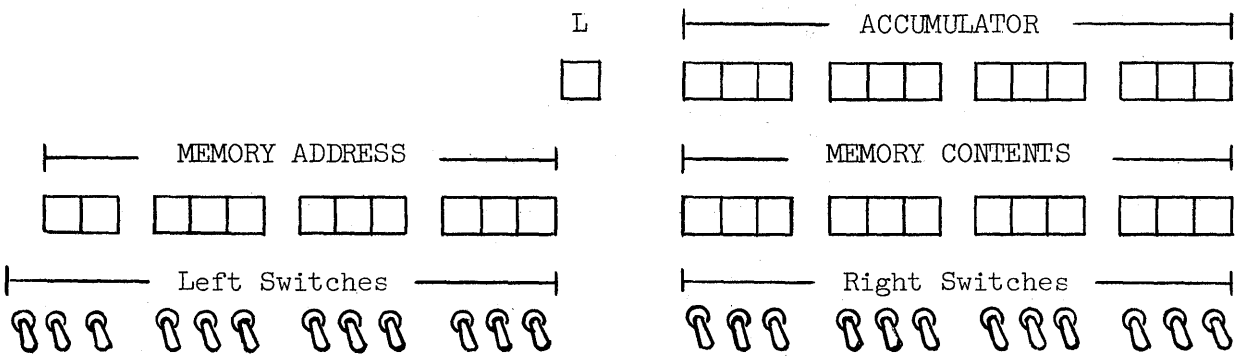
Code number = 0267

Similar coding is used with ROTATE RIGHT, ROR i n, 300 + 20i + n, and SCALE RIGHT, SCR i n, 340 + 20i + n.

5. LINC Memory and Memory Reference Instructions

Before we can proceed to other instructions it is necessary to introduce the LINC Memory. This Memory is to be regarded as a set of 1024 (decimal) registers* each holding 12-bit binary numbers in the manner of the Accumulator. These memory registers are numbered 0, 1, ..., 1023 (decimal), or 0, 1, ..., 1777 (octal), and we shall speak of "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 we can view it in terms of two registers of lights, namely the MEMORY ADDRESS register and the MEMORY CONTENTS register:



*See Appendix I for a discussion of the LINC as a "double memory" machine.

By using these two registers in conjunction with the Left Switches it is possible to find out what values the memory registers contain. If, for example, we are interested in the contents of register 3, we may set the Left Switches to the memory address 0003 and then push the button labeled EXAM. We will see 0003 in the Memory Address register, and the contents of register 3 will appear in the Memory Contents register. By setting the Left Switches to a memory address and pushing EXAM, we can examine the contents of any register in the LINC Memory.

The contents of any selected memory register may be changed by using both the Left and Right Switches and the pushbutton marked FILL. If, for example, we want the memory register whose address is 700 to contain -1 (i.e., 7776 octal) we again set the memory address, 0700, in the Left Switches. We set the Right Switches to the value 7776 and push the FILL button. A 0700 will appear in the Memory Address register and 7776 will appear in the Memory Contents 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 has taken its place. In this way any register in the LINC Memory can be filled with a new number.

None of the LINC instructions makes explicit reference to the Memory Address register or Memory Contents 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 Contents register.

The STORE-CLEAR Instruction

Now we can describe the first of the memory reference instructions, STORE-CLEAR X, STC X, which has the code number 4000 + X, where $0 \leq X \leq 1777$ (octal). (From now on we will use only octal numbers for addresses.) Execution of STC X has two effects: 1) the contents of the Accumulator are copied into memory register X, C(ACC) $\rightarrow$ C(X), and 2) the Accumulator is then cleared, 0 $\rightarrow$ C(ACC). (The Link Bit is not cleared.) Thus, for example, if C(ACC) = 0503 and C(671) = 2345, and we set the code

number for STC 671, i.e., 4671, in the Left Switches, then raising the DO lever will put 0 into the Accumulator and 0503 into register 671. The original contents of register 671 are lost.

It will be clear, now, that the Memory can be filled with new numbers at any time either by using the FILL pushbutton 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.

The ADD Instruction and Binary Addition

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., $C(X) + C(ACC) \rightarrow C(ACC)$. If the Accumulator is first cleared, ADD X will, of course, have the effect of merely copying into the Accumulator the contents of memory register X, i.e., $C(X) \rightarrow C(ACC)$. In any case, the contents of memory register X are unaffected by the instruction.

The addition itself takes place in the binary system,[3] 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 left-most column ("end-carry") is brought around and added into the right-most column ("end-around carry"). Some examples (begin at the right-most 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)
```

The reader should try some examples of his own, and incidentally 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 number we would like to have. For example, adding 1 to the largest positive integer in this system (+3777, octal) results in the largest negative integer (-3777, octal). This is sometimes called "overflowing the capacity of the Accumulator." *

6. The 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 is designed to execute such a "stored program" of instructions by fetching and carrying out each instruction in sequence, using a special 10-bit register called the INSTRUCTION LOCATION register, (IL), to hold the address of the next instruction to be executed. Using the FILL pushbutton and the Left and

* See Appendix III.

Right Switches already discussed, we can, for example, put the code numbers for a series of instructions into memory registers 20-24 which will divide by 8 the number held in memory register 30 and store the result in memory register 31:

	Memory Address	Memory Contents		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.

We can use the FILL pushbutton 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. Pushing the console button 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(IL) + 1 → C(IL). When the Instruction Location register contains 24, the computer encounters the instruction HLT, code 0000, which halts the machine. To run the program again we merely push the START 20 pushbutton. (The code numbers for the instructions will stay in memory registers 20 - 24 unless they are deliberately changed.)

The JUMP Instruction

The last Full Address instruction, JUMP to X, JMP X, with the code number 6000 + X, has the effect of setting the Instruction Location register to the value X; $X \rightarrow C(IL)$. 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 pushbutton, the Instruction Location register would hold the succession of values: 20, 21, 22, 23, 24, 20, 21, etc. (Later we will introduce instructions which increase C(IL) 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 Contents		Effect
0	JMP p+1	6000 + p+1	
.	.	.	
.	.	.	
→ p	JMP X	6000 + X	$X \rightarrow C(IL)$, and "JMP p+1" $\rightarrow C(0)$.
p+1	.	.	
.	.	.	
.	.	.	
X	→ -	-	Next instruction.

This "JMP p+1" code replaces the contents of register 0 every time a JMP X instruction is executed <u>unless X = 0</u>, in which case the contents of 0 are unchanged. The use of memory register 0 in this way is relevant to a programming technique involving "subroutines" which will be described later.

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 through 210 with the appropriate code and data numbers, the program must be started at memory register 175. Since there is no "START 175" button on the console, this is done by setting the Right Switches to 0175 and pushing the console button labeled START RS (Start Right Switches).

Memory Address	Memory Contents		Effect
Start → 175	CLR	0011	$0 \rightarrow C(ACC)$.
176	ADD 201	2201	$N_1 \rightarrow C(ACC)$.
177	COM	0017	Forms $-N_1$.
200	JMP 204	6204	Jumps around data; $204 \rightarrow C(IL)$, and JMP $201 \rightarrow C(0)$.
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	$(N_2-N_1) \rightarrow C(ACC)$.
205	SCR 2	0342	Divides by 4.
206	STC 203	4203	Stores result in 203; $C(ACC) \rightarrow C(203)$; $0 \rightarrow C(ACC)$.
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.

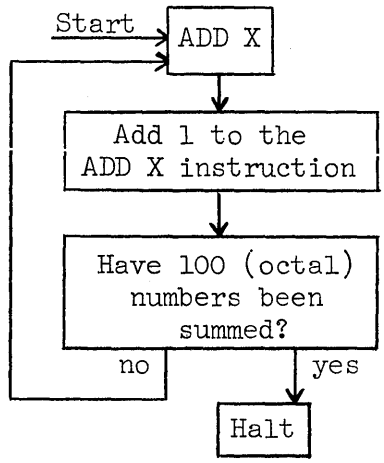
7. Address Modification and Program "Loops"

Frequently a program of instructions must deal with a large set of numbers rather than just one or two. Suppose, for example, that we want to add together 100 (octal) numbers and that the numbers are stored in the memory in registers 1000 through 1077. We want to put the sum in memory register 1100. We could, of course, write out all the instructions necessary to do this,

Memory Address	Memory Contents		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 will be more than 100 (octal) registers long. A more complex, but considerably shorter, program can be written using a programming technique known as "address modification." Instead of writing 100 (octal) ADD X instructions, we write only <u>one</u> ADD X instruction, which we repeat 100 (octal) 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 it is 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 we introduce a new instruction ACCUMULATOR ZERO, AZE, code 0450. This is one of a class of instructions known as the Skip instructions; it directs the LINC to skip the instruction in the next memory register when C(ACC) = either positive or negative zero (0000 or 7777, octal). If C(ACC) $\neq$ 0, the computer does not skip. For example, if C(ACC) = 7777, and we write:

Memory Address	Memory Contents	
→ p	AZE	0450
p+1	-	-
p+2	- ←	-

the computer will take the next instruction from p+2. That is, when the AZE instruction in register p is executed, p+2 will replace the contents of the Instruction Location register, and the computer will skip the instruction at p+1. If C(ACC) $\neq$ 0, then p+1 → C(IL) and the computer executes the next instruction in sequence as usual.

The following example sums the numbers in memory registers 1000 through 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 Contents		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 (octal) 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 through 36 will be executed 100 (octal) 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 register in the new table contains zero. When the program halts, the Accumulator contains the number of non-zero entries.

Memory Address	Memory Contents		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	
101	ADD 4	2004	Code for ADD 400 → C(106).
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) = zero, 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.

At the end of the program, register 106 will contain the code for ADD 451, and all numbers in the table will 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.

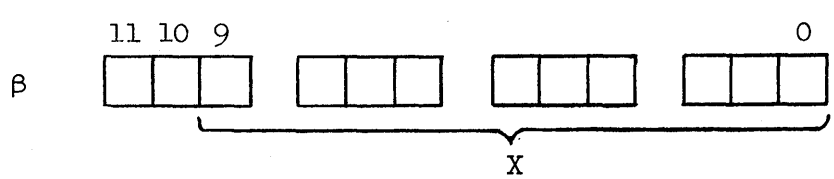
8. Index Class Instructions I

Indirect Addressing

The largest class of LINC instructions, the 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 (octal) register memory. The Index Class instructions, however, have only 4 bits reserved for an address, and can therefore address only memory registers 1 through 17 (octal). The instruction ADD TO ACCUMULATOR, ADA i β, octal code 1100 + 20i + β, is typical of the Index Class:

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

Memory register β should be thought of as containing a memory address, X, in the right-most 10 bits,



and we speak of X(β), meaning the right 10-bit address part of register β. The left-most bit can have any value whatever, and, for the present, bit 10 must be zero.* In addressing memory register β, an Index Class instruction

* See Appendix I.

tells the computer where to find the memory address to be used in executing the instruction. This is sometimes called "indirect" addressing.

For example, if we want to add the value 35 to the contents of the Accumulator, and 35 is held in memory register 270, we can use the ADA instruction in the following manner:

Memory Address	Memory Contents		Effect
β	0270	0270	Address of register containing 35.
⋮	⋮	⋮	
0270	0035	0035	
⋮	⋮	⋮	
→ p	ADA β	1100 + β	C(270) + C(ACC) → C(ACC).
⋮	⋮	⋮	

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 through 17 in this way, the Index Class instructions can refer to any register in the memory.

Two other Index Class instructions, LOAD ACCUMULATOR, LDA i β, and STORE ACCUMULATOR, STA i β, 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 1000 + 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 Contents		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.

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 through 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 Contents		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 left-most 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. The utility of Index Registers in scanning tables of numbers should be obvious.

Indexing involves only 10-bit numbers, and <u>does not involve end-around carry</u>. 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, SKIP IF ACCUMULATOR EQUALS, SAE i β, 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 through 1777; the SAE instruction is used to decide whether the last 0000 has been stored.

Memory Address	Memory Contents		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 through 354, the "loop," will be executed 400 (octal) times before the program halts. Zero is first stored in register 1400, next in 1401, etc.

ADM

BCL

Another program scans the memory to see if a particular number, Q, appears in any memory register 0 through 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 Contents		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.

If no memory register 0 through 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, ADD TO MEMORY, ADM i β, 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 0 through 5; bits 6 through 11 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 Contents		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 ACC.
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.

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 β BIT CLEAR code: 1540 + 20i + β

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 β BIT SET code: 1600 + 20i + β

Set to ONE 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 β BIT COMPLEMENT code: 1640 + 20i + β

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
```

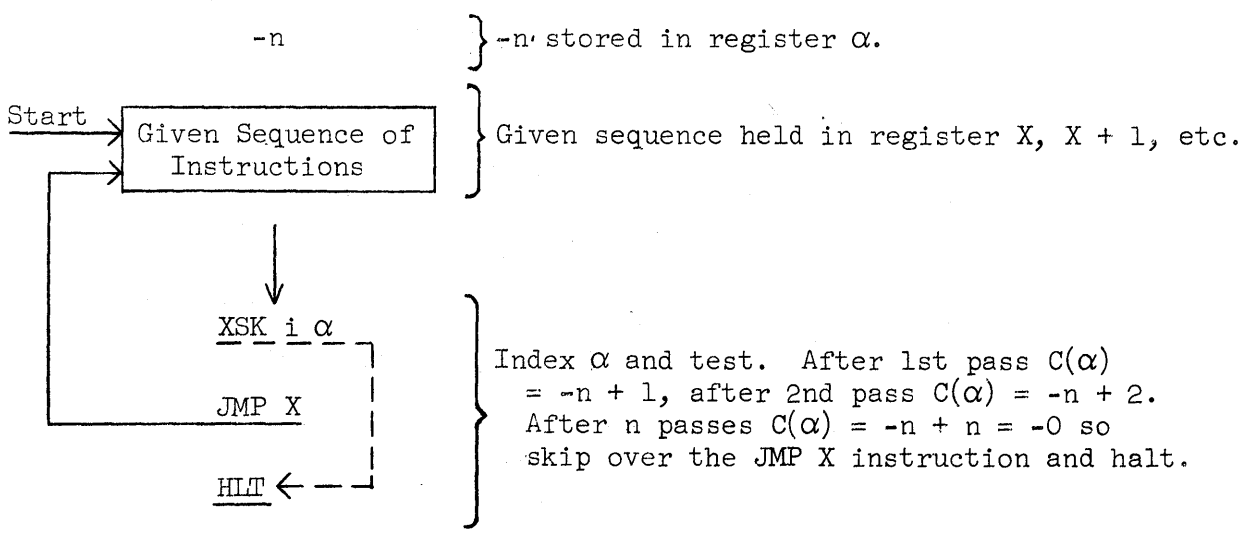
These instructions have a variety of applications, some of which will be demonstrated later.

9. 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.

The INDEX AND SKIP Instruction

The INDEX AND SKIP instruction, XSK i α, code 200 + 20i + α, refers to registers 0 through 17 ($0 \leq \alpha \leq 17$).* It tests to see whether the address part of register α 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 α by 1. Like the Index Class instructions, XSK indexes register α <u>before</u> examining it, and it indexes from 1777 to 0000 without affecting the left-most 2 bits. We can therefore give these 2 bits any value whatever. In particular, we can set them both to the value 1 and then say that XSK i α has the effect of skipping the next instruction when it finds the number 7777, (-0), in register α. Now we can easily see how to execute any given sequence of instructions exactly n times, where $n \leq 1777$ (octal):



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

Suppose, for example, that we want to store the contents of the Accumulator in registers 350 through 357. Using register 6 to "count," we can write the short program:

Memory Address	Memory Contents		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, p. 25, which stores zero in memory registers 1400 through 1777, can be more efficiently written:

Memory Address	Memory Contents		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 27 can also be more efficiently programmed using XSK.

The SET Instruction

The second special instruction which is often used with the Index Class instructions is SET i α, code $40 + 20i + \alpha$, where α again refers directly to the first 20 (octal) 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 run the program over again the Index Registers must be reset to their initial values.

The SET instruction directs the LINC to set register α to the value contained in whatever memory register we specify. It is uniquely different from the instructions so far presented in that the instruction itself always occupies 2 consecutive memory registers, say p and p + 1:

Memory Address	Memory Contents	
p	SET i α	$40 + 20i + \alpha$
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 <u>memory address</u> for locating the value which will replace C(α). That is, register p + 1 is thought of as containing X,

Memory Address	Memory Contents		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 right-most 10 bits, the address part, of register p + 1; the left-most bit of C(p + 1) may have any value and, for the present, bit 10 must be zero.*

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

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

* See Appendix I.

The following program scans 100 (octal) 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 Contents		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 whatever, since the program itself will set them to the correct values.

Suppose we had wanted to SET two Index registers to the same value, say -100. We could write either:

Memory Address	Memory Contents		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	C(11), i.e., -100, → C(12).
23	11	0011	

We 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.

10. Index Class Instructions II

Double Register Forms

The Index Class instructions have been thought of as addressing an Index Register β, $1 \leq \beta \leq 17$, 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 Contents		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 left-most bit of $C(p + 1)$ may have any value, and, for the present, bit 10 must be zero.*

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

Memory Address	Memory Contents		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, we are not referring to memory register 0. In fact, when $\beta = 0$, no reference whatsoever is necessarily made to the Index Registers. As with SET, the computer automatically takes the next instruction from register $p + 2$.

* See Appendix I.

We may now think of the Index Class instructions as having four alternative ways of addressing the memory, which depend on i and β, and which are summarized below:

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

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

* The program does not, in fact, behave exactly as described here. Can the reader find the discrepancy?

Memory Address	Memory Contents		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	6777	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 will skip to location 46 (not 45) when the skip condition is satisfied. 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 will try to interpret the second register as an instruction:

Memory Address	Memory Contents	Effect
⋮	⋮	
p	XSK i β	
p + 1	LDA i	Go to p + 1 when $X(\beta) \neq 1777$.
p + 2	3 ←	Go to p + 2 when $X(\beta) = 1777$.
⋮	⋮	

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 what might be called "detailed machine language" with the following compact but entirely adequate restatement:

1) $0 \rightarrow C(CTR)$.

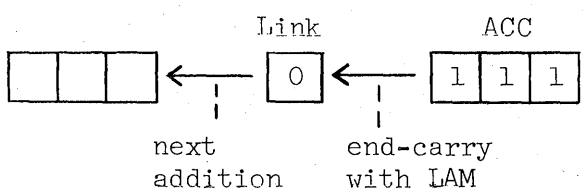
2) If $C(X) > T$ then $C(CTR) + 1 \rightarrow C(CTR)$, for X = 1350, 1351, ..., 1447.

3) $C(CTR) \rightarrow C(ACC)$.

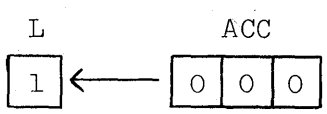
4) HALT

Multiple Length Arithmetic

An Index Class instruction, LINK ADD TO MEMORY, LAM i β, with the octal 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 11 of the Accumulator during addition is saved in the Link Bit; it is not added to bit 0 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 ones) and C(L) = 0, and we add 1 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 zero (as 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, let us 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 let us postulate 2 numbers, N_1 and N_2, each 6 bits long, which occupy a total of 4 of our 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 we must

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 <u>right</u> 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 <u>left</u> halves of N_1 and N_2. Any new carry will again replace C(L).

000	111	N_1
101	001	N_2
$\boxed{0}$	$\boxed{1}$	
110	000	$N_1 + N_2 = -17$
2nd LAM No end-carry	1st LAM End-carry	

We see upon inspection that only the first LAM produced an end-carry.

To complete the illustration we must also consider the case in which a final carry appears in the Link Bit, as in the addition of +12 and -2,

```
        001      010        +12
        111      101        - 2
      ------------------   -----
  [1]        [0]
        000      111        + 7
   |          |
2nd LAM     1st LAM
End-carry   No end-carry
```

whose sum, in ones' complement notation is 001 000, or +10 (octal), but which with LAM results in +7 and an end-carry in the Link Bit. Since ones' complement representation depends on end-around carry, we must do some extra programming to restore our result to a true ones' complement number. This is, of course, the equivalent of adding 1 to our 2-register result. Assuming that the result is in M_1 and M_2

L	M_2	M_1
1	000	111

we can again use the LAM instruction. We must first clear the Accumulator <u>without</u> clearing the Link Bit (this can be done with an STC instruction). We 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. We 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 (octal) 24-bit numbers, N_0, N_1, ..., N_{77}, are to be added term by term to 100 (octal) numbers, R_0, R_1, ..., R_{77}, such 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, ..., N_{77} are in registers 100 - 177, the right halves in 200 - 277. The left halves of R_0, R_1, ..., R_{77} are in 1000 - 1077, the right halves in 1100 - 1177. The left halves of the sums, S_0, S_1, ..., S_{77}, will replace the contents of 1000 - 1077, the right halves will replace the contents of 1100 - 1177. A "memory map" is sometimes helpful when a program must move large amounts of data around the memory:

Memory Address	Memory Contents
11 - 14	β registers used
⋮	⋮
100 - 177	Left halves N_0-N_{77}
200 - 277	Right halves N_0-N_{77}
⋮	⋮
377 - 425	Program: $N_i+R_i=S_i$
⋮	⋮
1000 - 1077	Left halves R_0-R_{77}, then S_0-S_{77}
1100 - 1177	Right halves R_0-R_{77}, then S_0-S_{77}

Memory Address	Memory Contents		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	$0 \rightarrow C(ACC)$; $0 \rightarrow C(L)$.
413	LDA i 11	1031	Right half of $N_i \rightarrow C(ACC)$.
414	LAM i 13	1233	Right half of N_i + right half of $R_i \rightarrow C(ACC)$, and → right half of R_i. End-carry → C(L).
415	LDA i 10	1030	Left half of $N_i \rightarrow C(ACC)$.
416	LAM i 12	1232	C(L) + C(ACC) + left half of $R_i \rightarrow C(ACC)$, 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 $S_i \rightarrow C(ACC)$, and → right half of S_i. End-carry → C(L).
421	STC 377	4377	Clear Accumulator.
422	LAM 12	1212	C(L) + left half of $S_i \rightarrow C(ACC)$, and → left half of S_i.
423	XSK i 14	0234	Index 14 and test for 7777.
424	JMP 412	6412	$C(14) \neq 7777$, return to form next sum.
425	HLT	0000	C(14) = 7777, so halt.

Example 13. Summing Sets of Double Length Numbers Term by Term.

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 will always contain 0 after the computer executes the last LAM in location 422. Register 377 is used simply as a "garbage" register so that we can clear the Accumulator without clearing the Link Bit.

Multiplication

Another Index Class instruction which needs special explanation is MULTIPLY, MUL i β, code 1240 + 20i + β. 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 ones' complement numbers, and the sign of the product is left in both the Accumulator (bit 11) 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 left-most bit (bit 11) of register β is used to specify the form of the numbers.

When bit 11 of register β contains zero, the numbers are treated as integers; that is, the binary points are assumed to be to the right of bit zero of the Accumulator and the specified memory register. Given C(ACC) = -10, C(β) = 400 (bit 11 of register β = 0), and C(400) = +2, then the instruction MUL β will leave -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 <u>sign</u> of the product is correct, and that the overflow effectively occurred from bit <u>10</u>, not from bit 11.

When bit 11 of register β 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 11 and bit 10) of the Accumulator and the specified memory register. Given C(ACC) = +.2, C(β) = 5120 (bit 11 of register β = 1), and C(1120) = +.32, then execution of MUL β will leave +.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 right-most, or <u>least</u> significant, 11 bits of this product are left with the proper sign in the Accumulator, and for fractions the <u>most</u> significant 11 bits of the product are left with the proper sign in the Accumulator. If, for example,

```
C(ACC) =                    001100000000
   and     binary points ---->            <---- binary points
           for fractions ---->            <---- for integers
  C(M) =                    000010000000
```

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

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

and if bit 11 of register β contains 1, the most significant 11 bits with the proper sign, will be left in the Accumulator:

$$\begin{aligned} C(ACC) &= 0.\underbrace{000}\ \underbrace{001}\ \underbrace{100}\ 00 \\ (+.3)x(+.04) &= +.\ 0\quad 1\quad 4 \end{aligned}$$

Had bit 11 of register β contained zero, 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.

The use of bit 11 of register β is new to our concept of Index Registers and should be noted in connection with the four memory addressing alternatives which the Index Class instructions employ. When $\beta \neq 0$ then

bit 11 of $C(\beta)$, that is, bit 11 of the register which contains the memory address, is used. The same is true when $i = 0$ and $\beta = 0$, as in:

Memory Address	Memory Contents	
p	MUL	1240
p + 1	h;X	4000h + X

That is, bit 11 of $C(p + 1)$, the register containing the memory address, is used. We sometimes call this bit the h-bit, whether in an Index Register or in register p + 1. When, however, $i = 1$ and $\beta = 0$, it will be recalled that p + 1 is itself the memory address:

Memory Address	Memory Contents	
p	MUL i	1260
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 $\pm .0176$, that is, whose most significant five bits after the sign (bits 6-10) 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 Contents		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 zeros or ones, depending on the sign, into the low order 5 bits of the Accumulator. Since this amounts to a "scale left" operation, it thereby 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.

* See also Appendix III.

Memory Address	Memory Contents		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	0063	Set addresses for storing products.
1401	1077	1077	
1402	SET i 4	0064	
1403	1177	1177	
1404	SET i 5	0065	Set 5 to address multiplier as fraction.
1405	4000+777	4777	
1406	SET i 6	0066	Set 6 to address multiplier as integer.
1407	777	0777	
1410	SET i 7	0067	
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 0 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 11 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	
1424	BCL i	1560	Clear bit 11 of right half.
1425	4000	4000	
1426	BSE i	1620	C(bit 0 of left half) → C(bit 11 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.

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.

There are two remaining Index Class instructions, SKIP ROTATE, SRO i β, and DISPLAY CHARACTER, DSC i β, which will be discussed later in connection with programming the oscilloscope display.

11. 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. We speak of LH(X), meaning the contents of the left 6 bits of register X, and RH(X), meaning the contents of the right 6 bits. We can then think of C(X) = LH|RH, or C(X) = 100LH+RH.

Half-word instructions always use the <u>right half</u> 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 11, the h-bit, of register β.

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

Memory Address	Memory Contents		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 Contents		Effect
40	LDH	1300	Since h = 1, RH(500), i.e., 76,
41	1;500	4500	→ RH(ACC). 0 → LH(ACC).
⋮	⋮	⋮	
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, we can write:

Memory Address	Memory Contents		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 <u>left</u> half of C(1001) is affected. If, for example, C(ACC) = 5017, then 17 replaces LH(1001), and the contents of register 1001 become 1715.

SKIP IF HALF DIFFERS, SHD i β, 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) = 5671, and we write:

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

then the computer will skip because RH(376), i.e., 52, ≠ RH(ACC), or 71. Had we written 376 at location 400, that is, h = 0, then 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 <u>half</u>-words, the computer adds 4000 to C(β) with end-around carry. This has the effect of complementing h(β) <u>every</u> time register β is indexed, and stepping X(β) every <u>other</u> time. Suppose, for example, that our 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 will first add 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) will be indexed to 4400 and the Accumulator will be

loaded from RH(400). Continuing then, 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, we may describe the memory address, when i = 1, as

$$\bar{h};X+h$$

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

$\bar{h}$;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 β ≠ 0	LDH β	Single Register	Register β holds half-word operand address.
2	i = 1 β ≠ 0	LDH i β	Single Register	First, index register β by 4000 with end-around carry. Then, register β holds half-word operand address.
3	i = 0 β = 0	LDH h;X	Double Register	Second register holds half-word operand address.
4	i = 1 β = 0	LDH i 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.

12. The KEYBOARD Instruction

Before continuing with Half-Word Class programming examples, the KEYBOARD instruction, KBD i, code 515 + 20i, is introduced. The LINC uses a simple, externally-connected keyboard for coded input. Each key has a 6-bit code number, 0-55 (octal), (See Chart II), which can be transferred into the Accumulator by the KBD i instruction when a key is struck. KBD i directs the LINC to clear the Accumulator, copy into the right half of the Accumulator the code number of the struck key, and release the key. 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 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.

This 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 (octal), is struck, which stops the program.

Memory Address	Memory Contents		Effect
7	[h;X]	[-]	Half-word index register.
⋮	⋮	⋮	
→ 20	SET i 7	0067	Set index register 7 to one half-word location less than initial location.
21	1;77	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 16. Filling Half-Word Table from the Keyboard.

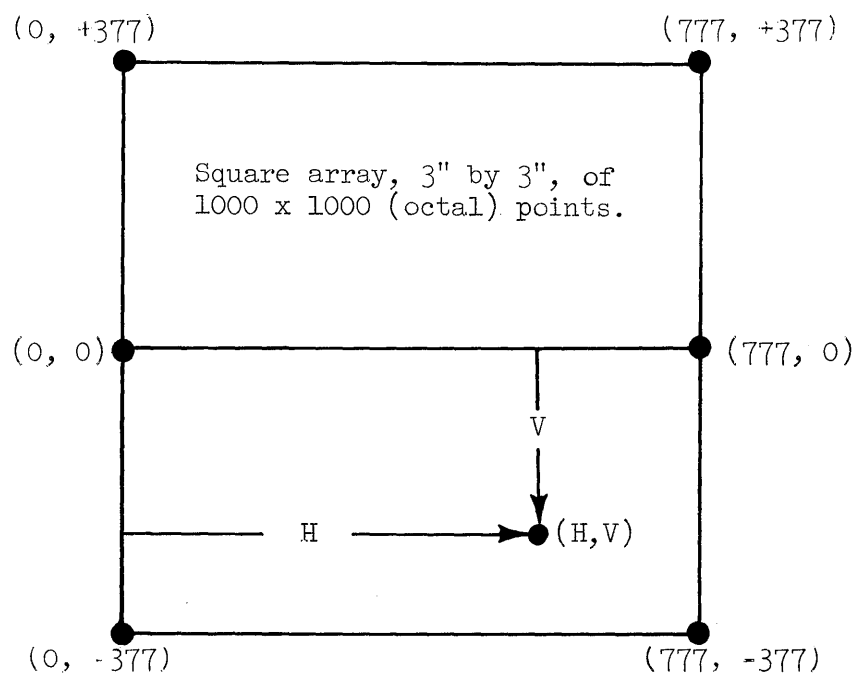
Another example reads key code numbers and stores at consecutive half-word locations only those code numbers which represent the letters A through Z, codes 24 - 55 (octal). Other key codes are discarded, and the program stops when 100 (octal) letters have been stored.

Memory Address	Memory Contents		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	1;106	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	

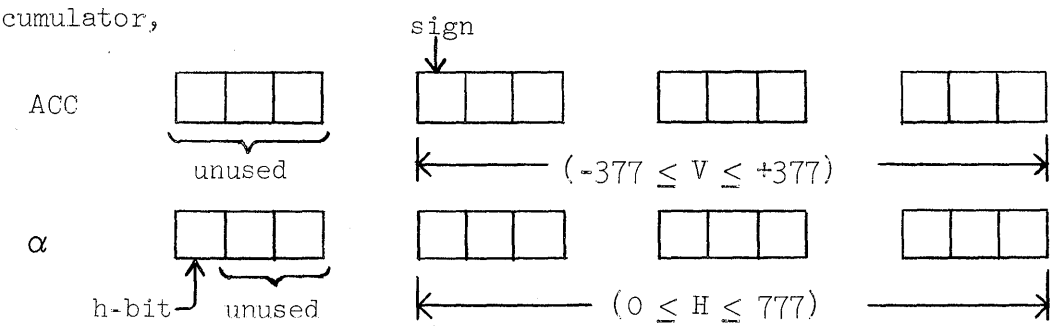
Example 17. Selective Filling of Half-Word Table from the Keyboard.

13. The LINC Scopes and the Display Instructions

The LINC has two cathode ray tube display devices called Display Scopes, each of which is capable of presenting a square array of 512 by 512 (decimal) spots (1000 by 1000, octal). A special instruction, DISPLAY, DIS i α, code 140 + 20i + α, 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 α. The vertical coordinate, $-377 \leq V \leq +377$ (octal), is held in the Accumulator during a DIS i α instruction; the horizontal coordinate, $0 \leq H \leq 777$ (octal), is held in register α, $0 \leq \alpha \leq 17$. The spot in the lower left corner of the array has the coordinates (0, -377):



The coordinates are held in the right-most 9 bits of register α and the Accumulator,



so that if C(ACC) = 641, i.e., -136, and C(5) = 430, then DIS 5 will cause a spot to be intensified at (430, -136) on the scope(s).

Both scopes are positioned at the same time. The production of a bright spot on either scope depends upon the state of the left-most bit (the h-bit) of register α and an external channel selector located on the face of each 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. Either Display Scope may be manually set to intensify Channel #0, Channel #1, or both.

The i-bit in DIS i α is used in the usual way to specify whether to index the right 10 bits of register α 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(s) via Display Channel #0:

Memory Address	Memory Contents		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 right-most 10 bits) and display. Repeat endlessly.
24	JMP 23	6023	

Example 18. Horizontal Line Scope Display.

Another example displays as a curve the values found in a set of consecutive registers, 1400 through 1777. The vertical coordinates are the most significant 9 bits of each value. Since we have only 400 (octal) points to display, the curve will be positioned in the middle of the scope. Channel #1 is used.

Memory Address	Memory Contents		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 19. Curve Display of a Table of Numbers.

Character Display

The 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.

If, for example, we want to display the letter A, the array on the scope might look like:

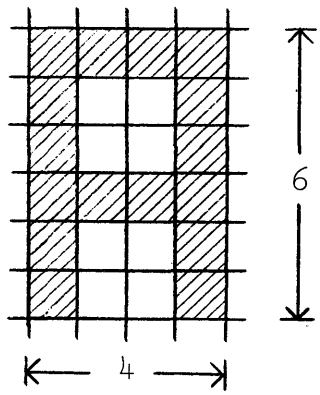


fig. a

5	11	17	23
4	10	16	22
3	9	15	21
2	8	14	20
1	7	13	19
0	6	12	18

fig. b

where the shaded areas of fig. 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).

We 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 (decimal) registers to hold both coordinates for all the points which are intensified. Instead we arbitrarily decide upon a scope format, say 4 x 6, and make up a pattern word in which <u>ones</u> represent points to be intensified and <u>zeros</u> points which are not intensified. To specify a 4 x 6 pattern of 24 bits we need 2 memory registers. We also decide, for efficiency of programming, to display the points in the order shown numerically in fig. b, that is, from lower left to upper

right, column by column. If we examine bit 0 of the pattern word first, bit 1 next, bit 2, etc., then the pattern word for the left half of the letter A (the left two columns) will look like:

	11	10	9	8	7	6	5	4	3	2	1	0
First pattern word	1	0	0	1	0	0	1	1	1	1	1	1

where the bit positions of the pattern word correspond to the numbered scope positions 0 - 11 of fig. b. The pattern word for the right half of the letter will then look like:

	11	10	9	8	7	6	5	4	3	2	1	0
Second pattern word	1	1	1	1	1	1	1	0	0	1	0	0

with bits 0 - 11 corresponding to scope positions 12 - 23 respectively.

An Index Class instruction, SKIP ROTATE, SRO i β, 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 0 of the specified memory register contains 0. If bit 0 contains 1, the computer does not skip. In either case, however, <u>after</u> examining bit 0, 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 0, then bit 1, bit 2, etc. Executing the SRO instruction 12 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 Contents		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 0 of pattern word is zero. Rotate the pattern word 1 place to right.
67	DIS 6	0146	If bit 0 of pattern word was one, 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 0 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 if 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 20. Character Display of the Letter A.

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.

DISPLAY CHARACTER, DSC i β, 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 0 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 will be displayed at H = 370, the second at H = 374, and register 1 will contain 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 right-most 5 bits (bits 0 - 4) 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. Furthermore, the right-most 5 bits of the Accumulator always contain 30 (octal) at the end of a DSC instruction, so that if the initial C(ACC) = +273, the initial V will equal +240 and C(ACC) will equal +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 p. 62 (Example 20) does, but with greater efficiency using the DSC instruction. Since we cannot have an initial V = -10 with DSC, the program uses V = 0.

Memory Address	Memory Contents		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	0;364	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 21. 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, we could, of course, display an entire row of characters.

The following program repeatedly displays a row of 6 digits. The pattern words for the characters 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. The keyboard codes for the characters to be displayed are located in a half-word table from 1400 through 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 Contents		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	Increase H by 4 to provide space between characters.
36	4	0004	
37	ADM	1140	
40	1	0001	
41	XSK i 2	0222	Index X(2) and check to see whether 6 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 22. Displaying a Row of Characters.

Suppose, for example, that one of the 6 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 x 2 = 16 octal) 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 we could add pattern words for all the keyboard characters to our pattern table; if we organize 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.*

14. 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 (numbered 0 - 17 octal) 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 through 7 are slow speed inputs built to receive signals in the range -1 to -7 volts at a maximum frequency of 200 cycles per second. These eight lines are equipped with potentiometers, appearing on the Display panel as numbered black knobs, whose voltage is varied by turning the knobs. Lines 10 through 17, located at the Data Terminal module, are for high frequency signals which may range from -1 to +1 volts at a maximum of ~ 20,000 cycles per second.

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 ±177, leaving the result in the Accumulator. Thus, for example, a voltage of zero 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,

* See Chart III.

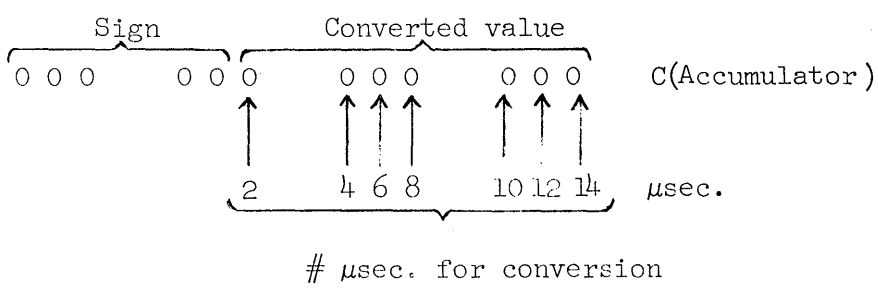
cause +177 (octal) to be left in the Accumulator. Voltages less than or equal to -1V will cause -177 to be left in the Accumulator.

Memory Address	Memory Contents		Effect
→ p	SAM i 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.

The i-bit in the SAMPLE instruction can be used to shorten the length of time the instruction requires, occasionally with some sacrifice of precision. When i = 0, the SAMPLE instruction lasts 24 μsec.* and the conversion is completed for all bits of the Accumulator (through bit 0). When i = 1, however, the computer proceeds to the next instruction in sequence after only 8 μsec. and before the conversion process is finished. The conversion is not, however, terminated. It will continue in the Accumulator for 14 more μsec. while the computer executes succeeding instructions. If the Accumulator is not disturbed during this time, the correct converted value will be accessible after 14 μsec. If the Accumulator is disturbed, however, the converted value in the Accumulator after 14 μsec. will be incorrect.

During the 14 μsec. one bit is converted every 2 μsec., beginning with the most significant conversion bit (bit 6) of the Accumulator:



* See Appendix II: LINC Order Code Summary, for instruction execution times.

Suppose that the instruction following a SAM i n when i = 1 is STC, Store-Clear. During execution of an STC instruction the contents of the Accumulator are stored in the memory 10 μsec. after the STC instruction is initiated. The low order 3 bits (bits 2, 1, and 0, converted after 10, 12, and 14 μsec.) will not be converted by this time, and should therefore be disregarded. Furthermore, the STC instruction may not leave the Accumulator clear, because the conversion process will continue for 4 μsec. after the clear time of the STC instruction. In general, examination of the Instruction Timing Diagrams[4] will show when it is feasible to use SAM with i = 1.

To illustrate the use of this instruction, we 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 every 56 μsec. (This is determined by adding the execution times for SAM i, 8 μsec.; DIS, 32 μsec.; and JMP 1002; 16 μsec.)

Memory Address	Memory Contents		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 i 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 23. Simple Sample and Display.

Note that since here we want a continuous display, 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 (decimal) word segment of memory registers 0 through 1777. The 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 Contents		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 200 to make the value positive, rotate left 2 places to produce an address for display, and store in register 12.
23	ADA i	1120	
24	200	0200	
25	ROL 2	0242	
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 (decimal) 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 24. Moving Window Display Under Knob Control.

At locations 23 - 25 a memory address is computed for the first vertical coordinate by adding 200 to the sample value. This leaves the value in the range +1 to +377; it is then rotated left 2 places to produce an initial address in the range 4 through 1774 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 are used to 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, that since we are using memory registers 1401 - 1777, the same index register (register 2) may be interpreted both as address (location 104) and counter (location 105). We do not need a separate counter because the final address (1777) will serve also as the basis of the skip decision for the XSK instruction. The same is true at locations 123 and 133.

Memory Address	Memory Contents		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	ADA i	1120	Add 1400+200 to the sample value to form an address for recording the event and store.
115	1600	1600	
116	STC 122	4122	
117	LDA i	1020	Add 1 to the contents of the register just located by the sample value to record the event.
120	1	0001	
121	ADM	1140	
122	[-]	[-]	
123	LDA i 2	1022	Index register 2 and put a histogram value in the Accumulator.
124	DIS i 3	0163	Index the H coordinate and display.
125	DIS 3	0143	Display without indexing.
126	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.
127	-1	7776	
130	SAE i	1460	
131	-400	7377	
132	JMP 125	6125	
133	XSK 2	0202	When bar is finished, check to see whether 377 values have been displayed. (X(2) = 1777?).
134	JMP 113	6113	If not, return to get next sample.
135	JMP 107	6107	If so, return to reset vertical coordinate address, H coordinate, and repeat.

Example 25. Histogram Display of Sampled Data.

15. 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. The 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.

```
                        condition
                SKP     0 ≤ c ≤ 13
                 ↓        ↓
SKP i c:        440 + 20i + c
                          ↑
                          |   { i = 0:  Skip only if condition c is met
                          |   {          or level n is negative.
     or                   |   {
                          |   { i = 1:  Skip only if condition c is not
                          ↓   {          met or level n is not negative.
SXL i n:        400 + 20i + n
                 ↑        ↑
                SXL     0 ≤ n ≤ 13
                        level line number
```

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)

The SKP i c instructions test 13 conditions, which, because of their variety, we choose to describe 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 11 = 0):

Case: i = 0

Memory Address	Memory Contents		Effect
p	APO	440 + 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 Contents		Effect
p	APO i	440 + 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 right to left, 0 through 5.)

The SXL i n instruction, SKIP ON NEGATIVE EXTERNAL LEVEL, checks for the presence of a -3 volt 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 KEY STRUCK, KST i, code 415 + 20i, checks whether a keyboard key has been struck (and not yet released). 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 (octal), for a set of 1000 (octal) samples appearing on input line #13. The number of peaks exceeding the threshold will be left in the Accumulator.

Memory Address	Memory Contents		Effect
7	[-n]	[-]	Counter for 1000 samples.
10	[n]	[-]	Counter for number above 100 (octal).
⋮	⋮	⋮	
→ 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 at 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 26. 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 (octal), is struck on the keyboard.

Memory Address	Memory Contents		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 (octal) 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 (octal). Return and continue sampling.
113	HLT ←	0000	If so, the value was 24 or greater; halt.

Example 27. Simple Sample and Display with Keyboard Control.

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

16. The Data Terminal Module and the OPERATE Instruction

We have already mentioned the OPERATE instruction (p. 55) in connection with KBD i. In general, OPERATE, OPR i n, code 500 + 20i + n, provides operating and synchronizing signals for external equipment. The number n, $0 \leq n \leq 13$ (octal) refers to one of twelve Operate Level lines sent to the Data Terminal Module, as well as to one of the twelve External Level lines (mentioned under SXL).

During the execution of an OPR instruction a negative output level is supplied on Operate Level line n 4 μsec. after the beginning of the instruction;[4] it remains for the duration of the instruction. The i-bit is used to direct the LINC to pause. If i = 0, there is no pause. If i = 1, the LINC pauses 4 μsec. after the beginning of the instruction and sends a "Beginning of Operate Pause" pulse, BEOP, 0.4 μsec. duration, to the Data Terminal module to signal that the pause has begun. The computer then waits in this state until a negative input signal is sent back on External Level line n. This signal automatically restarts the computer.

For example, execution of the instruction OPR i 6, code 526, provides an output signal on Operate Level line #6 and directs the LINC to pause, permitting an external device associated with line #6 to be synchronized with computer operation. Then when the external device is ready or has completed its operation, it in turn supplies a negative signal on External Level line #6, which restarts the computer.

In addition to the possible BEOP pulse, two other 0.4 μsec. pulses are sent to the Data Terminal module regardless of whether the computer has paused or not. The first, called OPR2.1, occurs 6 μsec. after the beginning of the instruction if there is no pause. If the computer has paused, the OPR2.1 pulse, which indicates that the computer is now running, will appear not less than 2 μsec. and not more than 4 μsec. after the restart signal is delivered by the external equipment over line n. The second pulse, OPR2.2, occurs 2 μsec. after OPR2.1.

The OPR instruction may be used in a variety of ways depending on need and the type of external equipment involved. It can be used simply to sense the occurrence of an event (such as an external clock pulse), or it can be used to control the transfer of digital information between the LINC and external equipment (such as a tape recorder). In this context the user has the option of transferring a single word (12 bits) either in or out of the LINC Accumulator or Memory Contents register, or he can choose to transfer a group of words directly into or out of the LINC memory. Various enabling levels supplied by the user at the Data Terminal module define the path and type of information transfer.

The Keyboard is a good example of a simple external device which is controlled by an Operate instruction, OPR i 15, synonymous with KBD i. The number 15 designates special external level and operate level lines, with which the Keyboard is permanently associated.

17. 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. We can picture the general structure of such a program as follows:

Main Program

Memory Address	Memory Contents
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 Contents	
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, we must be able to change the JMP instruction at location 1020 so that the first time the subroutine is used it will return to the main program with a "JMP 151" and the second time with 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 zero. (See page 14.) Thus, when the "JMP 1000" is executed at location 150, a "JMP 151" is automatically stored in register 0, thereby saving the return point for the subroutine. The subroutine might retrieve this information in the following way:

Subroutine:

	Memory Address	Memory Contents	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.

Clearly, a simple "JMP 0" at location 1020 will suffice when the subroutine does not, during its execution, destroy the contents of register zero. 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,

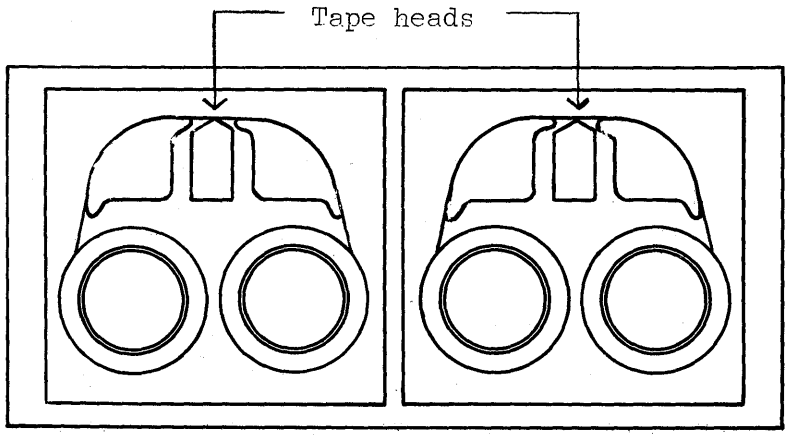
using the SET instruction, is possible when there is an available β register:

	Memory Address	Memory Contents	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.

18. Magnetic Tape Instructions

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

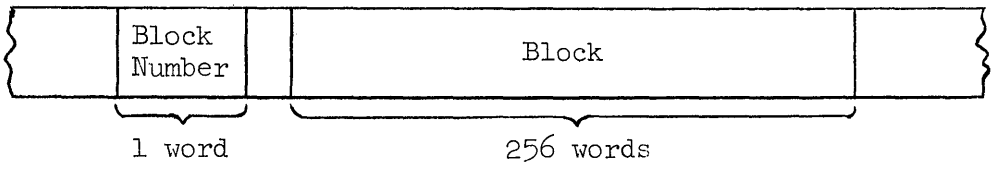
LINC MAGNETIC TAPES



Tape Unit #0 Tape Unit #1

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 <u>one</u> unit, however, is ever used at one time.

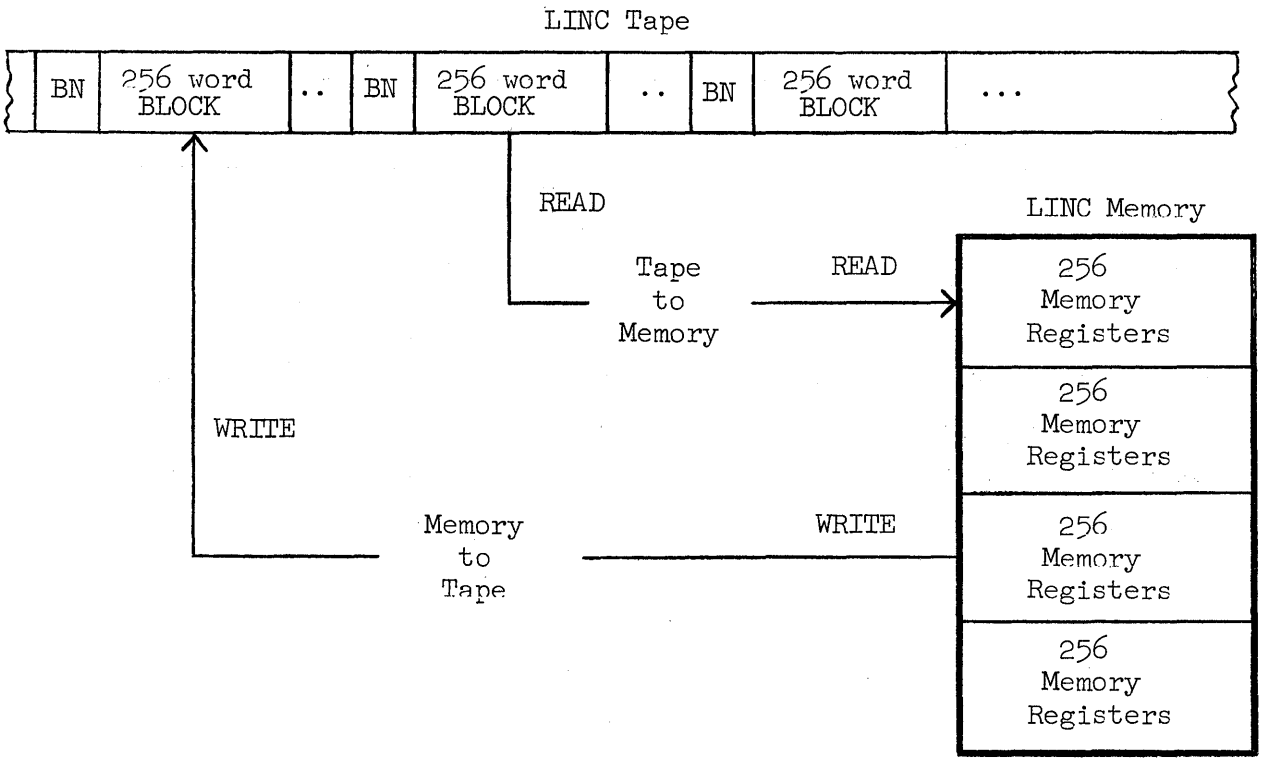
A LINC tape can hold 131,072 12-bit words of information, or the equivalent of 128 (decimal) full LINC memories. It is, however, divided into 512 (decimal) smaller segments known as blocks, each of which contains 256 (decimal) 12-bit words, a size equal to one-quarter of a LINC memory. Blocks are identified on any tape by block numbers, 0 through 777 (octal); Magnetic 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, we may look more specifically at one of the Magnetic Tape instructions, READ TAPE, RDE i u.

Block Transfers and Checking

READ TAPE is one of six Magnetic Tape instructions which copy information either from the tape into the LINC Memory (called READING), or from the memory onto the tape (called WRITING). These are generally called <u>block transfer</u> instructions because they transfer one or more blocks of information between the tape and the memory:



All of the Magnetic Tape instructions are double register instructions. RDE, typical of a block transfer instruction, is written:

Memory Address	Memory Contents	
p	RDE i u	702 + 20i + 10u
p + 1	QN|BN	1000QN + BN

The first register of the instruction has two special bits. The u-bit (bit 3) 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 inches per second. 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 4) 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, when we want to execute several Magnetic Tape instructions in succession. If we let it stop we will have 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 right-most 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 <u>quarter number</u>, 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 through 7,* and refer to the memory registers as follows (numbers are octal):

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

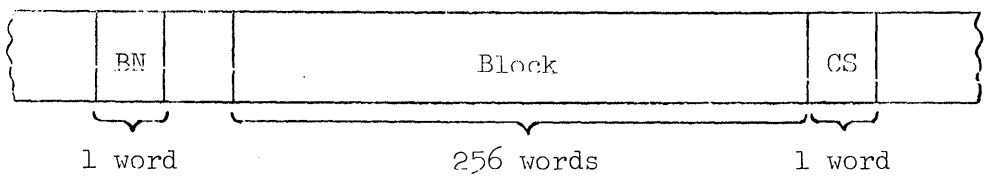
Suppose, for example, we want to transfer data stored on tape into memory registers 1000 - 1377. The data are in, say, block 267 and the tape is mounted on Unit #1:

Memory Address	Memory Contents		Effect
→ 200	RDE u	0712	Select Unit #1;
201	2|267	1000x2 + 267	C(block 267) → C(quarter 2).

This instruction will start to move the tape on Unit #1 if it is not already moving. It will then READ block 267 on that tape into quarter 2 of memory and stop the tape when the transfer is completed. The computer will go to location 202 for the next 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, <u>writing</u> affects only the tape and the Accumulator; the memory is left unchanged.

* See Appendix I.

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



The check sum, a feature common to many tape systems, is used to check the accuracy of the transfer of information to and from the tape. On a LINC tape the check sum 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, WRITE TAPE, WRI i u. This instruction writes the contents of the specified memory quarter in the specified block of the selected tape:

Memory Address	Memory Contents	
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 check sum is left in the Accumulator and the computer goes to p + 2 for the next 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 check sum 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 check sum from the tape is then added to this data sum and the result, called the transfer check, is left in the Accumulator. Clearly, if the information has been transferred correctly, the data sum will be the complement of the check sum, and the

transfer check will equal -0 (7777). We say that the block "checks." Thus, by examining the Accumulator after an RDE instruction, we can tell whether the block was transferred correctly. The following sequence of instructions does this and reads block 500 again if it does not check:

Memory Address	Memory Contents		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) = 7777, 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. READ AND CHECK, RDC i u, 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 instruction at p + 2. The RDC instruction is written:

Memory Address	Memory Contents	
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 we are given a tape, for example, which has in block 300 a program we want to run. We

are told that the program is 100 (octal) registers long starting in register 1250. We can mount the tape on either unit and then set and execute either RDE or RDC in the Left and Right Switches. If we use RDE, we 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 we mount the tape on Unit #1 and want to use RDC, the toggle switches should be set as follows:

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. We then start at 1250 from the console, using the Right Switches.

The remaining block transfer instructions will be described later.

A non-transfer instruction, called CHECK TAPE, CHK i u, 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 check sum 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 the other instructions:

Memory Address	Memory Contents	
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.

	Memory Address	Memory Contents		Effect
Start →	200	CLR	0011	Store zero 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	

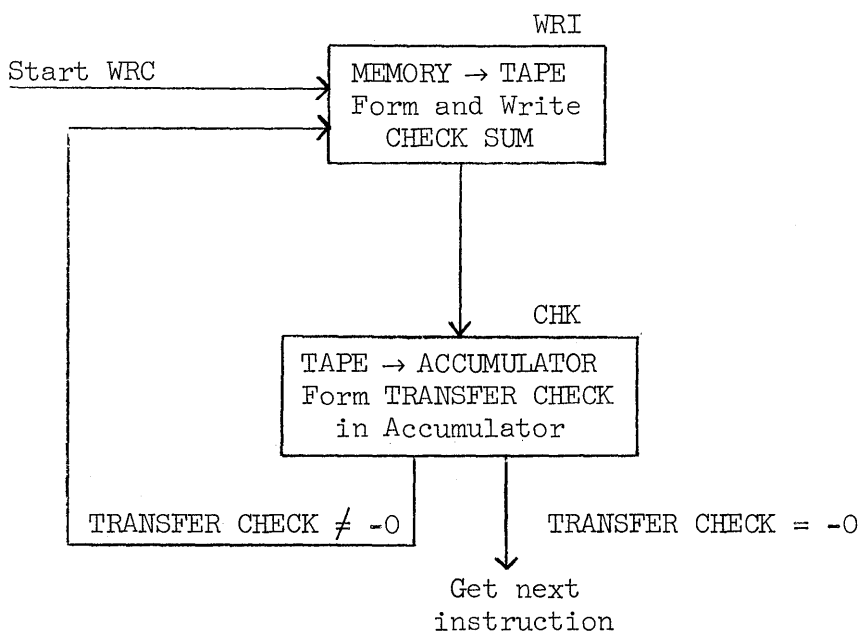
Example 28. Simple Check of an Entire Tape.

Note that the tape is left moving whenever the computer halts. This is generally undesirable, since it must then be stopped manually by the user at the console. Another tape instruction, MTB, can be used to avoid this situation, as will be shown in program example 33.

A block transfer instruction WRITE AND CHECK, WRC i u, 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 check sum in the Accumulator and writes the check sum on 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 instruction at p + 2. WRC is written:

Memory Address	Memory Contents	
p	WRC i u	704 + 20i + 10u
p + 1	QN|BN	1000QN + BN

This process of WRITE AND CHECK 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 they are 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, and 3 (registers 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 Contents		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	0720	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	0700	C(block 203) → C(quarter 3); transfer is checked and the tape stops.
107	3|203	3203	
110	JMP 400	6400	Jump to the new section.
⋮	⋮	⋮	
400	→ [-]	[-]	

Example 29. 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 Contents		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 (octal) 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 transfer. 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	
1023	HLT	0000	Halt the computer.

Example 30. Collecting Data and Storing on Tape.

Since the program saves itself when finished, the user 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.

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 READ AND CHECK GROUP, RCG i u, we write:

Memory Address	Memory Contents	
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 right-most 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 left-most 3 bits (usually QN) are used to specify the number of additional blocks to transfer. In the above example then, block 51 is read into quarter 1, and 2 additional blocks are also transferred: block 52 into quarter 2 and block 53 into quarter 3.

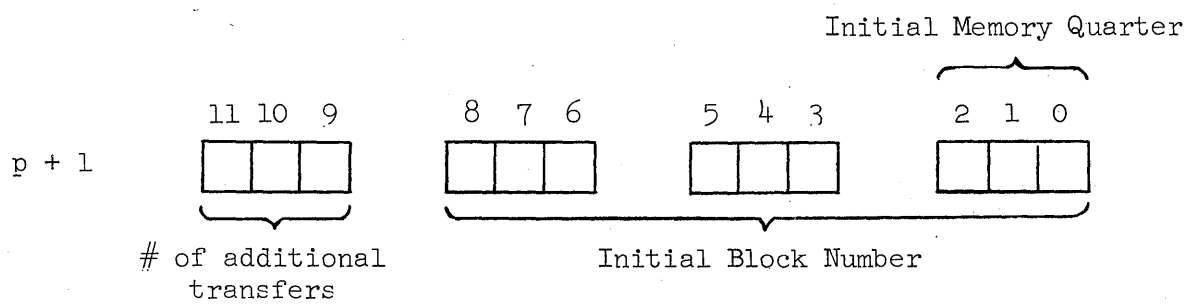
The format for WCG i u, WRITE AND CHECK GROUP, is exactly the same as for RCG:

Memory Address	Memory Contents	
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 left-most 3 bits are zero, that is "do zero 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, 341 and memory quarters 6, 7, 0, and 1, that is, quarter 0 succeeds quarter 7.* The transfers 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 check sums 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, the group transfer instructions leave -0 in the Accumulator and go to p + 2 for the next instruction.

* See Appendix I.

Using RCG instead of RDC, the program example on p. 90 can be more efficiently written:

Memory Address	Memory Contents		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.

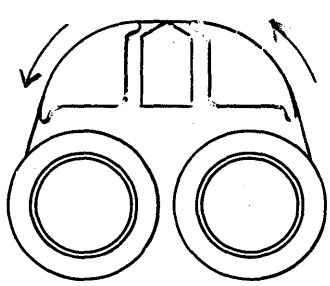
Example 31. Tape and Memory Exchange with Group Transfer.

Tape Motion and the MOVE TOWARD BLOCK Instruction

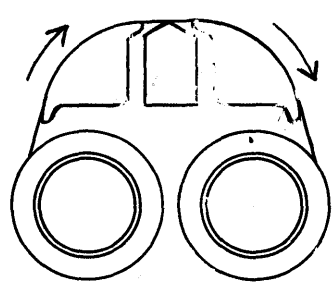
When the computer is searching the tape for a required block, it looks at each block number in turn until it finds the correct one. Since the tape may be positioned anywhere when the search is begun, it must be able to move either forward or backward to find the block.

By <u>forward</u> is meant moving from the low block numbers to the high numbers; physically the tape moves onto the lefthand reel.

Forward



Backward

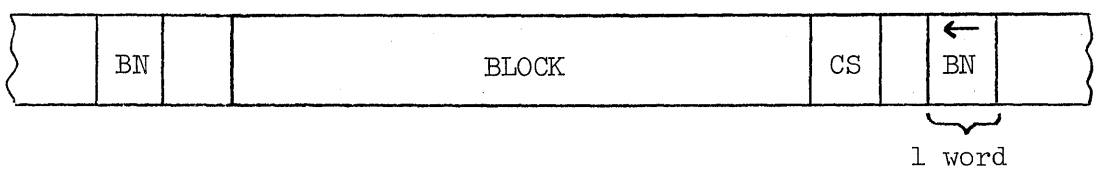


By <u>backward</u> is meant from the high numbers to the low; the tape moves onto the righthand reel.

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 and 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 <u>backward block number</u>, $\overleftarrow{BN}$, follows the check sum 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 <u>forward</u> 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 $\overleftarrow{BN}$ 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 the direction of motion until the 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.

For all Magnetic Tape instructions all 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 the delay in programs which make repeated references to the same block.

The last Magnetic Tape instruction illustrates some of the tape motion characteristics. MOVE TOWARD BLOCK, MTB i u, is written:

Memory Address	Memory Contents	
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 zero, 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. MTB i does not actually <u>find</u> 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.

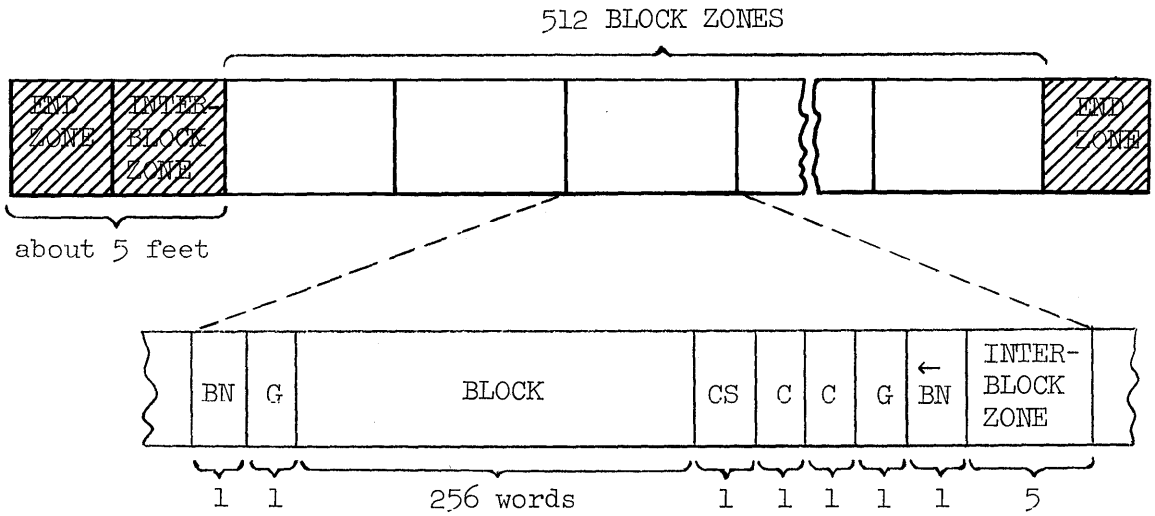
The MOVE TOWARD BLOCK instruction serves not only to identify tape position, but also can be used to save time. If, for example, a program must read block 700, and then, at some later time, write in block 50, it is efficient to have the tape move toward block 50 in the interim while the program continues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

Memory Address	Memory Contents		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 we may not be interested in the number left in the Accumulator after executing the MTB at location 102, the MTB does serve to reverse the tape. Then, when block 50 is called for, the delay in finding it will not be so long.

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 <u>end zone</u> 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 they affect tape timing. Words marked C are used by the computer 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 we must read block 500 into quarter 2; assume it does not matter when we read it in as long as we do so 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 at an inter block zone, the main program will then continue without having to wait for a block number to appear. For purposes of simplicity let us 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 will work only if the tape is stopped by the RDC instruction in register 32. If we do not stop the tape 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 tells us whether the transfer has been made or not, is necessary to guarantee that the transfer will be made before we get to location 650. At this point, if the transfer has not been made, the "JMP 32" at location 647 will be executed.

Memory Address	Memory Contents		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 $\overleftarrow{\text{BN}}$ 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 zero 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 32. Block Search Subroutine.

Tape Motion Timing

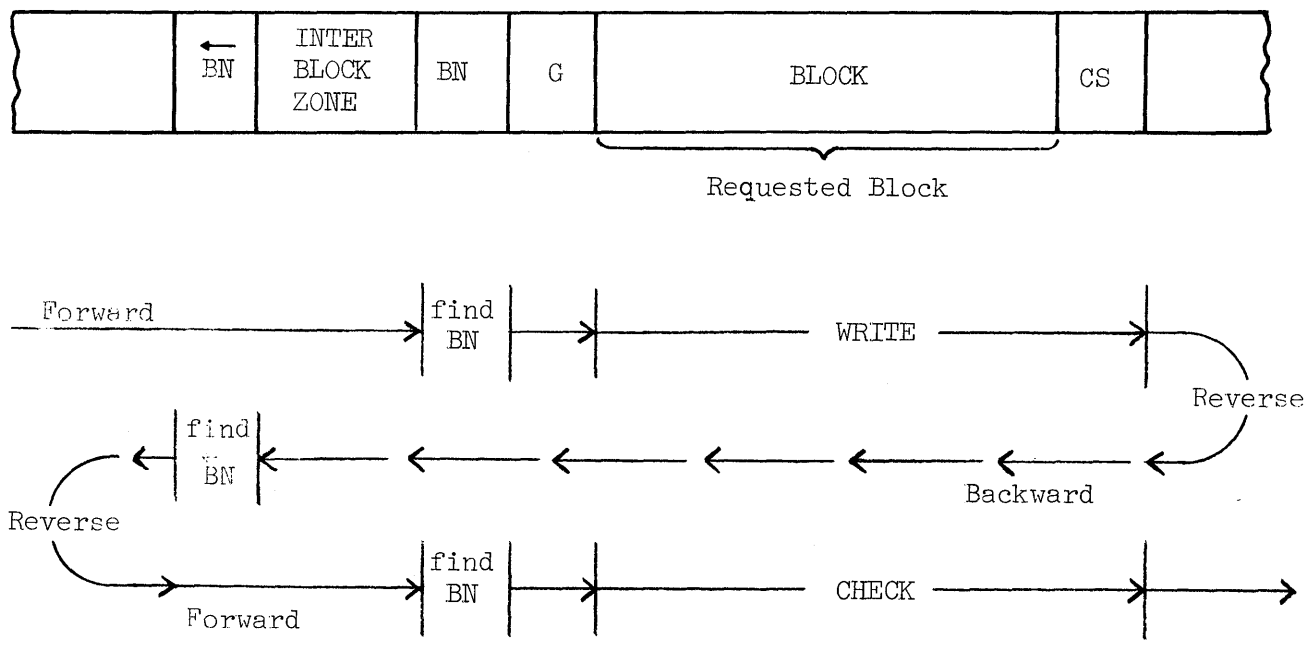
When a tape is moving at a rate of 60 ips, it takes approximately 43 msec. to move from one forward block number to the next, or 160 μsec. per word. The following table summarizes some of the timing factors:

LINC TAPE MOTION TIME	
START (from no motion to 60 ips)	approx. 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 costly in this respect. It first

must find and write in the block in the forward direction, then the tape must reverse and go backward until it is below the block, 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. Suppose we

want to 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 Contents		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	0727	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 33. 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. Clearly, 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 reduce 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 1,440 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.

Chart I. Classes of LINC Instructions

Miscellaneous
HLT
ZTA
ENI
CLR
MSC 13
ATR
RTA
NOP
COM

Shift
ROL i n
ROR i n
SCR i n

Full Address
ADD X
STC X
JMP X

Index
LDA i β
STA i β
ADA i β
ADM i β
LAM i β
MUL i β
SAE i β
SRO i β
BCL i β
BSE i β
BCO i β
DSC i β

Half-Word
LDH i β
STH i β
SHD i β

Skip
SXL i n
KST i
SKP i n
SNS i n
PIN i
AZE i
APO i
LZE i
IBZ i
OVF i
ZZZ i

Operate
OPR i n
KBD i
RSW
LSW

Magnetic Tape
RDC i u
RCG i u
RDE i u
MTB i u
WRC i u
WCG i u
WRI i u
CHK i u

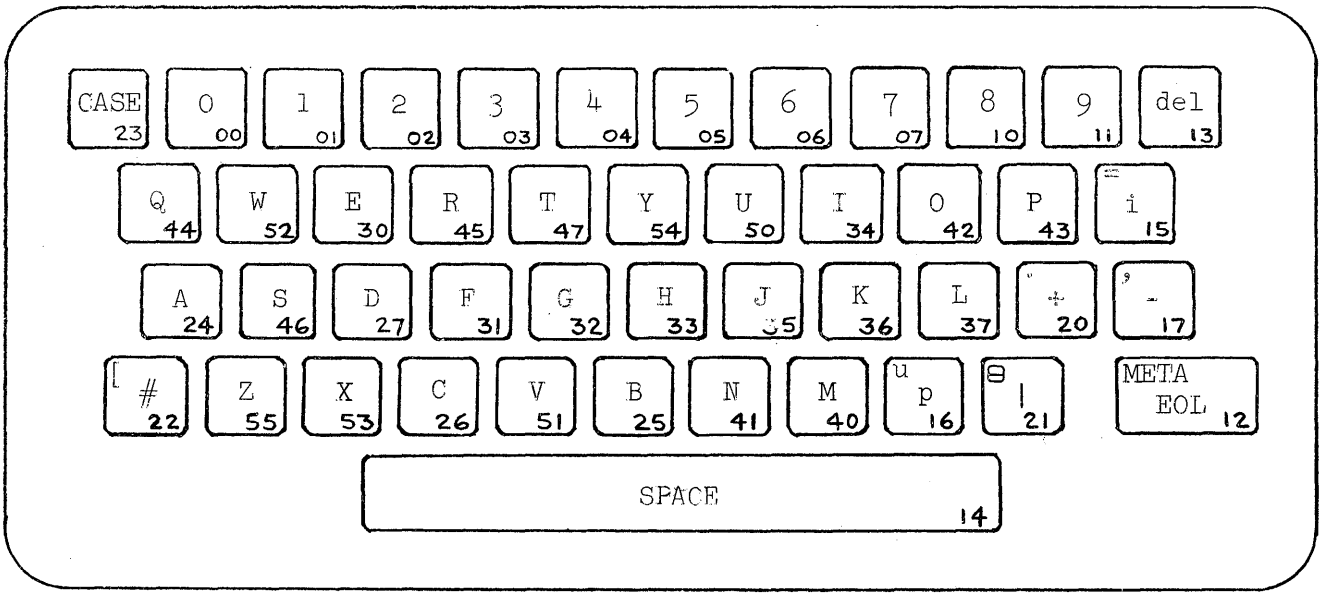
SET i α

SAM i n

DIS i α

XSK i α

Chart II. Keyboard Code



The Keyboard Code in Numerical Order

00	0	20	. / +	40	M
01	1	21	⊖ / |	41	N
02	2	22	[/ #	42	O
03	3	23	CASE	43	P
04	4	24	A	44	Q
05	5	25	B	[illegible]	R
06	6	26	C	46	S
07	7	27	D	47	T
10	8	30	E	50	U
11	9	31	F	51	V
12	META/EOL	32	G	52	W
13	delete	33	H	53	X
14	SPACE	34	I	54	Y
15	= / i	35	J	55	Z
16	u / p	36	K		
17	, / -	37	L		

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 zero.

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		

Chart IV. Instruction Code

Alphabetical

	13
ADA	1100
ADD	2000
ADM	1140
APO	451
ATR	14
AZE	450
BCL	1540
BCO	1640
BSE	1600
CHK	707
CLR	11
COM	17
DIS	140
DSC	1740
ENI	10
HLT	0
IBZ	453
JMP	6000
KBD	515
KST	415
LAM	1200
LDA	1000
LDH	1300
LSW	517
LZE	452
MSC	0
MTB	703
MTP	700
MUL	1240

NOP	16
OPR	500
OVF	454
PIN	446
RCG	701
RDC	700
RDE	702
ROL	240
ROR	300
RSW	516
RTA	15
SAE	1440
SAM	100
SCR	340
SET	40
SHD	1400
SKP	440
SNS	440
SRO	1500
STA	1040
STC	4000
STH	1340
SXL	400
WCG	705
WRC	704
WRI	706
XSK	200
ZTA	5
ZZZ	455

Numerical

0	HLT
0	MSC
5	ZTA
10	ENI
11	CLR
13	
14	ATR
15	RTA
16	NOP
17	COM
40	SET
100	SAM
140	DIS
200	XSK
240	ROL
300	ROR
340	SCR
400	SXL
415	KST
440	SKP
440	SNS
446	PIN
450	AZE
451	APO
452	LZE
453	IBZ
454	OVF
455	ZZZ
500	OPR
515	KBD

516	RSW
517	LSW
700	MTP
700	RDC
701	RCG
702	RDE
703	MTB
704	WRC
705	WCG
706	WRI
707	CHK
1000	LDA
1040	STA
1100	ADA
1140	ADM
1200	LAM
1240	MUL
1300	LDH
1340	STH
1400	SHD
1440	SAE
1500	SRO
1540	BCL
1600	BSE
1640	BCO
1740	DSC
2000	ADD
4000	STC
6000	JMP

Appendix I: Double Memory Programming

The LINC actually has two 12-bit 1024 (decimal) word memories, sometimes referred to as "lower" and "upper" memory, providing a total of 4000 (octal) words. The second, or upper, memory is addressable for data storage and retrieval; it can not, however, be used to hold running programs.

Bit 10 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:

$$\beta \quad 010 \; \underbrace{000 \quad 000 \quad 000}_{X}$$

Memory Select Bit

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, we speak of memory registers 2000 through 3777 (octal). 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 through 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 10, we can index through either memory we choose, but we 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 bit = 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 through 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 an 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 through 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 34. Indexing Across Memory Boundaries.

Appendix II

LINC Order Code Summary

Miscellaneous Class*

HLT	0000	----	HLT
HALT. Halt the computer. The Run light on the console is turned off. Perhaps the gong chimes. The computer can be restarted only from the console.			

CLR	0011	8 μsec.	CLR
CLEAR. Clear the Accumulator, the LINK bit, and the Z register.			

MSC 13	0013	8 μsec.	-
Turn on the write-gate for marking tapes if and only if the computer has been placed in the MARK mode by pressing the MARK button on the console. Warning: This instruction is to be used only for marking tapes.			

ATR	0014	8 μsec.	ATR
ACCUMULATOR TO RELAY. Copy the contents of the right half of the Accumulator (bits 0 - 5) into the Relay register. The contents of the Accumulator are not changed.			

RTA	0015	8 μsec.	RTA
RELAY TO ACCUMULATOR. Copy the contents of the Relay register into the right half of the Accumulator (bits 0 - 5) and clear the left half of the Accumulator. The contents of the Relay register are not changed.			

NOP	0016	8 μsec.	NOP
NO OPERATION. This instruction provides a delay of 8 μsec. before proceeding to the next instruction. It does nothing.			

COM	0017	8 μsec.	COM
COMPLEMENT. Complement the contents of the Accumulator.			

* See also Appendix III-6.

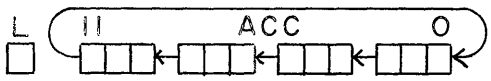
Shift Class

* Execution Times				
n (octal) $0 \leq n \leq 17$	0,1,2,3	4,5,6,7	10,11,12,13	14,15,16,17
time (decimal)	16 μsec.	24 μsec.	32 μsec.	40 μsec.

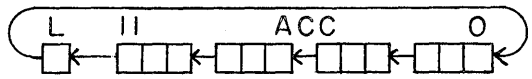
ROL i n	240 + 20i + n	*	ROL

ROTATE LEFT. Shift the contents of the Accumulator n places to the left, with or without the Link Bit. The i-bit specifies one of two variations:

i = 0



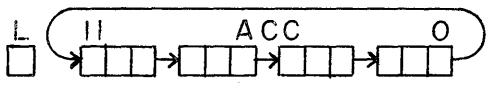
i = 1



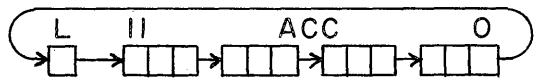
ROR i n	300 + 20i + n	*	ROR

ROTATE RIGHT. Shift the contents of the Accumulator n places to the right, with or without the Link Bit.* The i-bit specifies one of two variations:

i = 0



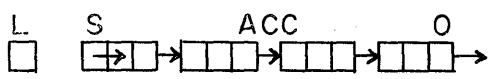
i = 1



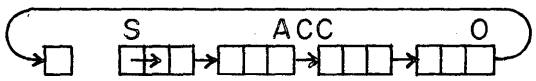
SCR i n	340 + 20i + n	*	SCR

SCALE RIGHT. Shift the contents of the Accumulator, with or without the Link Bit, n places to the right without changing the sign bit, replicating the sign in n bits to the right of the sign bit.* The i-bit specifies one of two variations:

i = 0



i = 1



* See also Appendix III.

Full Address Class

$0 \leq X \leq 1777$

ADD X	2000 + X	16 μsec.	ADD
ADD. Add the contents of register X to the contents of the Accumulator and leave the sum in the Accumulator, using 12-bit binary addition with end-around carry. The contents of register X are not changed.			

STC X	4000 + X	16 μsec.	STC
STORE AND CLEAR. Copy the contents of the Accumulator into register X and then clear the Accumulator.			

JMP X	6000 + X	*	JMP
JUMP. Set the Instruction Location register to X, i.e., take the next instruction from register X. If $X \neq 0$, and if JMP X is executed at location p, then the code number for JMP p + 1 is stored in register 0. * When $X = 0$, execution time is 8 μsec; when $X \neq 0$, 16 μsec.			

Skip Class*

Skip the next register in the instruction sequence if:

i = 0 and the specified condition is met

or if:

i = 1 and the specified condition is not met.

Otherwise, go on to the next instruction in sequence.

SXL i n	400 + 20i + n	8 μsec.	SXL
SKIP ON NEGATIVE EXTERNAL LEVEL. Condition: The signal on external level line n is -3 volts (as opposed to 0 volts). $0 \leq n \leq 13$.			

KST i	415 + 20i	8 μsec.	KST
KEY STRUCK. Condition: A key has been struck and is locked down.			

SNS i n	440 + 20i + n	8 μsec.	SNS
SENSE SWITCH. Condition: Sense Switch n is up. $0 \leq n \leq 5$.			

AZE i	450 + 20i	8 μsec.	AZE
ACCUMULATOR ZERO. Condition: Accumulator contains either 0000 or 7777.			

APO i	451 + 20i	8 μsec.	APO
ACCUMULATOR POSITIVE. Condition: The sign bit of the Accumulator is 0.			

LZE i	452 + 20i	8 μsec.	LZE
LINK ZERO. Condition: The Link bit is 0.			

IBZ i	453 + 20i	8 μsec.	IBZ
INTERBLOCK ZONE. Condition: Either tape unit is up to speed and at an interblock zone.			

* See also Appendix III-6.

Index Class

Operand Location, Y, in Index Class Instructions			
$1 \leq \beta \leq 17$		$\beta = 0$	
i = 0	i = 1	i = 0	i = 1
β Y ⋮ ⋮ → p LDA β ⋮ ⋮ Y OPERAND	β [Y-1]* ⋮ ⋮ → p LDA i β ⋮ ⋮ Y OPERAND	⋮ ⋮ → p LDA p + 1 Y ⋮ ⋮ Y OPERAND	⋮ ⋮ → p LDA i Y OPERAND ⋮ ⋮
t = 16 μsec. $0 \leq Y \leq 3777$			t = 8 μsec. Y = p + 1 $0 \leq Y \leq 1777$

* Indexing: The contents of the right-most 10 bits of register β are first indexed by 1, using 10-bit binary addition without end carry. The left-most two bits are not changed. Thus, 1777 is indexed to 0000; 3777, to 2000; 5777, to 4000; and 7777, to 6000.

LDA i β	1000 + 20i + β	(t + 8) μsec.	LDA
LOAD ACCUMULATOR. Copy the contents of register Y into the Accumulator. The contents of register Y are not changed.			

STA i β	1040 + 20i + β	(t + 8) μsec.	STA
STORE ACCUMULATOR. Copy the contents of the Accumulator into register Y. The contents of the Accumulator are not changed.			

ADA i β	1100 + 20i + β	(t + 8) μsec.	ADA
ADD TO ACCUMULATOR. Add the contents of register Y to the contents of the Accumulator and leave the sum in the Accumulator, using 12-bit binary addition with end-around carry. The contents of register Y are not changed.			

Index Class (continued)

ADM i β	1140 + 20i + β	(t + 16) μsec.	ADM

ADD TO MEMORY. Add the contents of register Y to the contents of the Accumulator and leave the sum in register Y and the Accumulator, using 12-bit binary addition with end-around carry.

LAM i β	1200 + 20i + β	(t + 16) μsec.	LAM

LINK ADD TO MEMORY. First, add the contents of the Link Bit (the integer 0 or 1) to the contents of the Accumulator and leave the sum in the Accumulator, using 12-bit binary addition with the end carry, if any, replacing the contents of the Link Bit; if there is no end carry, clear the Link Bit. Next, add the contents of register Y to the contents of the Accumulator using 12-bit binary addition with the end carry, if any, replacing the contents of the Link Bit (if no end carry arises, the contents of the Link Bit are not changed). The sum is left in the Accumulator and in register Y.

MUL i β	1240 + 20i + β	(t + 104) μsec.	MUL

MULTIPLY. Multiply the contents of the Accumulator by the contents of register Y and leave half of the product in the Accumulator. The contents of the Accumulator and register Y are treated as signed 11-bit ones' complement numbers and their full product as a signed 22-bit number.* The "h-bit," i.e., bit 11 of the register holding the address Y, specifies:

h = 0	h = 1
Integer Multiplication	Fraction Multiplication
The least significant 11 bits of the product with proper sign are left in the Accumulator.	The most significant 11 bits of the product with proper sign are left in the Accumulator.

The sign of the product is also left in the Link Bit. The contents of register Y are not changed.

If i = 1 and β = 0, use integer multiplication.

* See Appendix III.

Index Class (continued)

SAE i β	1440 + 20i + β	(t + 8) μsec.	SAE
SKIP IF ACCUMULATOR EQUALS. If the contents of the Accumulator match the contents of register Y, skip the next register in the instruction sequence; otherwise, go on to the next instruction in sequence. The contents of the Accumulator and of register Y are not changed. (See also the section on marking tapes.)			

SRO i β	1500 + 20i + β	(t + 8) μsec.	SRO
SKIP AND ROTATE. If the right-most bit of the contents of register Y is 0, skip the next register of the instruction sequence; otherwise, go on to the next instruction in sequence. In either case, rotate the contents of register Y one place to the right and replace in register Y. The contents of the Accumulator are not changed.			

BCL i β	1540 + 20i + β	(t + 8) μsec.	BCL
BIT CLEAR. For each bit of register Y which contains 1, clear the corresponding bit of the Accumulator. The contents of register Y and all other bits of the Accumulator are not changed.			

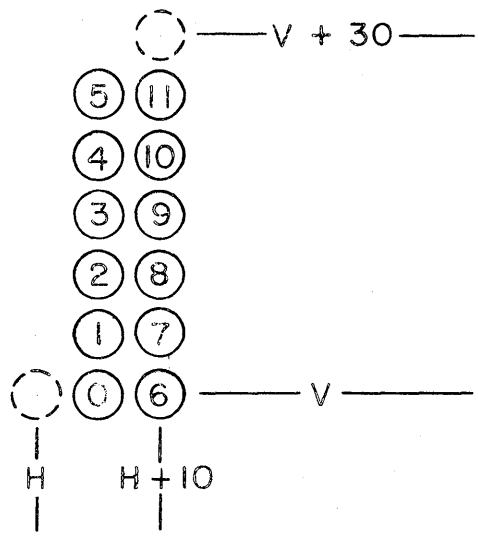
BSE i β	1600 + 20i + β	(t + 8) μsec.	BSE
BIT SET. For each bit of register Y which contains 1, set the corresponding bit of the Accumulator to 1. The contents of register Y and all other bits of the Accumulator are not changed.			

BCO i β	1640 + 20i + β	(t + 8) μsec.	BCO
BIT COMPLEMENT. For each bit of register Y which contains 1, complement the corresponding bit of the Accumulator. The contents of register Y and all other bits of the Accumulator are not changed.			

Index Class (continued)

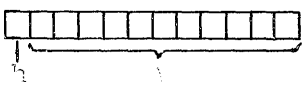
DSC i β	1740 + 20i + β	(t + 112) μsec.	DSC

DISPLAY CHARACTER. Intensify points in a 2 x 6 pattern on the Display Scope. Register Y holds the pattern word, which is examined from right to left beginning with bit 0; for each bit found to be 1 a point is intensified. Numbered points below correspond to bit positions of the pattern word:



The H coordinate is held in register 1, and bit 11 of register 1 selects the display channel. The initial contents of register 1, plus 4, is the H coordinate of point ⓪. The V coordinate is held in the Accumulator. The initial contents of the Accumulator with the right-most 5 bits (ACC_{0-4}) automatically cleared by the computer, is the V coordinate of point ⓪. Spacing between points is +4 in both horizontal and vertical directions. At the end of the instruction the value in register 1 has been augmented by 10 (octal) and bits 0 - 4 of the Accumulator contain 30 (octal). The contents of bits 5 - 11 of the Accumulator and the contents of register Y are not changed. The contents of the Z register are destroyed.

Half-Word Class

Operand Location, Y, in Half-Word Class Instructions			
$1 \leq \beta \leq 17$		$\beta = 0$	
$i = 0$	$i = 1$	$i = 0$	$i = 1$
β h;Y ⋮ → p LDH β ⋮ Y OPERAND	β $\bar{h}$;$(Y-\bar{h})$* ⋮ → p LDH i β ⋮ Y OPERAND	⋮ → p LDH p + 1 h;Y ⋮ Y OPERAND	⋮ → p LDH i Y OPERAND ⋮
t = 16 μsec. $0 \leq Y \leq 3777$ h;Y  OPERAND = LH(Y) if h = 0; RH(Y) if h = 1			t = 8 μsec. Y = p + 1 $0 \leq Y \leq 1777$ OPERAND = LH(Y)

* Indexing: $\bar{h}$ is value before indexing. The contents of register β are first indexed by 4000. Any end carry is added to the right-most 10 bits only; bit 10 is not changed. Thus: 0;1777 is indexed to 1;1777; 1;1777 to 0;0000; 0;0000 to 1;0000; 1;0000 to 0;0001. 0;3777 is indexed to 1;3777; 1;3777 to 0;2000; 0;2000 to 1;2000; 1;2000 to 0;2001. The Relay lights are probably not affected.

LDH i β	1300 + 20i + β	(t + 8) μsec.	LDH

LOAD HALF. Copy the contents of the designated half of register Y into the right half of the Accumulator. Clear the left half of the Accumulator. The contents of register Y are not changed.

STH i β	1340 + 20i + β	(t + 8) μsec.	STH

STORE HALF. Copy the contents of the right half of the Accumulator into the designated half of register Y. The contents of the Accumulator and of the other half of register Y are not changed.

SHD i β	1400 + 20i + β	(t + 8) μsec.	SHD

SKIP IF HALF DIFFERS. If the contents of the right half of the Accumulator do not match the contents of the designated half of register Y, skip the next register in the instruction sequence; otherwise, go on to the next instruction in sequence. The contents of the Accumulator and of register Y are not changed.

Operand Location, Y, in the SET Instruction	
i = 0	i = 1
α [-] ⋮ ⋮ → p SET α p + 1 Y ⋮ ⋮ Y OPERAND	α [-] ⋮ ⋮ → p SET i α Y OPERAND ⋮ ⋮
t = 8 μsec. $0 \leq Y \leq 3777$	t = 0 μsec. Y = p + 1 $0 \leq Y \leq 1777$

SET i α	$40 + 20i + \alpha$	(t + 24) μsec.	SET

SET. Copy the contents of register Y into register α. $(0 \leq \alpha \leq 17)$. Take the next instruction from register p + 2. The contents of register Y are not changed.

SAM i n	$100 + 20i + \alpha$	*	SAM

SAMPLE. Sample the signal on input line n $(0 \leq n \leq 17)$ and leave its numerical value, seven bits plus sign, in the right-most 8 bits of the Accumulator, replicating the sign in the left-most 4 bits of the Accumulator. Lines 0 through 7 are used by eight potentiometers located at the Display Scope. Lines 10 through 17 are used by analog inputs at the Data Terminal module; on these lines +1 volt corresponds to +177, and -1 volt corresponds to -177. The contents of the Z register are destroyed.

* Timing: If i = 0, the instruction requires 24 μsec. for execution. If i = 1, the computer goes on to the next instruction after 8 μsec., even though the conversion process will continue in the Accumulator for 14 more μsec. If, therefore, the instruction is used with i = 1, care must be taken not to disturb the Accumulator during the 14 μsec. following the instruction.

DIS i α	$140 + 20i + \alpha$	32 μsec.	DIS

DISPLAY. Display on the scope a point whose vertical coordinate is specified by the right-most 9 bits of the Accumulator and whose horizontal coordinate is specified by the right-most 9 bits of register α ($0 \leq \alpha \leq 17$). The left-most bit of register α specifies one of two display channels (further selected by a switch on the Display Scope). The left-most horizontal coordinate is 000; the right-most, 777. The lowest vertical coordinate is -377; the highest, +377. The contents of bits 9 through 11 of the Accumulator and of register α do not affect the position of the point.

If i = 1, the contents of the right-most 10 bits of register α are first indexed by 1, using 10-bit binary addition without end carry.

XSK i α	$200 + 20i + \alpha$	16 μsec.	XSK

INDEX AND SKIP. If the address part (the contents of the right-most 10 bits) of register α ($0 \leq \alpha \leq 17$) equals 1777, skip the next register in the instruction sequence; otherwise, go on to the next instruction in sequence. If i = 1, the address part of register α is first indexed by 1, using 10-bit binary addition without end carry. The left-most two bits are not changed. Thus, 1777 is indexed to 0000; 3777, to 2000; 5777, to 4000; and 7777, to 6000.

Operate Class

OPR i n	500 + 20i + n	16 μsec. minimum	OPR
OPERATE CHANNEL n. Generate a negative signal on output level line n ($0 \leq n \leq 13$). If i = 1, pause until a restart signal appears on external level line n. Send other control signals to, and sense other signals from, equipment at the Data Terminal module; transfer data into or out of the memory or Accumulator as specified by these control signals.			

KBD i	515 + 20i	16 μsec. minimum	KBD
KEYBOARD. If a key has been struck and is locked down, clear the Accumulator, release the key, and read its 6-bit code number into the right half of the Accumulator. If no key has been struck and i = 1, pause until a key is struck and continue as above. If no key has been struck and i = 0, clear the Accumulator and go on to the next instruction.			

RSW	516	16 μsec.	RSW
RIGHT SWITCHES. Copy the contents of the Right Switches into the Accumulator.			

LSW	517	16 μsec.	LSW
LEFT SWITCHES. Copy the contents of the Left Switches into the Accumulator.			

Magnetic Tape Class

→p	MTP i u	700 + 20i + 10u
p + 1	QN|BN	1000QN + BN

i: Motion Control

i = 0 Tape stops after instruction execution.

i = 1 Tape is left in motion after instruction execution.

u: Unit Selection

u = 0 Tape Unit #0.

u = 1 Tape Unit #1.

QN: Quarter Number $0 \leq QN \leq 7$

QN	Memory Registers
0	0 - 377
1	400 - 777
2	1000 - 1377
3	1400 - 1777

QN	Memory Registers
4	2000 - 2377
5	2400 - 2777
6	3000 - 3377
7	3400 - 3777

BN: Block Number $000 \leq BN \leq 777$ (octal)

1 Tape = 512 (decimal) blocks.

1 Block = 256 (decimal) words.

1 Word = 12 (decimal) bits.

Data sum = sum without end-around carry of 256 words in block.

Check sum = complement of data sum.

Transfer check = data sum + check sum.

= -0 if block is transferred correctly.

≠ -0 if block is transferred incorrectly.

RDC i u	700 + 20i + 10u	RDC

READ AND CHECK. Copy block BN into memory quarter QN and check the transfer. If the block is transferred correctly, leave -0 in the Accumulator and go on to the next instruction; otherwise, repeat the instruction. The information on tape is not changed.

Magnetic Tape Class (continued)

RCG i u	701 + 20i + 10u	RCG
READ AND CHECK GROUP. Copy block BN into the memory quarter whose number corresponds to the right-most 3 bits of BN (block 773 into quarter 3, etc.) and copy the following consecutive QN blocks into the following consecutive memory quarters (block 000 follows block 777, quarter 0 follows quarter 7). Check each block transfer and repeat if necessary until all blocks have transferred correctly, then leave -0 in the Accumulator and go on to the next instruction. The information on tape is not changed.		

RDE i u	702 + 20i + 10u	RDE
READ TAPE. Copy block BN into memory quarter QN and leave the transfer check in the Accumulator. The information on tape is not changed.		

MTB i u	703 + 20i + 10u	MTB
MOVE TOWARD BLOCK. Subtract the next block number encountered from BN, leaving the difference in the Accumulator. When i = 1, leave the tape moving forward if the difference is positive and backward if the difference is negative or -0. QN is ignored.		

Magnetic Tape Class (continued)

WRC i u	704 + 20i + 10u	WRC
WRITE AND CHECK. Copy the contents of memory quarter QN into block BN and check the transfer. If the memory contents are transferred correctly, leave -0 in the Accumulator and go on to the next instruction; otherwise, repeat the instruction. The contents of memory are not changed.		

WCG i u	705 + 20i + 10u	WCG
WRITE AND CHECK GROUP. Copy the contents of the memory quarter whose number corresponds to the right-most 3 bits of BN into block BN (quarter 5 into block 665, etc.) and copy the contents of the following consecutive QN quarters into the following consecutive blocks (quarter 0 follows quarter 7, block 000 follows block 777). Check each transfer and repeat if necessary until all blocks have been written correctly, then leave -0 in the Accumulator and go on to the next instruction. The contents of memory are not changed.		

WRI i u	706 + 20i + 10u	WRI
WRITE TAPE. Copy the contents of memory quarter QN into block BN and leave the check sum in the Accumulator. The contents of memory are not changed.		

CHK i u	707 + 20i + 10u	CHK
CHECK TAPE. Find block BN, form its transfer check and leave it in the Accumulator. The information on tape and the contents of memory are not changed. QN is ignored.		

Appendix III: LINC Modifications

Mishell J. Stucki and Maurice L. Pepper

In August 1965, based on findings of the LINC Evaluation Program,[5] an interrupt feature, the Z Register, and five new instructions were made available on the LINC.

1. The Z Register

This is a 12-bit register, not shown on the console, which can be thought of as being to the right of the Accumulator. It is used as a utility register with the DSC and SAM instructions, and it holds the least significant half of the product following a MUL instruction. Each shift of the Accumulator during ROR and SCR also shifts the contents of the Z Register right with $A_0 \rightarrow Z_{11}$. (ROR 14 transfers C(ACC) to Z.) The Z Register is cleared by CLR. MUL, DSC, SAM, ROR, SCR, and CLR are the only instructions which alter the contents of the Z Register.

Following MUL, the least significant 11 bits of the product are in Z_1 through Z_{11}. Though the half product in the Accumulator is left with the proper sign, the half in the Z Register is always positive. Since the sign is left in the LINK bit, the following will recover the least significant half as an 11-bit signed number:

```
ZTA*
LZE
COM
```

The most significant 11 bits are lost if an integer multiplication is executed.

* See III-6.

2. Overflow

The following instructions set an overflow flag: ADD, ADA, ADM, and LAM. If there is overflow during execution of one of these instructions, the overflow flag is set on; if there is no overflow, it will be set off. Overflow results when two numbers of the same sign are added and the sum is of the opposite sign.

3. Interrupt Feature

The interrupt feature permits a program to be interrupted in the course of its operation. This feature has no effect until activated by a special interrupt enable instruction, ENI (MSC 10). Thereafter, if an interrupt request occurs, the normal running of the program will be interrupted and the next instruction will automatically be taken from location 21. Two kinds of interrupt, a program interrupt and a data interrupt are available. Which one of these will occur depends on the instruction in location 21.

Data Interrupt: Data interrupts are used to transfer data between memory and an external piece of equipment. This is done by putting an OPR instruction in register 21 and executing it in the GULP mode. The BCOM operation normally performed at 2.2 time of an OPR is inhibited so that the Accumulator will not be affected unless it is intentionally disturbed by the assertion of CLEL, SNEL, or TNEL. At the end of the OPR instruction, the machine will resume running the interrupted program.

Program Interrupt: A program interrupt allows the program to execute a special routine (service routine) whenever an interrupt occurs. This routine may be located anywhere in memory; it may not, however, begin in locations zero or 21. To arrange for a program interrupt, one puts the instruction "JMP X" in register 21 (X being the address of the service routine). This accomplishes three things:

1. It transfers program control to the service routine.
2. It stores the instruction "JMP n" in register zero (n is the address of the next instruction in the interrupted program).
3. It disables the interrupt feature so that the machine cannot be interrupted during the service routine.

Requesting an Interrupt: A -3V level on the pin called INTREQ (FC30) will request an interrupt. The level may occur asynchronously with the main machine but it must remain until the interrupt actually occurs. At that time a -3V level will appear on the pin called "BDOINTFF[1]" (FC15), indicating that the instruction in register 21 is being executed. The interrupt request must be removed within 16 μsec of the time this level appears.

Where Interrupts Can Occur: If the interrupt mode has been activated and an interrupt request appears, the program will be interrupted as soon as one of the following occurs:

1. The end of a non-JMP instruction. A program cannot be interrupted at the end of a JMP instruction.
2. The end of a non-ENI instruction. A program cannot be interrupted at the end of the instruction ENI.

   NOTE: This assumes that the interrupt feature is being activated by the ENI. However, if the interrupt feature is already active, i.e., the ENI is redundant, an interrupt can occur at the end of the instruction.
3. The occurence of a pause. An MTP or OPR instruction can be interrupted during the paused state. The instruction will be terminated abruptly and the interrupt executed. At the end of the interrupt the machine will return to the next instruction; it will not return to the unfinished instruction.

Writing Interruptable Programs: Programs utilizing the interrupt feature must be specially written in any section that can be interrupted.

1. Programs incorporating a program interrupt: The very first instruction in each subroutine must save the contents of register zero. This is necessary since a program interrupt occurring during the subroutine will destroy the contents of register zero.

   NOTE: An interrupt cannot occur immediately before the first instruction in a subroutine since that instruction is preceded by a JMP.

2. Programs incorporating either interrupt: Whenever an instruction is interrupted in the paused state, a flip-flop called PINFF (Pause Interrupt Flip-Flop) is set to a one. The state of this flip-flop can be checked with the instruction PIN (SKP 6). The PINFF should be checked after every instruction that pauses and the instruction should be repeated if an interrupt occurred.

Example:

```
WRC
3|400

PIN i
JMP p-3
```

Writing Service Routines:

1. If a service routine uses
   A. the Accumulator: the initial contents of the Accumulator must be saved and restored to it at the end of the routine.
   B. a JMP instruction: the return JMP in register zero must be saved.
2. The interrupt feature is automatically disabled upon entering a service routine. If the interrupt feature is to be operative upon returning to the interrupted program, the service routine must reactivate it just prior to the return. The instruction ENI must be the very last instruction before the return JMP. If it occurs any earlier, the service routine itself may be interrupted.

Disabling the Interrupt Mode:

1. Manually: Pushing the STOP switch on the console disables the interrupt mode. It also clears the PINFF.
2. Programs incorporating a program interrupt: The interrupt mode is automatically disabled every time an interrupt occurs. If it is not reactivated by an ENI at the end of the service routine, it will remain disabled.
3. Programs incorporating either interrupt: Putting the instruction NOP in register 21 disables the interrupt mode.

   NOTE: This will not disable the interrupt mode until the next interrupt request occurs. At that time the NOP is executed and the interrupt mode disabled. At the end of the NOP, the machine resumes running the interrupted program.
4. The paused state can not be interrupted while the PINFF is set to a one.

Additions: LINC Order Code Summary

Miscellaneous Class

ZTA	0005	8 μsec.	ZTA
Z TO ACCUMULATOR. Clear the Accumulator and then transfer the contents of the Z register to the Accumulator. The transfer is offset, so that $Z_i \rightarrow A_{i-1}$ and $0 \rightarrow A_{11}$. Z_0 is not transferred.			

ENI	0010	8 μsec.	ENI
ENABLE INTERRUPT. Enable the interrupt mode.			

SKIP CLASS

ZZZ i	455 + 20i	8 μsec.	ZZZ
Z ZERO ZERO. Condition: Bit zero of the Z Register contains 0.			

OVF i	454 + 20i	8 μsec.	OVF
OVERFLOW. Condition: The overflow flag is on. This instruction does not clear the overflow flag.			

PIN i	446 + 20i	8 μsec.	PIN
PAUSE INTERRUPT. Condition: The PINFF (Pause Interrupt Flip-Flop) is set to a one. Execution of this instruction clears the flip-flop.			

Appendix IV: LINC Variants

The chart outlines the main differences between the classic LINC, μ-LINC 1, μ-LINC 300, and LINC-8 that affect programming. It has been checked by the Digital Equipment Corporation and by SPeAR, Inc., manufacturers of the machines indicated. For the most part program modifications which may be necessary between machines are trivial. The chart and notes are intended simply as a guide; your attention is called to the references given.

Programs written for the: (are compatible with the:)	Classic LINC	μ-LINC 1	μ-LINC 300	LINC-8 (LINC mode)
Classic LINC	Programming the LINC	unless program uses: 4 unless μ-LINC 1 has: 3, 5	unless program uses: 1, 2, 4, 6, 7	unless program uses: 1, 2, 4, 7, 13
μ-LINC 1	unless program uses: 3, 4, 8	Information from: SPeAR, Inc. Bear Hill Rd. Waltham, Mass. 02154	unless program uses: 1, 2, 4, 6, 7, 8	unless program uses: 1, 2, 4, 7, 8, 13
μ-LINC 300	unless program uses: 1, 3, 4, 9, 11, 12	unless program uses: 1, 4, 9, 11, 12 unless μ-LINC 1 has: 2, 5	Micro-LINC-300 Order Code, SPeAR, Inc., Bear Hill Rd., Waltham, Mass. 02154	unless program uses: 1, 4, 9, 12, 13, 14
LINC-8 (LINC mode)	unless program uses: 1, 3, 4, 10, 11	unless program uses: 1, 4, 10, 11 unless μ-LINC 1 has: 2, 5	unless program uses: 1, 4, 6, 10	Small Computer Handbook, doc. C-800, and PROGOFOP, doc. DEC-L8-SFAO-D, D. E. C., Main St., Maynard, Mass. 01754.

1. Programmed timing loops. Instruction cycle time: classic LINC and μ-LINC 1, 8 μsecs.; μ-LINC 300, generally 1 μsec.; LINC-8, generally 1.5 μsecs.
2. 8-bit SAM values.
3. 9-bit SAM values.
4. Printer output. Classic LINC: unbuffered teletype printer usually connected through bit 0, Relay Register, and held off with a one in bit 0.
   μ-LINC 1: as classic LINC, or connected through channel 2 (OPR 2).
   μ-LINC 300: as classic LINC, or buffered teletype (OPR 42), or buffered Kleinschmidt (OPR 45) printer. Kleinschmidt interprets vertical bar ASCII code as line feed.
   LINC-8: buffered teletype printer (OPR 14).
   There are other variations. (All machines have Soroban code as standard Keyboard input. A few individual installations, however, use ASCII.)
5. 0 potentiometers.
6. Potentiometers 4-7.
7. OPR 0-14.*
8. MTT (Magnetic Tape Two).**
9. Operations LMB***, UMB***, MSC 2 (Set Flag), MSC 3 (Proceed from Tape Pause), MSC 4 (TA to A), MSC 7 (Disable Interrupt), MSC 12 (Clear Flag), SKP 16 (Tape Transfer), SKP 17 (Flag), MTT**, OPR 0-14*, OPR 40-77.
10. Operations LMB***, UMB***, OPR 13 (PDP), OPR 14 (TYP), EXC, OPR 0-12*.
11. "Buffered" feature of 2nd word (block number) of tape instructions. The classic LINC and the μ-LINC 1 need this word in the memory until the tape operation is finished.
12. Shift key to present upper case keyboard values directly to ACC.
13. Tape blocks which may be occupied by the LINC-8 "Program of Operation," PROGOFOP (normally blocks 0-11).
14. Memory bank 0 (reserved for PROGOFOP).

*OPR 0-14 are compatible between the classic LINC and the μ-LINC 1, but not between these two, the μ-LINC 300, and the LINC-8. The timing pulse generation is different between the first two and the μ-LINC 300. The LINC-8 OPR requires PDP-8 programming. There are thus slight logical differences in using OPR for buffered printer output on the μ-LINC 300 and the LINC-8.

**The second tape transport is optional on the μ-LINC 1 and μ-LINC 300. The MTT instruction is compatible between machines which have the transport.
The LINC-8 may have a second transport, addressable, compatibly, with MTT by modifying PROGOFOP. It will not then have a general purpose EXC instruction.

***Memory bank selection logic is handled differently on the μ-LINC 300 and the LINC-8, although the LMB/UMB coding is the same. Either machine may have 4K to 32K words. The classic LINC and the μ-LINC 1 have 2048 words, not paged.

References

1. Clark, Wesley A. and Charles E. Molnar, "A description of the LINC," in Computers in Biomedical Research II, R. W. Stacy and B. Waxman, Eds., Academic Press, New York, 1965.

2. Wilkes, Mary Allen, LINC Control Console, Center Development Office, M.I.T., Cambridge, July 1963.

3. Thomae, Irving, "An introduction to binary numbers and binary arithmetic," LINC Vol. 16, LINC Programming and Use I, Sec. 1, Washington University, St. Louis, April 1965.

4. LINC Vol. 12, Logic Drawings and Timing Diagrams, from: Computer Systems Laboratory, Washington University, St. Louis.

5. Convocation on the Mississippi, Proc. Final LINC Evaluation Program Meeting, Washington University, St. Louis, March 18-19, 1965.

See also:

Micro-LINC 300 Order Code, Spear, Inc., Bear Hill Rd., Waltham, Mass.

Small Computer Handbook, doc. C-800, Digital Equipment Corp., Main St., Maynard, Mass.

PROGOFOP, doc. DEC-L8-SFAO-D, Digital Equipment Corp., Main St., Maynard, Mass.