Memorandum 6M-5780

Sheet 1 of 15 Sheets

Division 6 — Lincoln Laboratory
Massachusetts Institute of Technology
Lexington 73, Massachusetts

SUBJECT: SOME EXAMPLES OF TX-2 PROGRAMMING

To: Distribution List

From: H. Philip Peterson H. Philip Peterson

Date: July 23, 1958

Approved: WAC

Abstract: Six short programs are presented here to illustrate many of the somewhat inscrutable features of TX-2 programming. These programs are called:

I: A Checkerboard Pattern Generator

II: The Inchworm

III: The Memory Mirror

IV: An Autocorrelation Program

V: The Flexo-Octal Converter

VI: A Binary Read-in Routine

Distribution List:

Group 63 Staff	Frachtman, H. E.
Arden, Dean (Barta)	Frick, F.
Arnow, J.	Grandy, C.
Attridge, W.	Hazel, F. P.
Bagley, P. R.	Heart, F.
Bailey, D.	Holmes, L.
Briscoe, H.	Israel, D.
Buzzard, R.	Mason, Wm.
Daggett, N.	Pughe, E. (Servo Lab.)
Dinneen, G. P.	Rising, H. K.
Dustin, D. E.	Thomas, L. M.
Forgie, Carma	Tritter, A. L.
Vance, R. R.	Zraket, C. A.

1. This document has been prepared for internal use only. It has not been reviewed by Office of Security Review, Department of Defense, and therefore, is not intended for public release. Further dissemination or reproduction in whole or in part of the material within this document shall not be made without the express written approval of Lincoln Laboratory (Publications Office).

2. The research reported in this document was supported jointly by the Department of the Army, the Department of the Navy, and the Department of the Air Force under Air Force Contract No AF 19 (122)-458.

INTRODUCTORY REMARKS

The six example programs presented in this paper illustrate many of the somewhat inscrutable features of TX-2 programming. A few assumptions, however have been made by the author about the reader. These assumptions are:

1) that the reader knows how to program;
2) that the reader is familiar with TX-2 nomenclature: (this familiarity may be attained by studying "The Lincoln TX-2 Computer," (6M-4968); and
3) that the reader has a copy of "The TX-2 Programmer's Guide" for reference (6M-5807).

NOTATION

The code part of each instruction is written as a group of 3 capital letters (ADD, JMP, etc.). Any superscript numbers preceding the code part refer to a configuration memory location, except for JPX, JNX, JMP and SKM instructions. Superscript numbers following a code refer to an index memory location except for SKM type instructions where this is the number of the bit in the addressed word. Lower case numbers following a code are main memory addresses.

A colon means "hold control until the next instruction." Brackets mean "defer the address" and imply that bit 2.9 of the address is a ONE. Lower case letters are hopefully self-explanatory.

To the left of many instructions will be an explanatory notation using four little lines which show the permutation involved by how they cross; the active quarters of central machine registers by arrowheads, and, when necessary, the fracture (or coupling or subwords) by little cups. This configuration will be specified by the contents of the indicated configuration memory word.

A number or word followed by "slash equals" defines the address of the instruction or constant to the right of it. A word followed by "equals slash" is the name of the register following.

An address section with a large L prefixing it, as in 763 of Program I, means the address of "a register containing what is indicated."

All numbers in programs are octal unless otherwise indicated. Numbers are punctuated with commas separating the meaningful portions of the whole 36 bit word. A single comma separates 9 bit (3 octal digit) quarters when the word is dealt with in quarters. Two consecutive commas will separate the word into 18 bit (6 octal digit) pieces.

I. A Checkerboard Pattern Generator

The Problem

When a core memory is being checked for operating margins, a "bad" pattern of ONES and ZEROS is desired. (see Engineering Note E-488). One of the worst conditions starts with a checkerboard pattern which looks like this in each memory plane:

```
0 1 1 0 0 1 1 . . .
1 0 0 1 1 0 0 . . .
1 0 0 1 1 0 0 . . .
0 1 1 0 0 1 1 . . .
0 1 1 0 0 1 1 . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
```

The complement of this pattern is also a checkerboard. The addresses increase from left to right and top to bottom beginning with address 000 at the upper left. In the case of a 256^2 memory plane, it takes 8 bits to address a row or a column (16 address bits in all).

If one computes the parity of the two least significant bits of the row address and the two least significant bits of the column address, one will find that if the parity of these four bits is odd, a ONE will be at that address; if even, a ZERO will be there.

The problem is to construct a program which generates this pattern in all 65,536 bits of each memory plane. The program must fit into the 16 toggle switch registers.

The Solution

Program I generates the checkerboard pattern by using four SKZ instructions to look at the two sets of least-significant address bits. These bits are 1.1, 1.2, 1.9 and 2.1. When any one of them is a ONE, the SKZ doesn't skip and an MKC is executed which complements bit 3.1 of the E register. After examining the four address bits, E register bit 3.1 will be ZERO for an even parity or ONE for an odd parity. The whole address is kept in index register 1 and the DPX at 751* puts the address in the right half of the E register, leaving the left half all ZEROS since configuration 0 is used. After computing the parity, the left half of E is put in index register 2 and the LDE at 763* puts the word at 766 in E if the parity is even, or the word at 767 if it is odd. The word in E is stored away at the address and the address is counted down. The address was reset by the RSX in 750 to 177,777. This number is kept in the A register (377,740) which is being simulated by a toggle switch register as of this writing.

*The three most significant octal digits of addresses (377 in toggle switch addresses) will be omitted for brevity's sake.

OCTAL EQUIVALENT	ADDRESS		SYMBOLIC	
02 11 01 377,740	377 750		^{2}RSX^{1}	$\lfloor$177,777
00 16 01 377,744	751	↓↓↓↓	^{0}DPX^{1}	e reg
10 17 41 377,744	752		SKZ$^{2.1}$	e reg
03 17 61 377,744	753		MKC$^{3.1}$	e reg
10 17 31 377,744	754		SKZ$^{1.3}$	e reg
03 17 61 377,744	755		MKC$^{3.1}$	e reg
10 17 22 377,744	756		SKZ$^{1.2}$	e reg
03 17 61 377,744	757		MKC$^{3.1}$	e reg
10 17 21 377,744	377 760		SKZ$^{1.1}$	e reg
03 17 61 377,744	761		MKC$^{3.1}$	e reg
02 11 02 377,744	762		^{2}RSX^{2}	e reg
00 20 02 377,766	763	↓↓↓↓	^{0}LDE^{2}	$\lfloor$word
00 30 01 000,000	764	↓↓↓↓	^{0}STE^{1}	memory
36 06 01 377,751	765		$^{-1}$JPX^{1}	next
00 05 00 377,750	766	(word)	JMP	restart
77 72 77 400 027	767		(- word)	

Program I   A Checkerboard Pattern Generator

```
OCTAL EQUIVALENT          ADDRESS             SYMBOLIC

02 11 01 377,740          377 750             ²RSX¹ ⌊177,777
00 16 01 377,744              751             ⁰DPX¹    e reg
10 17 41 377,744              752              SKZ²·¹  e reg
03 17 61 377,744              753              MKC³·¹  e reg
10 17 31 377,744              754              SKZ¹·³  e reg
03 17 61 377,744              755              MKC³·¹  e reg
10 17 22 377,744              756              SKZ¹·²  e reg
03 17 61 377,744              757              MKC³·¹  e reg
10 17 21 377,744          377 760              SKZ¹·¹  e reg
03 17 61 377,744              761              MKC³·¹  e reg
02 11 02 377,744              762             ²RSX²    e reg
00 20 02 377,766              763             ⁰LDE²  ⌊word
00 30 01 000,000              764             ⁰STE¹ memory
36 06 01 377,751              765            ⁻¹JPX¹ next
00 05 00 377,750              766     (word)    JMP  restart
77 72 77 400 027              767               (- word)
```

Program I   A Checkerboard Pattern Generator

Note that the word stored away for even parity is the JMP instruction at 766 which restarts the process after the address has been counted down through 000. If one wishes to write the pattern just once, put ALL ZEROS in 766 and ALL ONES in 767. TX-2 will halt with an illegal instruction alarm (ICSAL) if it tries to execute an instruction with 00 as the operation code. Putting +0 in 766 and -0 in 767 has the advantage that the checkerboard patterns in each digit plane will be identical.

Exercises To Prove To Yourself That You Really Understand

By changing a single toggle switch, the checkerboard pattern will be complemented. Which switch? (Hint: any one or three of four will do.)

What would you do to put the pattern just in the lower addressed half of memory? Upper half? Middle quarter?

Two memory planes of the 38 will not have a checkerboard pattern in them. They are the parity bit plane and the meta bit (4.10) plane. What program changes will put the pattern in either plane? (Note: only an SKM can modify a meta bit.)

II. The Inchworm

The Problem

A classical programming exercise is to design a routine which will move itself along through memory, carrying with it as it goes all necessary constants for repeating this "inchworm" process. The program for starting the inchworm on its way must fit into TX-2's 16 toggle switch storage registers, naturally.

The Solution

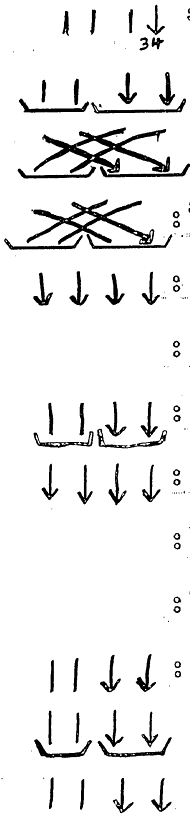
Program II solves the problem by storing in registers 001 - 007 the program shown. This program in 001 - 007 then forms the one in 010 - 016 which duplicates itself in 017 - 025 and so on. The program in togs works like the ones in main memory except for a few special setting up instructions.

The SPF instruction in 752 specifies that configuration 34 will permute quarter 3 into quarter 1 and extend its sign into quarter 2. The RSX in 755, 1, 10, etc. uses this oddball configuration to reset index register 71 to a -6 from quarter 3 of registers 762, 6, 15, etc. This trick allows the inchworm program to avoid carrying constants per se along with it. Each "old" inchworm setment can simply "fall" into the newly formed one without jumping around some constants.

Index register 2 contains the constant necessary for the program to move itself into the next location. When moving from togs to core memory, this constant is 400 023 and the RSX in 753 fixes it up. When moving on in core memory, this constant is 000,007 and the RSX in 764 sets it up.

The RSX in 754 resets index register 3 to snap back to togs after the last address desired is reached. For illustrative purposes the address constant 577,760 was chosen. Since the JPX in 763, 7, 16 etc. jumps when the index register is positive, for our purposes it must be negative until the end is reached. Consequently 400,000 is added to 177,760 and that number (577,760) is set up in left half of 750. Each time an inchworm "segment" is executed, the corresponding JPX will subtract 7 from the contents of index register 3. When control gets to the segment in 177,757 - 766, index register 3 will have become positive and control will be transferred to togs (after a segment is written into 177,767 to 75) starting the process over again.

The routine in 755 to 763 maps itself into 000 to 007, preserving the address parts of the instructions in 757, 760 and 763 as they go to 003, 004, and 007 by the action of the SKN in 757 which skips over the ADX when bit 3.2 of a word is a ONE. These three invariant instructions refer to fixed locations so they must not be changed by the ADX as the other four are. Bit 3.2 was arranged to be ONE in the invariant instructions and ZERO in the variable instructions.



OCTAL EQUIVALENT	ADDRESS	SYMBOLIC
577 760,,400 023	377 750	last address,,1st const
362	751	config 34 set up
34 21 00 377,751	752	^{34}SPF 377,751
01 11 02 377,750	753	^{1}RSX^{2} 377,750
02 11 03 377,750	754	^{2}RSX^{3} 377,750
74 11 71 377,762	755	:^{34}RSX^{71}377,762
40 20 71 377,763	756	: ^{0}LDE^{71}377,763
54 17 62 377,744	757	: SKN$^{3\circ 2}$e reg
41 15 02 377,744	377 760	: ^{1}ADX^{2} e reg
40 30 71 000,007	761	: STE^{71}000,007
41 0 771 377,756	762	:$^{+1}$JNX^{71}377,756
70 06 03 377,752	763	:$^{-7}$JPX^{3} restart
41 11 02 377,761	764	: ^{1}RSX^{2} 377,761
01 15 02 377,744	765	^{1}ADX^{2} e reg
01 30 00 000,005	766	^{1}STE 5
00 05 00 000,001	377 767	JMP 1

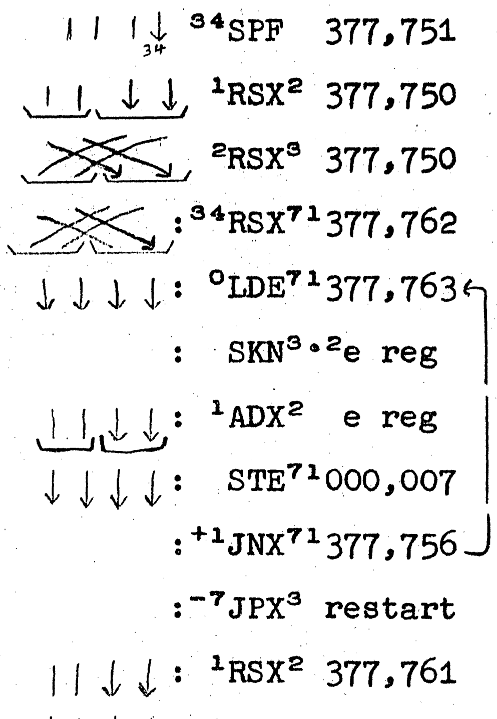
This program then forms this one

001	:^{34}RSX^{71} 6	010	:^{34}RSX^{71} 15
2	: ^{0}LDE^{71} 7	11	: ^{0}LDE^{71} 16
3	: SKN$^{3\circ 2}$377,744	12	: SKN$^{3\circ 2}$377,744
4	: ^{1}ADX^{2} 377,744	13	: ^{1}ADX^{2} 377,744
5	: ^{0}STE^{71} 16	14	: STE^{71} 25
6	:$^{+1}$JNX^{71} 2	15	:$^{+1}$JNX^{71} 11
7	:$^{-7}$JPX^{3} 377,752	16	:$^{-7}$JPX^{3} 377,752

and so on...

<u>Program II</u> <u>The Inchworm</u>

OCTAL EQUIVALENT	ADDRESS	SYMBOLIC
577 760,,400 023	377 750	last address,,1st const
362	751	config 34 set up
34 21 00 377,751	752	^{34}SPF 377,751
01 11 02 377,750	753	^{1}RSX^{2} 377,750
02 11 03 377,750	754	^{2}RSX^{3} 377,750
74 11 71 377,762	755	:^{34}RSX^{71}377,762
40 20 71 377,763	756	: ^{0}LDE^{71}377,763
54 17 62 377,744	757	: SKN$^{3\cdot 2}$e reg
41 15 02 377,744	377 760	: ^{1}ADX^{2} e reg
40 30 71 000,007	761	: STE^{71}000,007
41 0 771 377,756	762	:$^{+1}$JNX^{71}377,756
70 06 03 377,752	763	:$^{-7}$JPX^{3} restart
41 11 02 377,761	764	: ^{1}RSX^{2} 377,761
01 15 02 377,744	765	^{1}ADX^{2} e reg
01 30 00 000,005	766	^{1}STE 5
00 05 00 000,001	377 767	JMP 1



This program		then forms	this one
001	:^{34}RSX^{71} 6	010	:^{34}RSX^{71} 15
2	: ^{0}LDE^{71} 7	11	: ^{0}LDE^{71} 16
3	: SKN$^{3\cdot 2}$377,744	12	: SKN$^{3\cdot 2}$377,744
4	: ^{1}ADX^{2} 377,744	13	: ^{1}ADX^{2} 377,744
5	: ^{0}STE^{71} 16	14	: STE^{71} 25
6	:$^{+1}$JNX^{71} 2	15	:$^{+1}$JNX^{71} 11
7	:$^{-7}$JPX^{3} 377,752	16	:$^{-7}$JPX^{3} 377,752

and so on...

Program II The Inchworm

The "flaw in the ointment" is that register 005 will contain 400,023 + 000,007 = 400,032 after the first mapping. The STE instruction in 005 would have a deferred (indirect) reference to 32 and this is clearly bad. It must be changed to a direct reference to 016. This is accomplished by the ADX in 765 which adds the 000,007, which by then is in index register 2, to the 000,007 which remains in the right half of the E register (after the RSX in 764) resulting in an 000,016 in the E register. The STE in 766 puts it away into 005 and the JMP transfers control to 001 continuing the process in core memory.

Exercises To Prove To Yourself That You Really Understand

Write a program which uses another approach to the problem of what to put in the 16 toggle switch registers to make core memory look as it does above.

Is it possible to use a JPX in 762 and, if so, what would the program look like then?

III. Through the Looking Glass

The Problem

If all the registers in any block of memory registers were laid end to end, what program would put the mirror image of this mess back into the memory block? For example, if the block consisted of three 4-bit words, the transformation would look like this:

$$\begin{array}{cccc} F_1 & F_2 & F_3 & F_4 \\ S_1 & S_2 & S_3 & S_4 \\ T_1 & T_2 & T_3 & T_4 \end{array} \Longrightarrow \begin{array}{cccc} T_4 & T_3 & T_2 & T_1 \\ S_4 & S_3 & S_2 & S_1 \\ F_4 & F_3 & F_2 & F_1 \end{array}$$

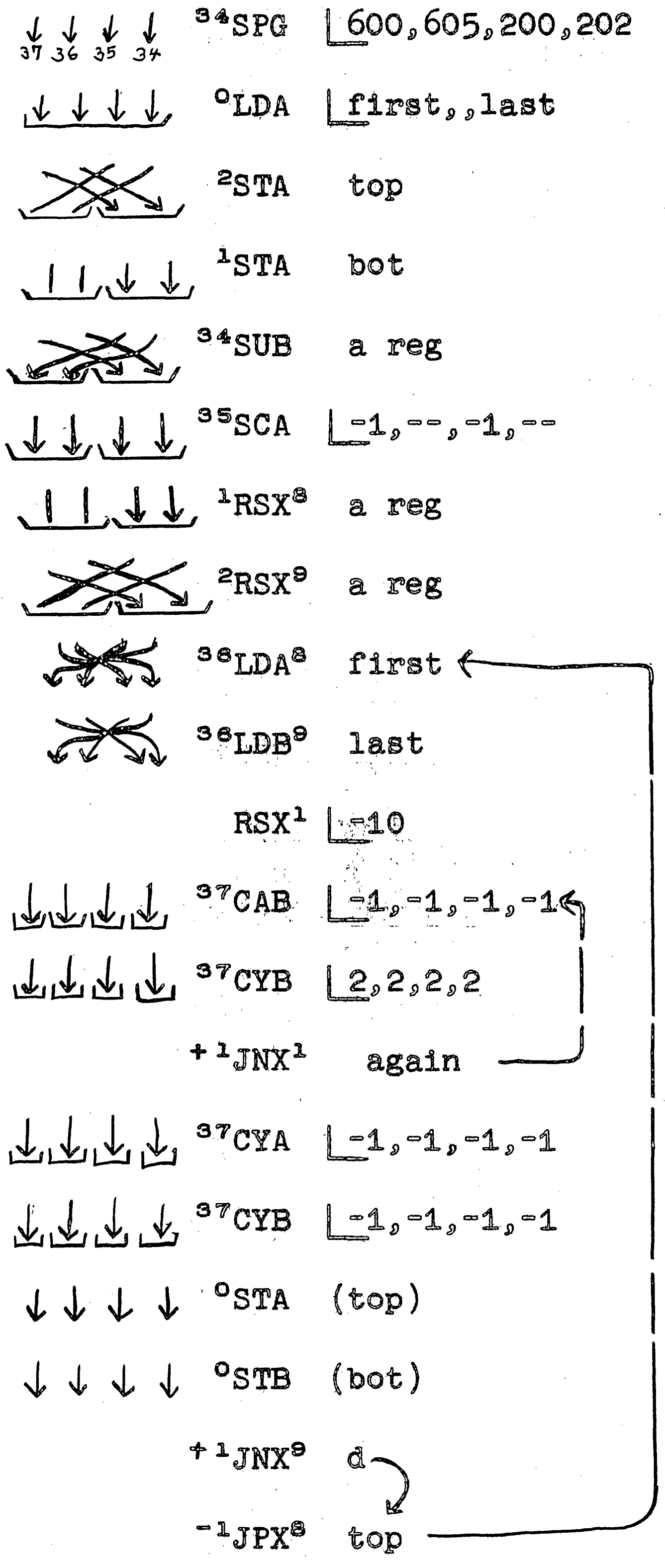
The Solution

Program III, which is written with floating addresses, performs this mirroring by the use of configurations and simultaneous cycling with only 20 instructions.

Four unusual configurations are needed and these are set up in configuration memory locations 37, 36, 35 and 34 by the SPG instruction.

From some register containing the first and last addresses of the memory block, the A register is set up and the "first" is put into the address section of the LDA instruction called "top," and the "last" is put into the LDB called "bot." The general idea is to index through the block, taking a pair of words at a time and exchanging and reversing them. One word comes from the top half of the block and the other comes from the bottom half. If the block has an odd number of words in it, the first pair will be the middle word used twice. If the block has an even number of words, the first pair will be the middle two words. The last pair dealt with is always the first and last words of the block.

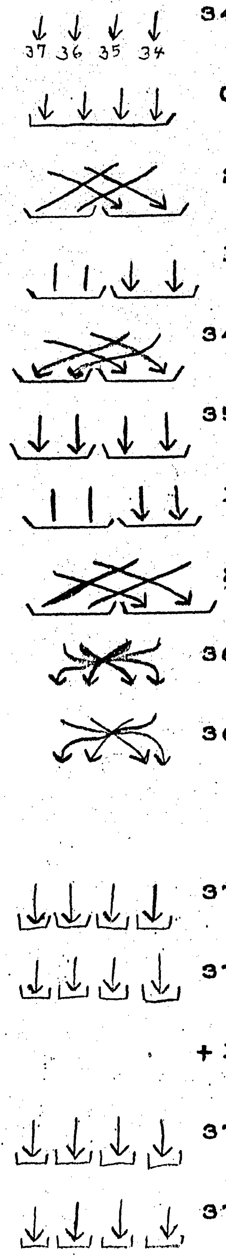
Index register 8 contains a positive number which counts back from the middle of the block to the first. Index register 9 contains a negative number which counts up from the middle to the last. If there are 2n+2 or 2n+1 words in the block, index 8 starts out with +n and index 9 starts out with -n. These numbers are obtained from the first and last addresses after only two instructions. The first instruction is a SUB which subtracts, simultaneously, the last from the first and the first from the last! The left half of the A register then contains -(2n+1) for even blocks and -(2n) for odd blocks. The right half of A contains the complement of the left half.



start=|	^{34}SPG	⌊600,605,200,202
	^{0}LDA	⌊first,,last
	^{2}STA	top
	^{1}STA	bot
	^{34}SUB	a reg
	^{35}SCA	⌊-1,--,-1,--
	^{1}RSX^{8}	a reg
	^{2}RSX^{9}	a reg
top=|	^{36}LDA^{8}	first
bot=|	^{36}LDB^{9}	last
	RSX^{1}	⌊-10
again=|	^{37}CAB	⌊-1,-1,-1,-1
	^{37}CYB	⌊2,2,2,2
	$+^{1}$JNX^{1}	again
	^{37}CYA	⌊-1,-1,-1,-1
	^{37}CYB	⌊-1,-1,-1,-1
	^{0}STA	(top)
	^{0}STB	(bot)
	$+^{1}$JNX^{9}	d
d=|	$^{-1}$JPX^{8}	top

Done, halt or something...

<u>Program III</u> <u>Memory Mirror</u>



```
start=|   34SPG   ⌊600,605,200,202
           0LDA   ⌊first,,last
           2STA    top
           1STA    bot
          34SUB    a reg
          35SCA   ⌊-1,--,-1,--
          1RSX8    a reg
          2RSX9    a reg
top=|    36LDA8    first
bot=|    36LDB9    last
           RSX1   ⌊-10
again=|   37CAB   ⌊-1,-1,-1,-1
          37CYB   ⌊2,2,2,2
         +1JNX1     again
          37CYA   ⌊-1,-1,-1,-1
          37CYB   ⌊-1,-1,-1,-1
           0STA   (top)
           0STB   (bot)
         +1JNX9    d
d=|      -1JPX8    top
```

Done, halt or something...

<u>Program III</u>   <u>Memory Mirror</u>

The next instruction, SCA, shifts each half one place to the right, leaving -n in the left half and +n in the right half of A. Index registers 8 and 9 are then set up from the appropriate half of A.

The basic iterative loop starts now and is executed n times. The inner loop is executed 9 times for each of the n times through the outer loop. This number 9 is the number of bits in a quarter of a TX-2 word. If the reader wishes to work through an example with, let's say, 4 bit quarters, then he should go through the inner loop four times. The index register (1) is preset to -8 however, since the JNX jumps on zero.

The STA and STB instructions (at d-3) have deferred addresses which they get from "top" and "bot" respectively. This is actually inefficient timewise if n is greater than 2. Two more instructions when setting up could have put direct references to "first" and "last" in these STA and STB instructions. This would have cost 4 memory time cycles. However, each deferred address costs one memory cycle and so 2n-4 extra memory cycles are being executed in the basic loop. This illustrates how one can trade space for time or vice versa.

The two decimal numbers 8 and 9 were used to indicate general index registers. Of course, 1 is general too.

Exercises To Prove To Yourself That You Really Understand

One need execute the inner loop only 8 times if a slightly different correction is made afterwards. What are the new correcting cycle instructions?

Configuration 35 is not really needed. What other one used by Program III would serve just as well?

IV. 50 Million Multiplications Can't Be Wrong

The Problem

In the analysis of electroencephalographic data, the autocorrelation function of the data is often desired (see B. G. Farley). A specific useful example is the following: about 50 thousand samples are stored away in memory. Each sample is a sign and 8 bits (9 bits in all).

We wish to find

$$\sum_{j=1}^{j=50,000} Sj \cdot Sj+1, \qquad \text{for } i=0,1,\dots,1000$$

where Sj is the j^{th} sample. These 1000 numbers are proportional to the autocorrelation function.

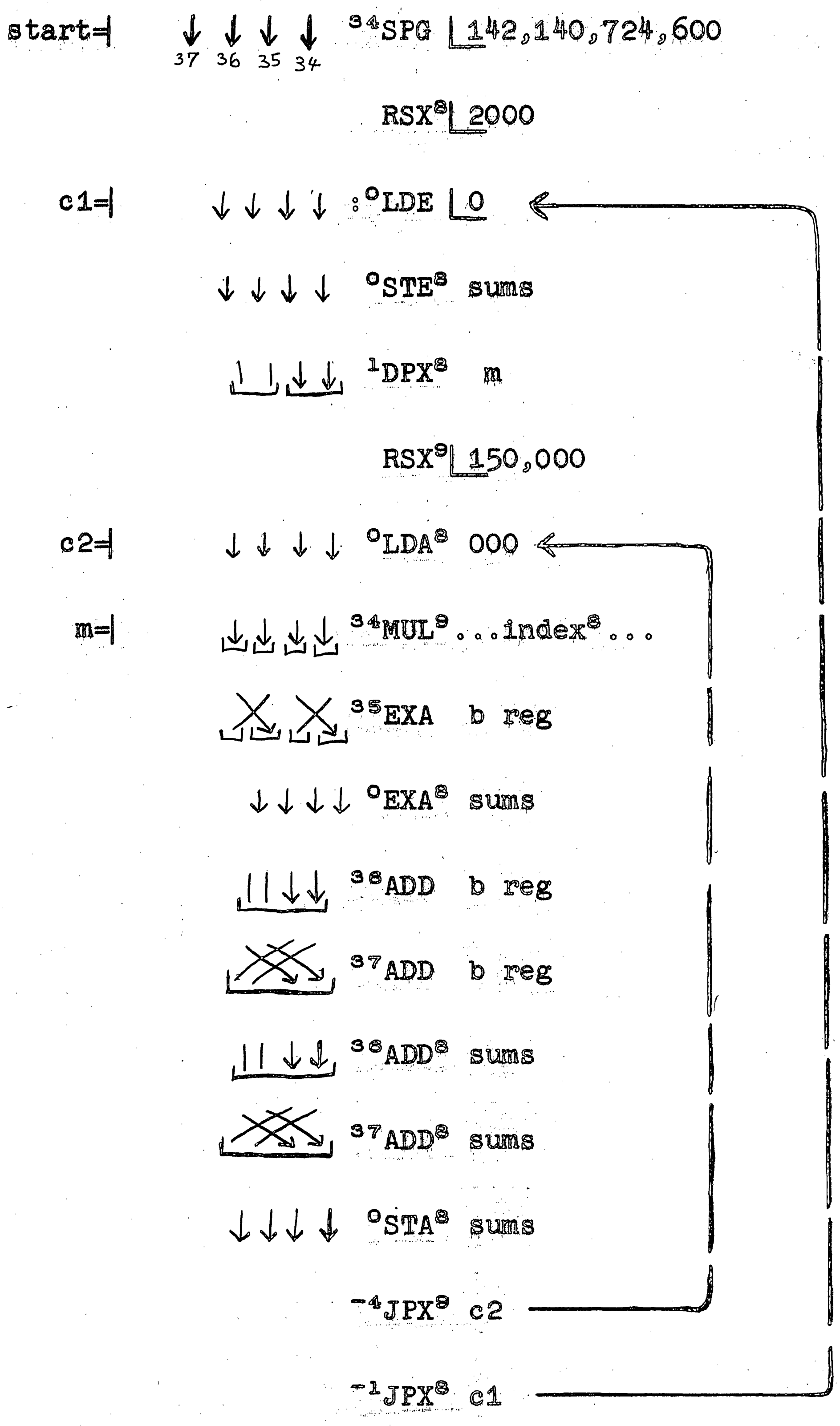
The Solution

Program IV computes this function in a most efficient way timewise. The key to the speed is to do four multiplications simultaneously. The data, however, must be in memory in a particular format, namely

0 =	S_1,	S_2,	S_3,	S_4
1	S_2,	S_3,	S_4,	S_5
2	S_3,	S_4,	S_5,	S_6
3	S_4,	S_5,	S_6,	S_7
4	S_5,	S_6,	S_7,	S_8
5	S_6,	S_7,	S_8,	S_9

Note that there are four of the 9 bit samples (Sj) in each TX-2 word and that registers 0, 4, etc. and 1, 5, etc. will contain eight different successive samples.

The program starts out by setting up the four special configurations needed and reseting index register 8 to 2000 octal (about 1000 decimal). Index register 8 corresponds to the subscript i in the summation above.



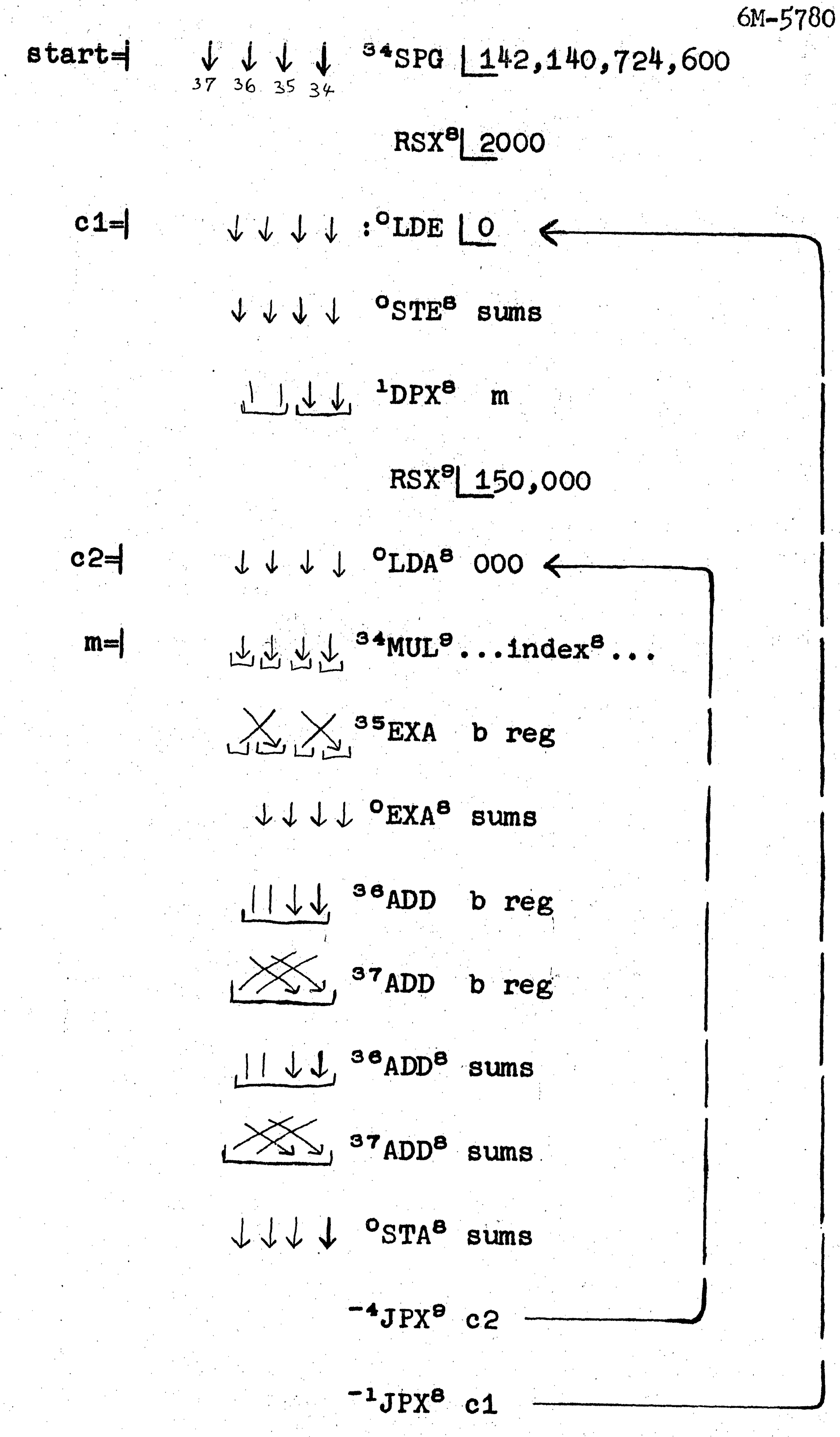
```
start=|   ↓  ↓  ↓  ↓   34SPG ⌊142,140,724,600
          37 36 35 34

                        RSX8⌊2000

c1=|         ↓ ↓ ↓ ↓  :0LDE ⌊0   <-----------+
                                             |
             ↓ ↓ ↓ ↓   0STE8 sums            |
                                             |
                       1DPX8  m              |
                                             |
                        RSX9⌊150,000         |
                                             |
c2=|         ↓ ↓ ↓ ↓   0LDA8 000  <-----+    |
                                        |    |
m=|                   34MUL9...index8...|    |
                                        |    |
                      35EXA  b reg      |    |
                                        |    |
             ↓ ↓ ↓ ↓   0EXA8 sums       |    |
                                        |    |
                      36ADD  b reg      |    |
                                        |    |
                      37ADD  b reg      |    |
                                        |    |
                      36ADD8 sums       |    |
                                        |    |
                      37ADD8 sums       |    |
                                        |    |
             ↓ ↓ ↓ ↓   0STA8 sums       |    |
                                        |    |
                      -4JPX9 c2  -------+    |
                                             |
                      -1JPX8 c1  ------------+
```

Done, display results (the 2000 sums)...

<u>Program IV</u>   <u>An Autocorrelation Program</u>



```
start=|   ↓ ↓ ↓ ↓   ³⁴SPG ⌊142,140,724,600
                   RSX⁸⌊2000
c1=|      ↓ ↓ ↓ ↓  :⁰LDE ⌊0
          ↓ ↓ ↓ ↓   ⁰STE⁸ sums
                    ¹DPX⁸  m
                   RSX⁹⌊150,000
c2=|      ↓ ↓ ↓ ↓   ⁰LDA⁸ 000
m=|                ³⁴MUL⁹...index⁸...
                   ³⁵EXA  b reg
          ↓ ↓ ↓ ↓   ⁰EXA⁸ sums
                   ³⁶ADD  b reg
                   ³⁷ADD  b reg
                   ³⁶ADD⁸ sums
                   ³⁷ADD⁸ sums
          ↓ ↓ ↓ ↓   ⁰STA⁸ sums
                  ⁻⁴JPX⁹ c2
                  ⁻¹JPX⁸ c1
```

Done, display results (the 2000 sums)...

Program IV   An Autocorrelation Program

The outer iterative loop then clears the i th <u>current sum register</u> and puts i into the address section of the MUL instruction at m. Index register 9 is set to 150,000 octal (about 50,000 decimal). Index 9 corresponds to the subscript j in the summation. This outer loop is executed about a thousand times.

The inner loop computes one complete summation (fixed i) taking four samples at a time. After the multiplication, the A and B registers look like this:

$$A = P_1, \ P_2, \ P_3, \ P_4$$

$$B = L_1, \ L_2, \ L_3, \ L_4$$

where P is the most significant 9 bits of the product and L is the least significant.

To eliminate round-off errors, the sums of each whole 18 bit product are accumulated. To put the 9 bit pieces of the product together, the A register is exchanged with the B register in such a manner that the result looks like this:

$$A = P_1 \ L_1 \ ,, \ P_3 \ L_3$$

$$B = P_2 \ L_2 \ ,, \ P_4 \ L_4$$

These four 18 bit numbers are then added to the current sum which is a 36 bit number. Notice how the sign extension feature allows a signed 18 bit number to be added to a signed 36 bit number.

Index register 9 is counted down by 4 (!!) since only every fourth register of four samples need be multiplied. This means the inner loop is executed only about 13,000 times instead of 50,000 times.

The whole program with its 50,000,000 multiplications will take 8 minutes if the overlapped memory feature is used (i.e. if instructions and data are in different memories).

<u>Exercises To Prove To Yourself That You Really Understand</u>

The data should extend to register 152,000. Why?

Write a program, using appropriate configurations (no shifting) and the TSD instruction, which will read the samples into memory in the desired format. This program would operate in the Epsco Datrac (an analog-to-digital converter) sequence. Each TSD will put a signed 9 bit number into quarter 1 of the E register. Ignore In-Out Select instructions. Nine instructions will do nicely.

Write a new inner loop to Program IV which handles data with only one sample per word. Five instructions including the JPX will do it. This inner loop will have to be executed the full 50 million times. How long will it take?

V. The Flexo-Octal Converter

The Problem

In the beginning of a binary computer's programming life, it is difficult to communicate with the machine. A series of programs must be written to "bootstrap" one's way into easy communication. This bootstrap series might go like this:

First) A three (or so) word program in toggle switch storage which would allow words to be written into memory one at a time. Call this P1.

Second) A short routine to convert programs to binary which have been typed on a flexo in a rigid, simple, fixed-address format. Call this P2. Associated with P2 is a program to punch out storage as a binary tape and a program to read in this binary tape. P1 loads P2 into memory. P2 converts the punch-out and read-in programs. The punch-out program punches out P2, the read-in routine and itself. From now on, the read-in routine can read in P2 and the punch-out routine, eliminating the need for P1.

Third) A longer routine which converts programs typed in a symbolic code, relative-address format. Call this P3. P2 converts P3 and punches it out, eliminating the need for P2.

Fourth) A routine to convert programs typed in a symbolic code, floating address format (P4). P4 is written in P3 format and converted by P3. At this point P1, P2 or P3 aren't needed any more and communication is fairly easy. In TX-0, P4 was called TODAL. A fifth stage might be an algebraic format converter like FORTRAN.

Programs V, VI and VII are proposed examples of the second stage. The octal converter recognizes the eight flexo symbols 0,1,2,3,4,5,6 and 7 takes their order into account. Some control characters are needed, such as carriage return to signify the end of a word and slash to allow address specifications. The space, tab, and comma are used to give some format control. The nullify is recognized so that tape "goofs" can be fixed up. The last four are ignored by the converter. A stop code signifies the end-of-tape condition.

The Solution

The program to do the octal conversion is Program V.

To decide what action to take on each character as it is read in, an Action Table is set up as is shown beside the program. An entry is made at the address, starting at 100, whose last 2 digits correspond to the flexo code of the appropriate character. The right 18 bits of each entry tell where to transfer control when that character is read in, and the left nine bits tell what the binary equivalent is

ACTION TABLE	Description	ROUTINE			
105| = 0,0,000 201	start at →	200| =		:IOS^{52}read	unsplayed
107| = 3,0,000 210	if slash →	201	||↓↓	^{1}STA	213
110| = 0,0,000 204		202	╳↘	^{2}RSX^{2}	105
113| = 4,0,000 210		203	↓↓↓↓	^{0}DPX^{0}	a reg ←
117| = 2,0,000 210	if ignored →	204		IOS^{52}	dismiss ←
123| = 5,0,000 210		205		:TSD	e reg
125| = 1,0,000 210		206	|||↓	^{3}RSX^{1}	e reg
127| = 7,0,000 210	←	207		^{0}JMP^{1}	(100)
133| = 6,0,000 210	if number →	210	↓↓↓↓	^{0}CYA	107
145| = 0,0,000 204		211	///↘	^{6}ADD^{1}	100
151| = 0,0,000 213		212		JMP	204
161| = 0,0,000 216	if car/ret →	213	↓↓↓↓	^{0}STA^{2}	memory
176| = 0,0,000 210		214	///↘	^{6}AUX^{2}	125
177| = 0,0,000 204		215		JMP	203
	if stop code →	216		IOS^{52}shut off	

address| = value,0,where to go

Program V The Flexo-Octal Converter

```
ACTION TABLE                Description      ROUTINE

105| = 0,0,000 201          start at →       200| = :IOS⁵²read unsplayed

107| = 3,0,000 210          if slash →       201     ||↓↓ ¹STA    213

110| = 0,0,000 204                           202          ²RSX²   105

113| = 4,0,000 210                           203     ↓↓↓↓ ⁰DPX⁰   a reg  ←┐
                                                                          │
117| = 2,0,000 210          if ignored →     204          IOS⁵² dismiss ←┐│
                                                                         ││
123| = 5,0,000 210                           205          :TSD    e reg  ││
                                                                         ││
125| = 1,0,000 210                           206     |||↓ ³RSX¹   e reg  ││
                                                                         ││
127| = 7,0,000 210                      ←    207          ⁰JMP¹  (100)   ││
                                             ─────────────────────────── ││
133| = 6,0,000 210          if number →      210     ↓↓↓↓ ⁰CYA    107    ││
                                                                         ││
145| = 0,0,000 204                           211          ⁶ADD¹   100    ││
                                                                         ││
151| = 0,0,000 213                           212          JMP     204  ──┘│
                                             ───────────────────────      │
161| = 0,0,000 216          if car/ret →     213     ↓↓↓↓ ⁰STA²   memory  │
                                                                          │
176| = 0,0,000 210                           214          ⁶AUX²   125     │
                                                                          │
177| = 0,0,000 204                           215          JMP     203  ───┘
                                             ───────────────────────
                            if stop code →   216          IOS⁵²shut off

address| = value,0,where to go
```

Program V The Flexo-Octal Converter

when the character is a number. Quarter 3 of each entry is not used.

The IOS in 200 sets the mode of the PETR to read one contiguous 6-bit flexo code (unsplayed) into the right 6 bits of the E register, clearing the other 3 bits in that quarter.

Starting at 203 with a DPX which clears the A register, the character is read in and placed in index register 1. The JMP then defers control to a location specified by the appropriate Action Table entry. Note that all instructions with deferred addresses are indexable.

If the character is a number, then control goes to 210 where the A register is cycled left 3 places and the binary equivalent of the number is added into A, returning control to 204.

If the character is a slash, control "bounces off" register 105 to register 201 where the number in A is stored in 213 and index register 2 reset to a zero. The slash then causes the number that has been built up in A to be the new address of the word which follows.

If the character is a carriage return, 213 has control and stores the word in A away in the proper memory location. The AUX in 214 adds a 1 to index register 2 so that the next time a carriage return appears, the word in A will be stored in the memory register following the last one.

The nullify, space, and tab simply return control to 204 to read-in the next character. When a stop code comes along, the IOS in 216 shuts off the photo reader and dismisses the sequence.

The sequence must be dismissed after each character is read and the IOS in 204 does this. The TSD in 205 empties a buffer that has been filled by the PETR. When the buffer is filled, the sequence is activated and the character read-in is dealt with.

Exercises To Prove To Yourself That You Really Understand

What are the implications of throwing out the IOS in 204 and not holding on the TSD which follows? In other words, let the TSD dismiss the sequence after transferring the data. Work out the new program and format rule(s).

The instructions in 210-11 are on rather shaky ground because TX-2 is an allegedly multi-sequence machine. Some lower priority sequence may have been using the A register and will be very upset at finding it disturbed. What changes will fix this up? Don't forget 203!!

Is there anything fishy about the RSX in 206?

VI. A Binary Read-In Routine

The Problem

In one of its modes, the photoreader reads the six bits of a line of tape into every sixth bit of some specified word and cycles the word left one place. This is the "splayed" mode of the photoreader sequence. After reading in six lines, a full 36 bit word is assembled. This mode would usually be used to read in binary tapes.

The main problem associated with a binary read-in routine is what format to use. In general, data words are read into blocks of consecutive memory registers and three provisions are made; (1) to read in more than one block, (2) to check the sum of each block thereby detecting almost any error, (3) to specify what should happen to control after all blocks are read in.

The Solution

Program VI uses the following format for each block of binary words:

```
        -n ,, last address

           Word 0

           Word 1
             .
             .
             .
           Word n

        more? ,, - sum
```

The first word in each block consists of two 18 bit numbers (see instructions at 3 and 4) which designate the addresses of the actual data words which follow.

The right half of the last word is the complement of the sum of all the other half words in the block. In other words, if all the words in a block are added up in 18 bit pieces (instructions at 24 and 25) the sum must be zero (instructions at 12 and 13) or there has been an error. If there is an error, the tape is backed up (instruction 17) and read in again. (TX-2, as you may have guessed by now, can read paper tape in either direction and can identify the front of the tape.)

The sign bit (4.9) of the left half of the last word in a block tells whether there are more (if 4.9 is a ONE) blocks to be read in or not (if 4.9 is a ZERO, see instruction 14). If there are more

	00| =		IOS^{52} read forward, splayed, dismiss	
NEW BLOCK	1		RSX^{3}	26
	2	"save"	$^{2}JMP^{4}$	21
SET UP RANGE	3	$\cancel{\searrow\searrow}$:	$^{2}RSX^{2}$	30
	4	|| ↓↓	^{1}STE	07
	5	"save"	$^{2}JMP^{4}$	21
PUT DATA AWAY	6	↓↓↓↓ :	^{0}LDE	30
	7	↓↓↓↓	$^{0}STE^{2}$	last
	10		$^{+1}JNX^{2}$	5
	11	"save"	$^{2}JMP^{4}$	21
CHECK THE SUM	12		$^{+0}JPX^{3}$	17
	13		$^{+0}JNX^{3}$	17
	14		$SKZ^{4,9}$	30
	15		JMP	1
DONE	16		IOS^{52} shut off, dismiss	
GOOF	17		IOS^{52} back up tape, dismiss	
	20		JMP	000

SUB-ROUTINE TO READ 6 LINES

21| =	|| ↓↓	$^{1}RSX^{1}$	27
22		TSD	30
23		$^{-1}JPX^{1}$	22
24	$\cancel{\searrow\searrow}$	$^{2}AUX^{3}$	30
25	|| ↓↓	$^{1}AUX^{3}$	30
26	"index"	$^{1}JMP^{4}$	000
27		0,0,0,5	
30		word read in	

Program VI A Binary Read-in Routine

```
00| =              IOS⁵²read forward, splayed, dismiss

NEW        1                RSX³   26
BLOCK      2  "save"       ²JMP⁴   21
-----------------------------------------            SUB-ROUTINE
                                                     TO READ 6 LINES
SET        3             :²RSX²    30            21| =  ||↓↓ ¹RSX¹   27
UP         4  ||↓↓        ¹STE     07            22            TSD   30
RANGE      5  "save"      ²JMP⁴    21            23          ⁻¹JPX¹  22
-----------------------------------------
PUT        6  ↓↓↓↓       :⁰LDE     30            24            ²AUX³ 30
DATA       7  ↓↓↓↓        ⁰STE²  last            25       ||↓↓ ¹AUX³ 30
AWAY      10             ⁺¹JNX²     5            26  "index" ¹JMP⁴   000
                                                 -----------------------
          11  "save"      ²JMP⁴    21            27         0,0,0,5
-----------------------------------------
CHECK     12             ⁺⁰JPX³    17            30     word read in
THE       13             ⁺⁰JNX³    17
SUM       14              SKZ⁴·⁹   30
          15              JMP       1
-----------------------------------------
DONE      16              IOS⁵²shut off, dismiss
-----------------------------------------
GOOF      17              IOS⁵²back up tape, dismiss
          20              JMP     000
-----------------------------------------
```

Program VI   A Binary Read-in Routine

blocks, control goes to register 1 and reads in the next block, providing of course that there were no check sum errors. If there are no more blocks, instruction 16 shuts off the PETR and dismisses the sequence.

Exercises To Prove To Yourself That You Really Understand

Note that there is no provision made in the tape format of Program VI for turning on any other sequence after the last block has been read in. There is really no necessity for a control change since the Start-Over sequence can start up the program just read in at the poke of a button.

However, pay homage to the (W. A.) Clarkian philosophy of minimal button poking and make the necessary additions of Program VI and its format which will start the program in sequence #S at a register called START if bit 4.8 of the last word in the last block is a ONE. If 4.8 is a ZERO, make Program VI do what it does now. This addition can be accomplished with eleven more words (maybe fewer).

Why are the CF bits of instruction 12 all ZEROS?

Do they need to be ZEROS in instruction 13? Why?

VII. A Punch Out Routine

To prove to yourself that you really, really understand, write a program to punch out storage in the block format required by the read in routine (VI). Control it from a toggle switch register in the following manner:

> Let the left half of the toggle switch register be the first address, and the right half, the last address of the block to be punched out.
>
> Let the meta bit (4.10) designate whether this is the last block or not.
>
> Let bit 4.9 be a ONE when the toggles are being changed, and a ZERO when the program can look at the register.

The author has written this program with 33 instructions. The best solution submitted by a reader, will be published in a supplement to this memo.

CONCLUDING REMARKS

The six programs in this memo illustrate many of the characteristics of TX-2. There are other features which haven't been illustrated. For example, conditionally saving the P and/or Q register in E after a JMP; using multiple step deferred (indirect) addresses; using the Boolean instructions or the skip if E is different from word instruction; using the operate class commands and many sequences operating simultaneously.

There will be supplements to this memo from time to time which illustrate features such as those mentioned in the preceding paragraph. Any suggestions, improvements, discoveries, or remarks in general will be appreciated by the author and probably also by his associates.

HPP/mk

Insertions:

Pages 3a
6a
9a
12a
15a
18a

NOTES