

Preliminary Larry Bower  
(for corrections)

**LID**

**Nano Processor**

**User's Guide**

**Drawing Number**  
**A-5955-0331-1**



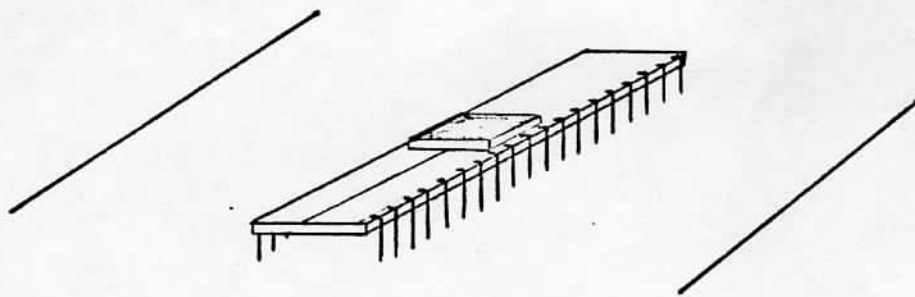
HEWLETT  PACKARD



**L I D**

**NANOPROCESSOR**

---



---

**Users  
Guide**

HEWLETT  PACKARD

7

## PREFACE

The Loveland Instrument Division Nano Processor is a control oriented device designed for instrument applications. The Nano Processor is not arithmetic oriented. The motivation for such a design was three fold. First, it was felt that ASM designs were too limiting; second, "off-the-shelf" microprocessors had too many "real time" limitations; and finally, there was a need for a common building block among LID designs. Thus the two major objectives for the Nano Processor were the design of a general purpose LSI device optimized for instrument control and to provide a software method of implementing complex control algorithms.

Some of the key features of the Nano Processor are an internal data base of sixteen 8-bit registers, seven direct control I/O lines, fixed time high speed instructions, high speed vectored interrupt, and bit oriented control instructions. The Nano Processor can operate at speeds up to 500 nanoseconds per any of its 42 instructions, while dissipating less than one watt from a ceramic 40 pin package. The factory cost of this device is less than \$20 or less than \$27 with an ALU.

50  
60  
70

Foot to inch line

3  
2  
1

Running feet



## TABLE OF CONTENTS

	Page
I. INTRODUCTION .....	1
II. HARDWARE STRUCTURE .....	1
III. PROCESSOR TIMING .....	5
IV. PROGRAM ADDRESSING .....	8
V. NANO PROCESSOR INSTRUCTION SET .....	9
VI. INTERFACING THE NANO PROCESSOR .....	14

### APPENDICES

- A. SPEC. SHEET
- B. ROM-RAM SIMULATOR
- C. NANO PROCESSOR DESIGN EXAMPLES
- D. SOFTWARE
- E. GENERAL INFORMATION

## I. INTRODUCTION.

The -hp- Nano Processor (NP) is a single chip, N-channel MOS, 8 bit parallel, control oriented central processing unit designed by the Loveland Instrument Division for internal control and interfacing of instruments.

The NP coupled with a program ROM forms the minimum nano processor control computer. The NP can directly address up to 2048 8-bit bytes of program memory, and with simple block switching techniques, up to 512 K of 8-bit bytes.

All instructions and data are transferred in and out of the NP with the bidirectional 8-bit parallel data bus (D0 through D7).

The NP allows data transfers with up to 15 input and 15 output ports addressed by a 4-bit device select code and an I/O read/write control line.

The normal program may be interrupted by use of the interrupt request control line. This interrupt is a fully vectored interrupt with 256 possible vectors.

The NP can control external circuits and check their status through the use of the 7 direct control lines (DC0 through DC7).

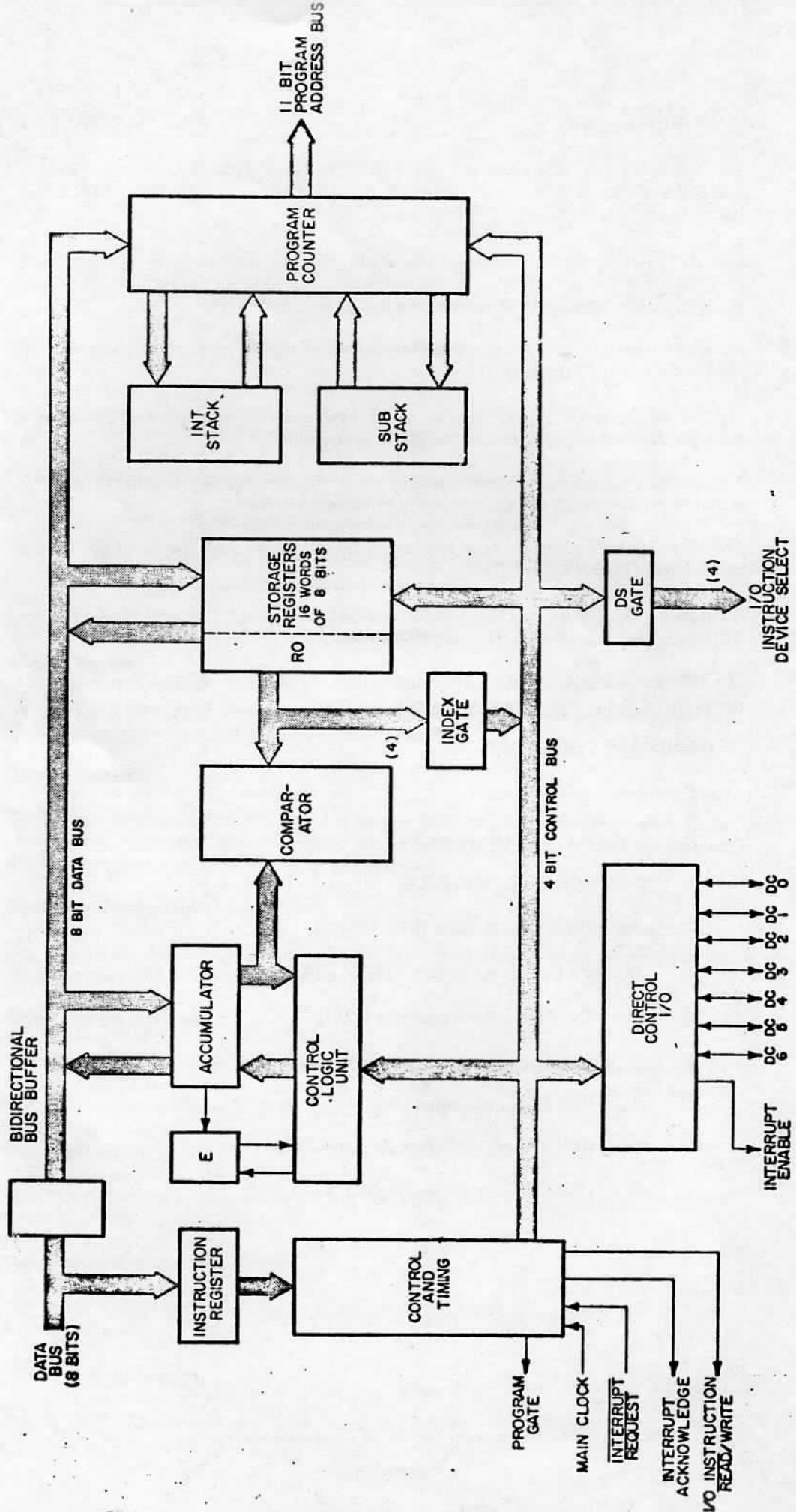
All inputs and outputs are TTL compatible. Each output will sink one standard power TTL load. Each input has an internal pull-up device.

The NP instruction set numbers 42 including data transfers, bit manipulation, magnitude comparisons, jump, and jump to subroutine.

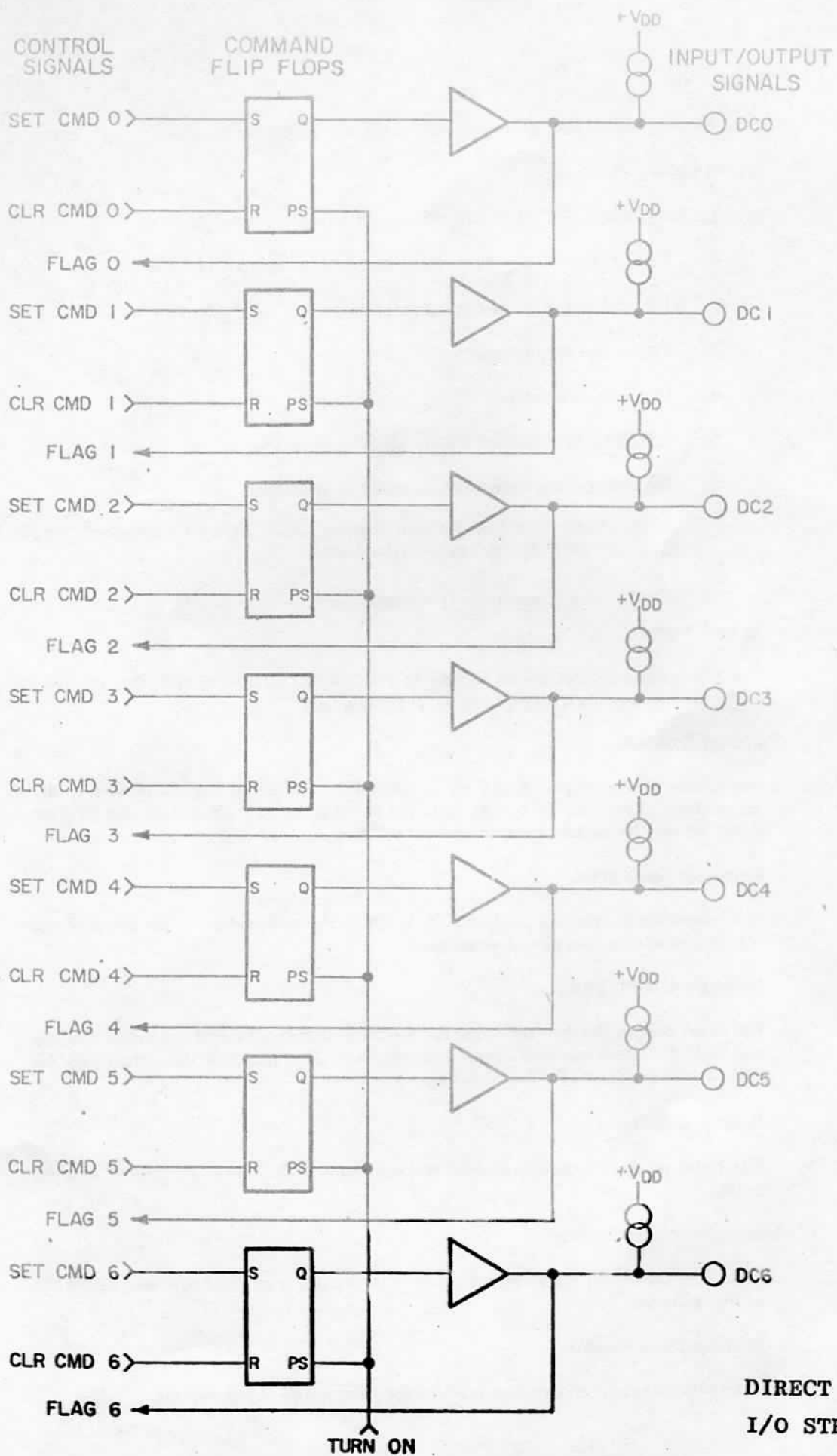
## II. HARDWARE STRUCTURE.

The NP contains:

- A. One 8-Bit Accumulator (ACC)
- B. One Control Logic Unit (CLU)
- C. One 1-Bit Extend Register (E)
- D. Sixteen 8-Bit Storage Registers (R0 - R17)
- E. One 8-Bit Magnitude Comparator (CMP)
- F. Seven Bidirection Direct Control I/O Lines (DC0 - 6)
- G. One 11-Bit Program Counter (PC)
- H. One 11-Bit Subroutine Return Register (SRR)
- I. One 11-Bit Interrupt Return Register (IRR)



NANO PROCESSOR BLOCK DIAGRAM



### **Accumulator.**

The 8-bit accumulator may be loaded from or output to the 8-bit data bus.

### **Control Logic Unit.**

The CLU is the heart of the NP. It provides the following functions:

1. Test, set or clear any bit of the accumulator or the extend register.
2. Set or clear any of the command flip-flops.
3. Test any of the flag inputs.
4. Clear the accumulator.
5. Increment or decrement the accumulator in binary.
6. Increment or decrement the accumulator in decimal.

*(Note: Two Binary Coded Decimal (BCD) digits are assumed and the output is two BCD digits and overflow/carry.)*

7. Complement accumulator (1's complement)

### **Extend Register.**

The 1-bit extend register is used to indicate overflow (carry) from or underflow (borrow) to the accumulator, or it may be used as an internal flag.

### **Storage Registers.**

The sixteen 8-bit storage registers are for general data use. They may be recalled to the accumulator. They may be loaded from the accumulator or directly from the program ROM. R0 may be used for comparisons and indexing.

### **Magnitude Comparator.**

The magnitude comparator compares the 8 bits of the accumulator to the 8 bits of the R0 for greater than, less than or equal to.

### **Direct Control I/O Lines.**

The direct control I/O lines are 7 lines (DC0 – DC6) that may be used for output with set and clear functions on their controlling flip-flops. The status of the output may be directly tested as inputs for feedback flags.

### **Program Counter.**

The 11-bit program counter provides direct addressing of the control program up to 2048 bytes.

### **Subroutine Stack Register.**

The 11-bit subroutine stack register provides for a single level of subroutining within the control program.

### **Interrupt Stack Register.**

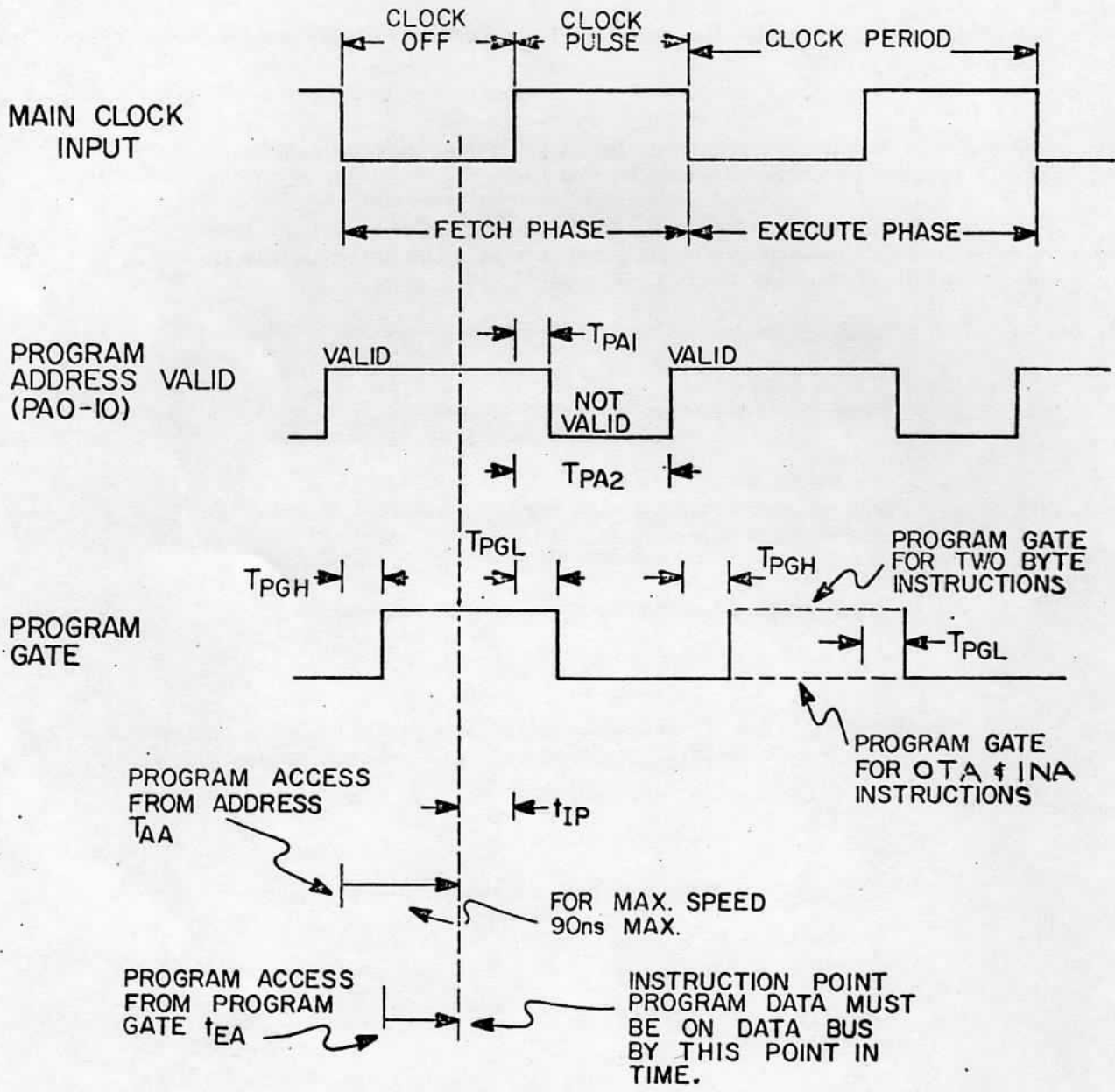
The 11-bit interrupt stack register provides for a single level of interruption.

### III. PROCESSOR TIMING.

The NP is designed with a quasi static structure. The clock may be stopped *in the low state* with no loss of data.

The maximum clock rate is 4 MHz for the fast (A series) chips. *All* instructions are executed in two clock periods or 500 ns with this clock rate.

To obtain a 500 ns cycle time the program ROM must have  $< 85$  ns access from address to output and  $< 65$  ns access from output enable to output. (A list of possible ROM's to be used with the NP is liated in Appendix A.)



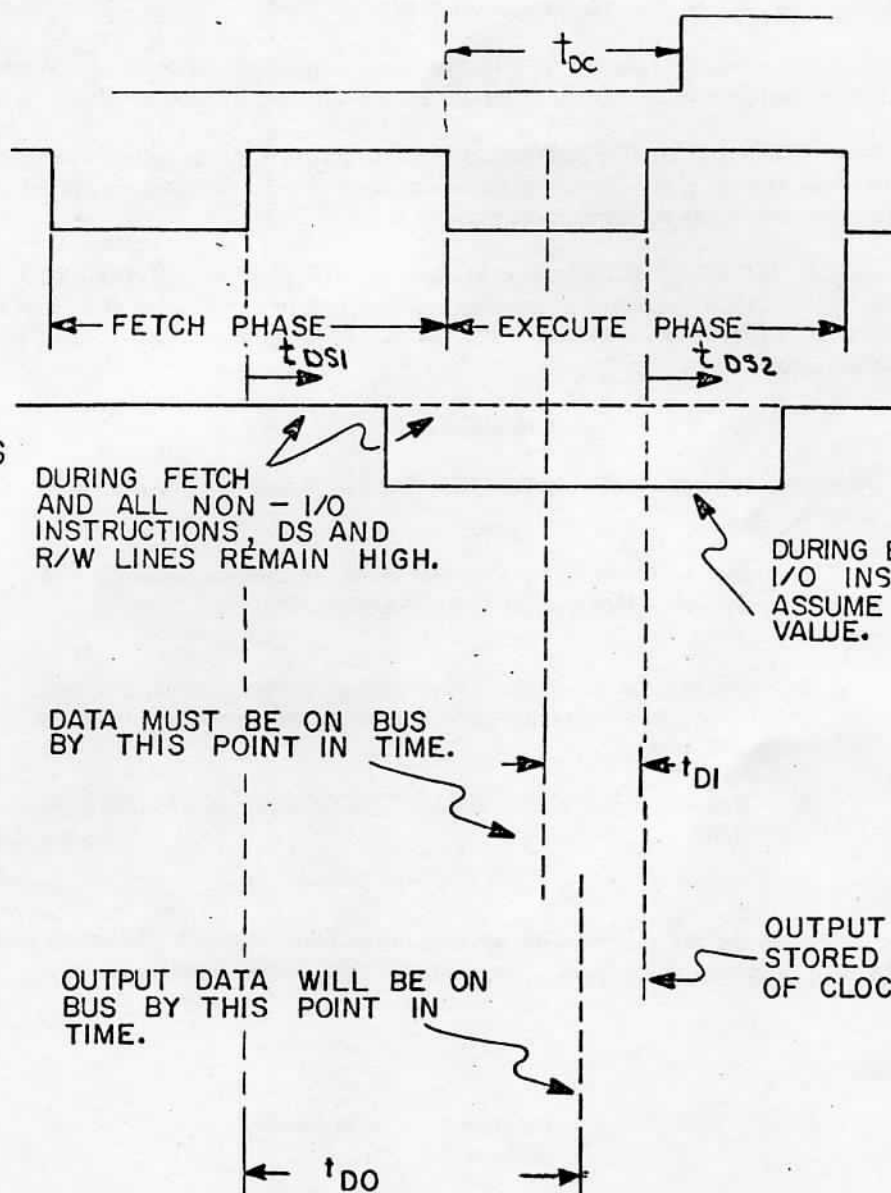
DC / I/O LINES

MAIN CLOCK INPUT

DEVICE SELECT AND R/W OUTPUTS

DATA INPUT (R/W LOW)

DATA OUTPUT (R/W HIGH)



DURING FETCH AND ALL NON - I/O INSTRUCTIONS, DS AND R/W LINES REMAIN HIGH.

DURING EXECUTE OF I/O INSTRUCTION LINES ASSUME PROGRAMMED VALUE.

DATA MUST BE ON BUS BY THIS POINT IN TIME.

OUTPUT DATA SHOULD BE STORED ON LEADING EDGE OF CLOCK.

OUTPUT DATA WILL BE ON BUS BY THIS POINT IN TIME.

OUTPUT DATA NO LONGER VALID



#### IV. PROGRAM ADDRESSING.

For ease of discussion the program address (11 bits) will be looked at as a 3-bit page number (PA 10 - PA 8) and an 8-bit page offset (PA 7 - PA 0).

In all instruction except jump and skip instructions, the program address is incremented. It is incremented once in one byte instructions and twice in two byte instructions.

In a JUMP (JMP) or JUMP TO SUBROUTINE (JSB) instruction, the page number from the first byte and the page offset from the second byte of the instruction are loaded into the program counter during the execute phase.

In the JUMP INDIRECT INDEXED (JAI) and the JUMP INDIRECT INDEXED TO SUBROUTINE (JAS) instructions, the page number is formed the same as an indexed register address (but only the bottom 3 bits are used) and the page offset is taken from the accumulator.

#### CAUTIONS:

*These two instructions allow great addressing power but they also have great dangers.*

1. *Due to the indexing structure, a JAI instruction executed with R03 set will be executed as a JAS instruction.*
2. *Due to the subroutine return address storage system, the byte after a JAS instruction will not be executed upon return from the subroutine.*
3. *Remember that this is an OR FUNCTION not an ARITHMETIC ADD.*

All branching in the NP is done with the skip instructions. The skip instruction causes two bytes of program to be skipped if the condition being tested is true.

#### Example:

	Program Address	Instruction
After the skip instruction	N	SBS 3 Skip if accumulator bit 3 is set.
This instruction is executed →	N + 1	JMP EXIT (Jump instructions require two bytes.)
if Bit 3 is zero	N + 2	
This instruction is executed →	N + 3	CBN 3 Clear accumulator bit 3
if Bit 3 is Set		

## V. THE NANO PROCESSOR INSTRUCTION SET.

The NP instruction set is divided into groups:

1. Accumulator group
2. Register transfer group
3. Input/output group
4. Comparator group
5. Program control group

### Instruction Listing Format.

SBS	N	$\boxed{00010N}$	
Instruction Mnemonic	Operand(s)	Operation Code	Operand Code

### Register Addressing.

The sixteen internal 8-bit registers may be directly addressed with LOAD (LDA), STORE (STA) and STORE ROM DATA (STR) instructions or indexed address may be used with LOAD INDEXED (LDI) and STORE INDEXED (STI).

The effective indexed address is the "or" function of the bottom ( $I_0 - I_3$ ) 4 bits of the instruction with the bottom 4 bits of  $R_0(R_{00} - R_{03})$ .

Example:

$I_0 - I_3$	1001
$R_{00} - R_{03}$	0101
Effective Register	1101

Address

*Note: This is an "or" function instead of an add, therefore, no carry takes place.*

**\*\*\*Note: Since  $R_0$  is used as the index, caution should be used so that  $R_0$  is not the effective destination of a Store instruction.\*\*\***

## V - A. ACCUMULATOR GROUP.

SBS N	0 0 0 1 0 N
	Skip on accumulator bit #N Set (1)
SBZ N	0 0 1 1 0 N
	Skip on accumulator bit #N zero (0)
SBN N	0 0 1 0 0 N
	Set accumulator bit #N
CBN N	1 0 1 0 0 N
	Clear accumulator bit #N
INB	0 0 0 0 0 0 0 0
	Increment accumulator as an 8-bit binary number. The extend register is set if overflow occurs.
IND	0 0 0 0 0 0 1 0
	Increment accumulator as two BCD code decimal numbers ( ) ( ). Carry between digits is automatically handled. The extend register is set if overflow occurs.
DEB	0 0 0 0 0 0 0 1
	Decrement accumulator as an 8-bit binary number. The extend register is set if underflow occurs.
DED	0 0 0 0 0 0 1 1
	Decrement accumulator as two BCD coded decimal digits. Borrow between digits is automatically handled. The extend register is set if underflow occurs.
CLA	0 0 0 0 0 1 0 0
	Clear accumulator. Does <i>not</i> affect the extend register.
CMA	0 0 0 0 0 1 0 1
	Complement accumulator The accumulator is treated as an 8-bit binary number and one's complement is performed.
LSA	0 0 0 0 0 1 1 1
	Left shift accumulator 1-bit shift with zero (0) fill. Does <i>not</i> affect extend register.
RSA	0 0 0 0 0 1 1 0
	Right shift accumulator 1-bit shift with zero (0) fill. Does <i>not</i> affect extend register.
SES	0 0 0 1 1 1 1 1
	Skip on extend register set (1).
SEZ	0 0 1 1 1 1 1 1
	Skip on extend register Zero (0).
LDR	1 1 0 0 1 1 1 1
	ROM Data. Load accumulator with ROM data. (ROM data is the second byte of this instruction)

15

15

## V - B. REGISTER TRANSFER GROUP.

LDA R	0 1 1 0 R
Load accumulator with data from register #R.	
STA R	0 1 1 1 R
Store accumulator at register #R.	
LDI Z	1 1 1 0 Z
Load accumulator with data from register addressed by (Z)v(R0). (See description of indexing.)	
STI Z	1 1 1 1 Z
Store accumulator at register addressed by (z) v (R0).	
STR R,	1 1 0 1 R
ROM Data.	ROM Data
Store ROM data at register #R.	
ROM data is the second byte of this instruction.	

## V - C. EXTEND REGISTER GROUP.

STE	1 0 1 1 0 1 0 0
Set extend register.	
CLE	1 0 1 1 0 1 0 1
Clear extend register.	

## V - D. INTERRUPT GROUP.

DSI	1 0 1 0 1 1 1 1
Disable the interrupt.	
ENI	0 0 1 0 1 1 1 1
Enable the interrupt.	

## V - E. COMPARATOR GROUP.

All comparisons are made based on R0 and the accumulator containing 8-bit unsigned binary numbers.

SLT	0 0 0 0 1 0 0 1
Skip on accumulator less than R0.	
SEQ	0 0 0 0 1 0 1 0
Skip on accumulator equal to R0.	
SAZ	0 0 0 0 1 0 1 1
Skip on accumulator equal to zero (0).	
SLE	0 0 0 0 1 1 0 0
Skip on accumulator less than or equal to R0.	
SGE	0 0 0 0 1 1 0 1
Skip on accumulator greater than or equal to R0.	
SNE	0 0 0 0 1 1 1 0
Skip on accumulator <i>not</i> equal to R0.	
SAN	0 0 0 0 1 1 1 1
Skip on accumulator not equal to zero (0).	
SGT	0 0 0 0 1 0 0 0
Skip on accumulator greater than R0.	

## V - F. INPUT/OUTPUT GROUP.

INA DS	0 1 0 0 DS
Input data from device #DS to accumulator.	
OTA DS	0 1 0 1 DS
Output accumulator data to device #DS.	
OTR DS, ROM DATA	1 1 0 0 DS
ROM Data.	ROM DATA
Output ROM data to device #DS	
ROM data is the second byte of this instruction.	
STC K	0 0 1 0 1 K
Set direct control.	
Bit #K	
CLC K	1 0 1 0 1 K
Clear direct control.	
Bit #K	
SFS J	0 0 0 1 1 J
Skip on direct control.	
Flag #J Set (1).	
SFZ J	0 0 1 1 1 J
Skip on direct control flag #J zero (0).	

RTI	Return from interrupt. An unconditional jump to the location stored in the interrupt stack register is performed. The interrupt control bit is <i>not</i> affected.	1 0 0 1 0 0 0 0
RTE	Return from interrupt and enable interrupt. Same as RTI instruction except that the interrupt control bit is <i>set</i> allowing future interrupt.	1 0 1 1 0 0 0 1
NOP	NO Operation.	0 1 0 1 1 1 1 1
JAI	Jump indirect (through accumulator) indexed. The page number is the indexed value (Z) v (R0). The page offset is the accumulator. An uncondition jump to the address formed from the page number and page offset.	1 0 0 1 0 Z
JAS	Jump indirect (through accumulator) indexed to subroutine. Same as JAI with the addition that the location of the JAS instruction <i>Plus 2</i> is stored in the subroutine stack register.	1 0 0 1 1 Z

**CAUTIONS:**

*These two instructions allow great addressing power but they also have great dangers.*

1. *Due to the indexing structure, a JAI instruction executed with R03 set will be executed as a JAS instruction.*
2. *Due to the subroutine return address storage system, the byte after a JAS instruction will not be executed upon return from the subroutine.*

60  
61

3  
2  
1

line

ect

M

5

## V - G. PROGRAM CONTROL GROUP.

JMP	ADDRESS	1 0 0 0 0 Page Offset	Page Number
	The address is broken into two section page number and page offset. The first byte contains operation code and page number. The second byte contains the page offset. An unconditional jump to the address is performed.		
JSB	ADDRESS	1 0 0 0 1 Page Offset	Page Number
	(See jump for address format) An unconditional jump to the address is performed and the address of the next ROM location after the page offset is stored in the subroutine stack register. Note: Since the subroutine stack register is a single level deep, subroutines <i>cannot</i> be nested.		
RTS		1 0 1 1 1 0 0 0	
	Return from subroutine. An unconditional jump to the location stored in the subroutine stack register is performed. The location of the RTS instruction <i>Plus 2</i> is stored in the subroutine stack register, thus co-routine linkages may be performed.		
RSE		1 0 1 1 1 0 0 1	
	Return from subroutine and enable interrupt. Same as RTS instruction except that the interrupt control bit is set allowing future interrupt.		

## VI . INTERFACING THE NANOPROCESSOR.

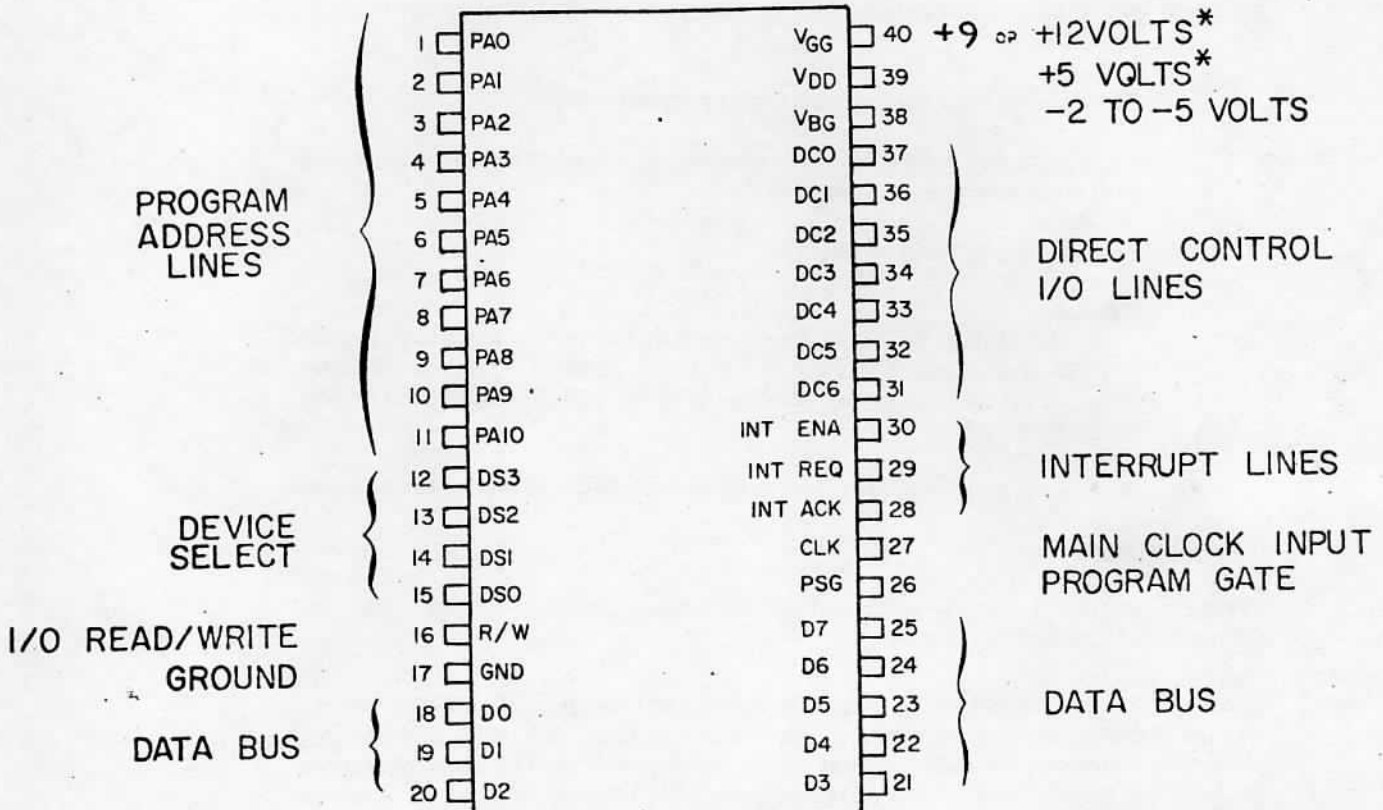
The interface of the NP is divided into five sections:

1. Program Access
2. I/O Port
3. Direct Control Lines
4. Interrupt System
5. Power Supplies and Clock

### Program Access.

The NP accesses its program through the use of the 11 program address lines (PA<sub>0</sub> - 10) and the program and gate line.

When the program gate is high the program source should supply the program data referenced by the program address onto the data bus.



\* ±5% , < IW TOTAL

### PIN OUT



## **I/O Ports.**

The NP can address up to 16 input and 15 output data ports through the use of its device select and I/O Read/Write lines.

The external devices may be numbered 0 through 17 in octal. OTA 17 is used as the NOP instruction.

## **Direct Control Lines.**

The seven bidirectional direct control lines may be used in one of four modes for each line.

1. As a dc static output line with set/clear program control.
2. As an input flag (internal flip-flop must be set – this is the turn-on condition) with direct testing by the program.
3. As a bidirectional control line.

### **Example:**

The NP puts DC Line 2 low to signal an external device to start and the external device holds the line low until finished. Thus, the NP (after setting dc lines again) can determine the end of the external devices cycle.

4. As an internal program flag with set/clear and direct testing by the program.

## **Interrupt System.**

The NP's interrupt system is controlled by three lines: Interrupt Request, Interrupt Acknowledge, and Interrupt Enable.

During the execute phase of every instruction (except an interrupt disable – clear control #7) the status of the interrupt request line is checked. If that line is low, an interrupt phase will follow regardless of the state of the interrupt enable. The interrupt phase is indicated by the interrupt acknowledge line going high. Daisy chaining of the interrupt acknowledge line can be used for interrupt priority.

During the interrupt phase the interrupt enable is automatically turned off; the vector address is input and the return address is stored in the interrupt stack register.

The interrupt request line input is *always* active. The interrupt enable output *may* be used externally to gate this input if interrupt enable/disable capability is required. See Interrupt System Timing.

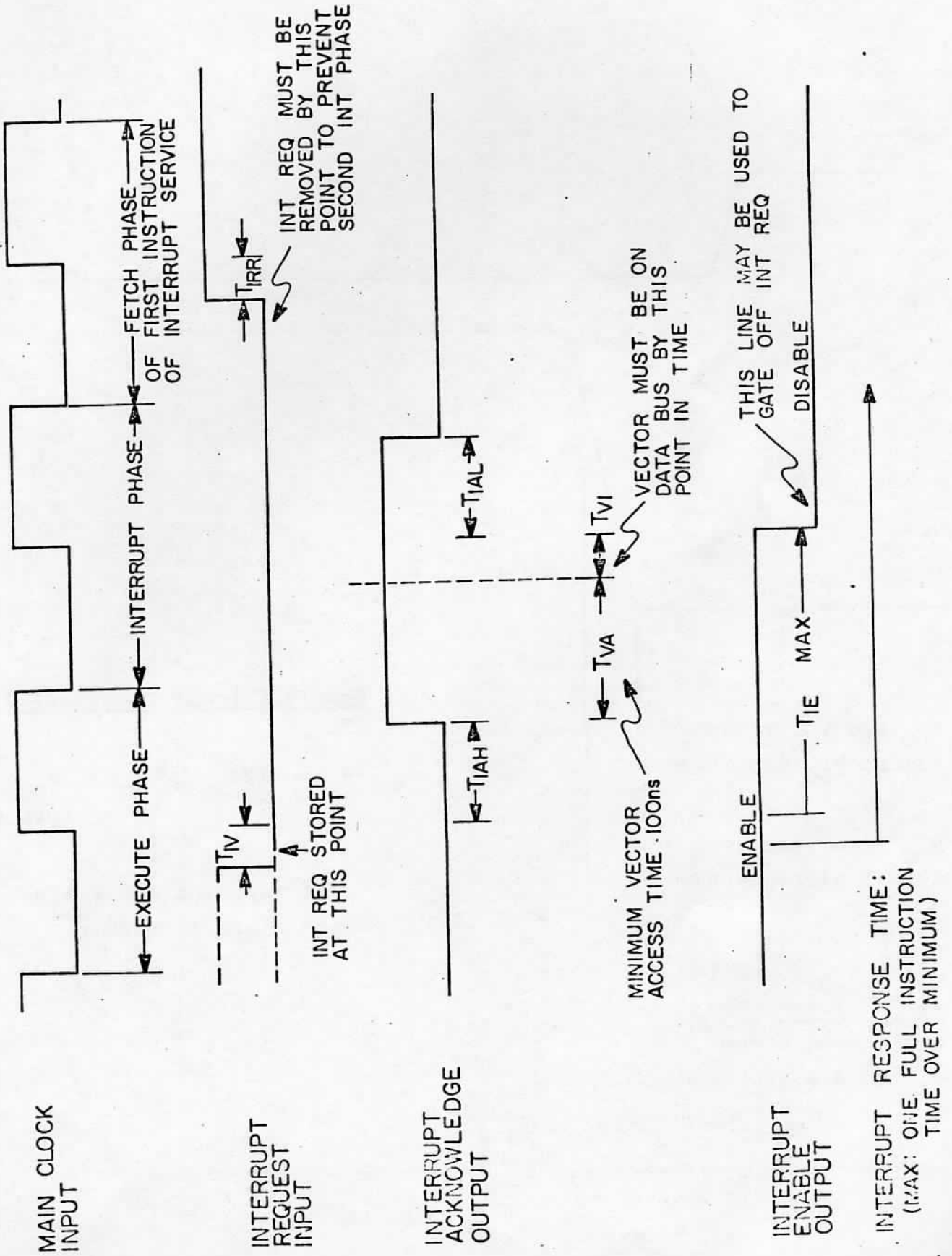
## **Power Supply And Clock.**

Three power supplies are required by the NP: + 12 or + 9 volts and + 5 volts for the main logic and - 2 to - 5 volts for backgate bias.

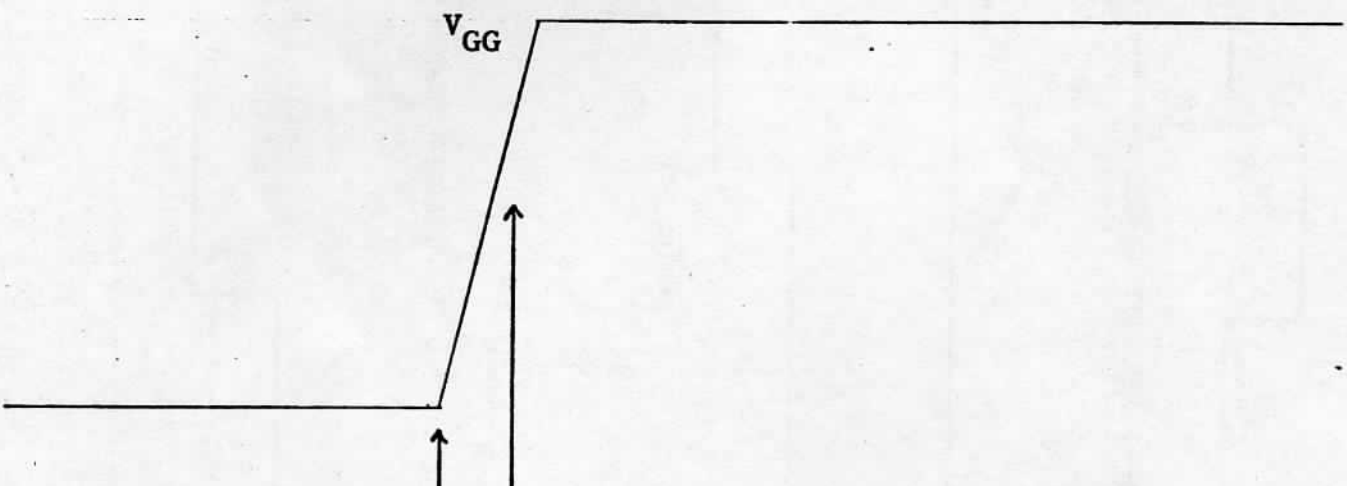
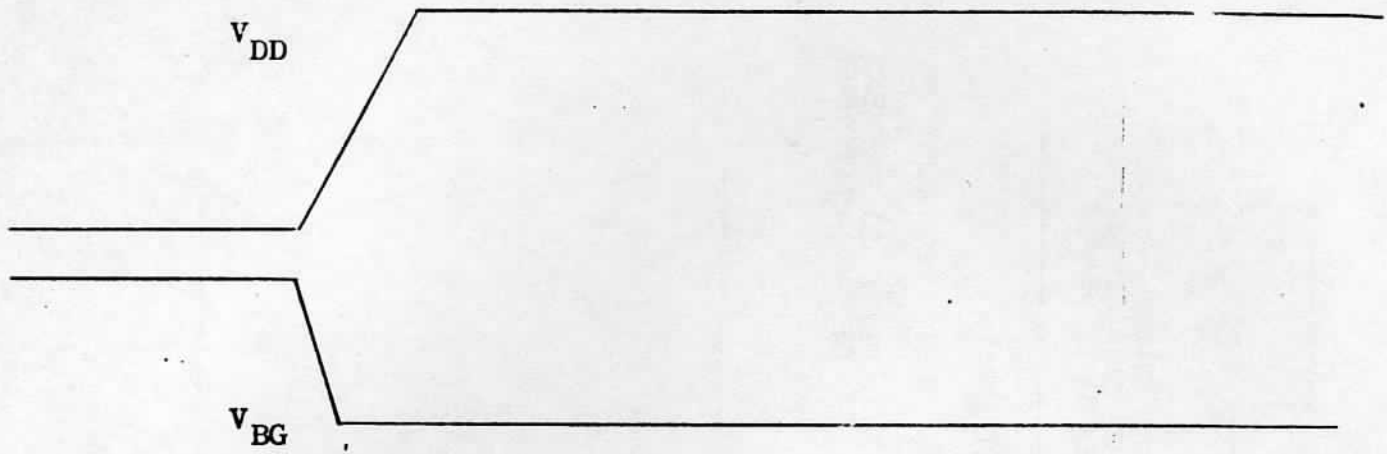
The clock input is (as *all* inputs are) TTL compatible. That is, no external pullup resistors are normally required. (But see "Data Bus Application Hints" for special cases.) It should be noted that to provide a fast clock edge, the internal clock is pulled up with a current of approximately 3 mA.

Power supplies must turn on as shown in the Nano Processor Turn-On Valid Start-Up Sequence Diagram.

# INTERRUPT SYSTEM TIMING



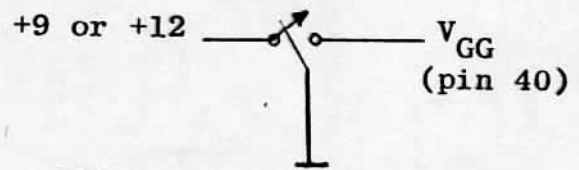
Nano Processor Turn On  
Valid Startup Sequence



$V_{DD}$  and  $V_{BG}$  valid  
before  $V_{GG}$  applied

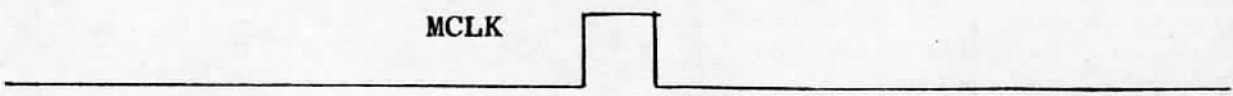
$V_{GG}$  rise time less  
than 1 microsecond

Sample Circuit (Schematic)



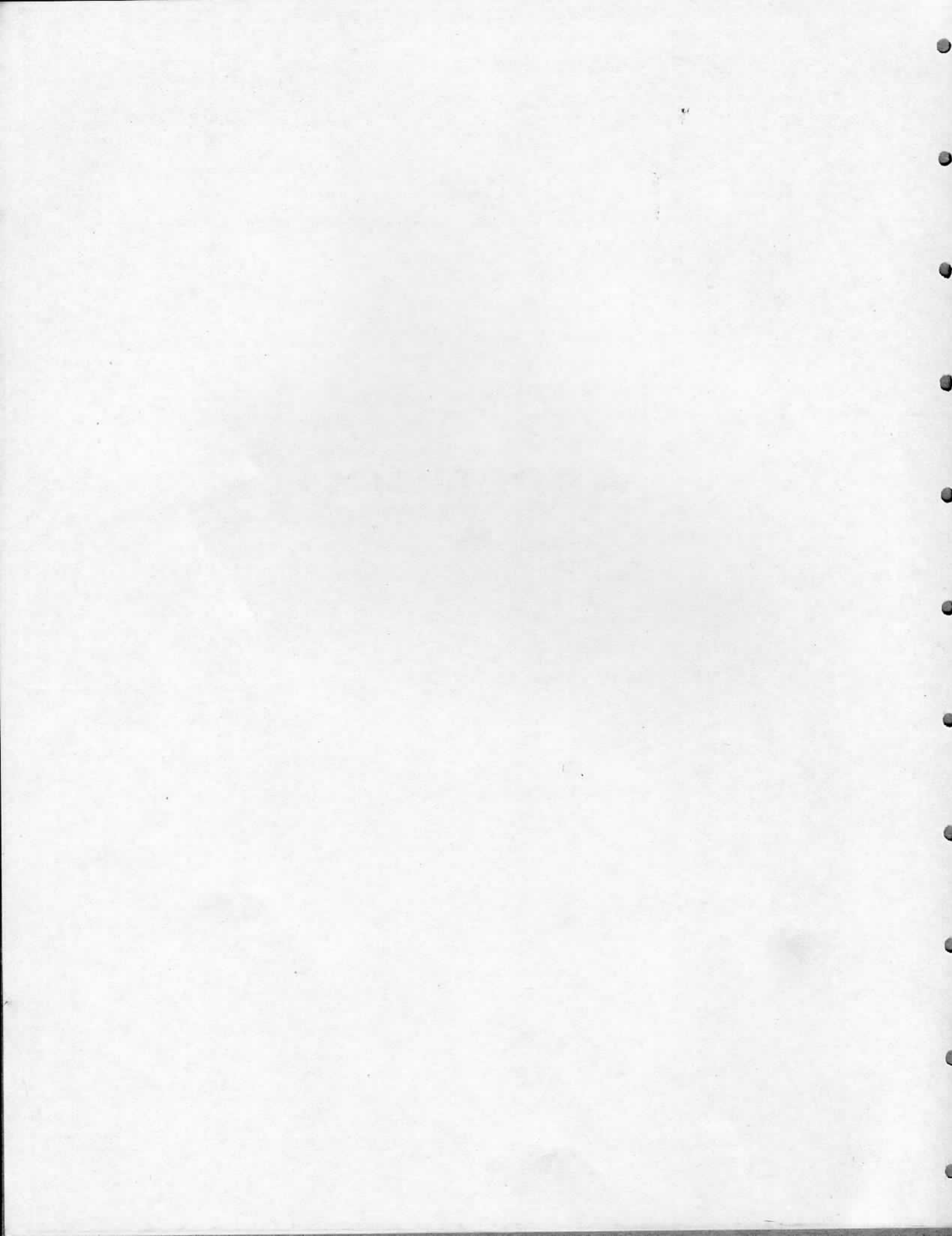
All supplies are valid  
and clock is running

After  $V_{GG}$  is valid  
at least one clock  
pulse must occur  
within 4 microseconds



**APPENDIX**

**A**



# NANOPROCESSOR SPECIFICATIONS

Revision Date 11/10/75

1820-1692 NP-A <sup>\$18.00</sup>  
(500ns)

1820-1691 NP-B <sup>\$13.00</sup>  
(750ns)

	MIN	TYP	MAX	MIN	TYP	MAX
$V_{GG}$	11	+12	12.5	8.5	+9	11
$I_{GG} (mA)^{12}$		30	40		20	26
$V_{GG}$	4.75	+5	5.5	4.5	+5	5.5
$I_{DD} (mA)^{13}$			110			90
$V_{BG}$	-2	-3	-5	-2	-3	-5
$P_d (mW)$		800	1000		550	650
$I_{BG} (\mu A)$			500			350
CLK ON	100ns	100ns	1μsec.	135ns		1μsec.
CLK OFF	100ns		∞	175ns		∞
CLK $\tau$	250ns		∞	375ns		∞
$T_{PA1}$	25ns			25ns		
$T_{PA2}$	60ns	95ns	125ns	75ns	140ns	180ns
$T_{PGH}$	15ns	20ns	30ns <sup>35V</sup>	15ns	27ns	45ns
$T_{PGL}$	5ns	15ns	25ns <sup>30V</sup>	5ns		30ns
$T_{IP}^8$	35ns			50ns		
$T_{AA}$	90ns			145ns		
$T_{EA}$	35ns			60ns		
$T_{DC}$	40ns	90ns	150ns	40ns	110ns	225ns
$T_{DS1}$			85ns			120ns
$T_{DS2}$	30ns		85ns	30ns		120ns
$T_{DI}$	40ns			60ns		
$T_{DO}$			150ns			215ns
$T_{DV}$	40ns			50ns		
$T_{IV}^{14}$	30ns			40ns		
$T_{IRR}^{14}$	30ns			40ns		
$T_{IAH}$			115ns			160ns
$T_{IAL}$			100ns			140ns
$T_{VA}^9$	95ns			155ns		
$T_{V1}$	40ns			60ns		
$T_{IE}$			250ns			375ns
AMBIENT TEMPERATURE	Moving air		80°C			80°C
	Still air		70°C			70°C

I.C. will be marked →

CLOCK AT <sup>10,11</sup>  
TYP. VOLTAGES

PROG. ACCESS

DCIO

DS & R/W

DATA IN

DATA OUT

INTERRUPT acknowledge

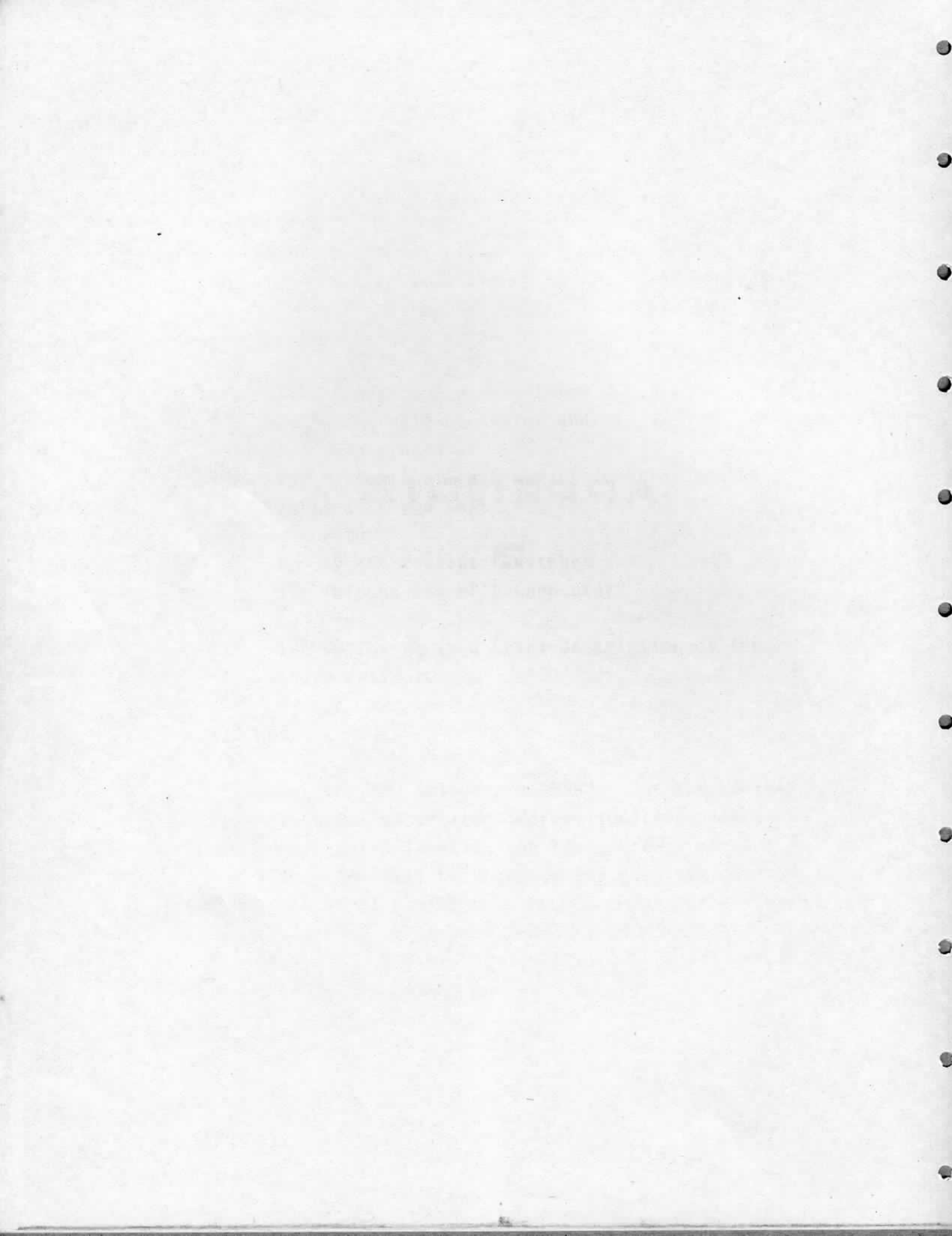
AMBIENT TEMPERATURE

1. All outputs can sink 1.6ma at 0.6 volts or less.
2. Inputs have own pull-ups and may require up to 1.6ma to be sinked when input low is 0.4 volts. See MCLK for an exception.
3. MCLK may require up to 3.0ma to be sinked when low (0.4 volts) and external ckt may have to provide a pull-up capability to 5.0 volts for high speed operation.
4. Data bus speed vs. capacitance must be treated in accordance with data bus application hints.
5. It is preferred that other outputs drive less than 20pF for max. speed; however, 30pF is usually acceptable.
6. All input levels must equal or exceed 4.0 volts bilevel and  $\leq 0.4$  volts low levels.
7. Turn-on must be in accordance with "Turn-on Methods".
8. TEA = CLK off - TIP - TPGH; TAA = CLKT - TIP - TPA2.
9. TVA = CLKT - TIAH - TVI.
10. Max. pulse r.t. 50nsec. up to 4.0V; Max. pulse f.t. 100nsec. down to 0.8V.
11. Pulse ht. = 5V; approx. rise & fall times 10nsec. (test).
12. At VGG = 12.0 volts.
13. At VDD = 5.0 volts.
14. Min. & max. delay times from Interrupt Request Input till fetch phase of first instruction (vector has already been serviced) is: min. = TIV + CLK PW + CLKT and max. = TIV + CLK PW + 3 CLKT.

**APPENDIX**

**B**





ROM-RAM SIMULATOR

NOTE

DUE TO THE COMPLETION OF THE NANO PROCESSOR  
PROJECT, LID WILL NO LONGER SUPPORT OR PROVIDE  
ADDITIONAL INFORMATION ON THE ROM-RAM SIMULATOR  
AFTER JAN. 1, 1976.

## ROM - RAM SIMULATOR

ROM - RAM Simulator is a block of memory that simulates a ROM. A block diagram of the ROM - RAM is provided.

As the block diagram indicates ROM -RAM can be addressed in three ways:

1. Through address switches
2. By an HPIB connector and
3. By a processor

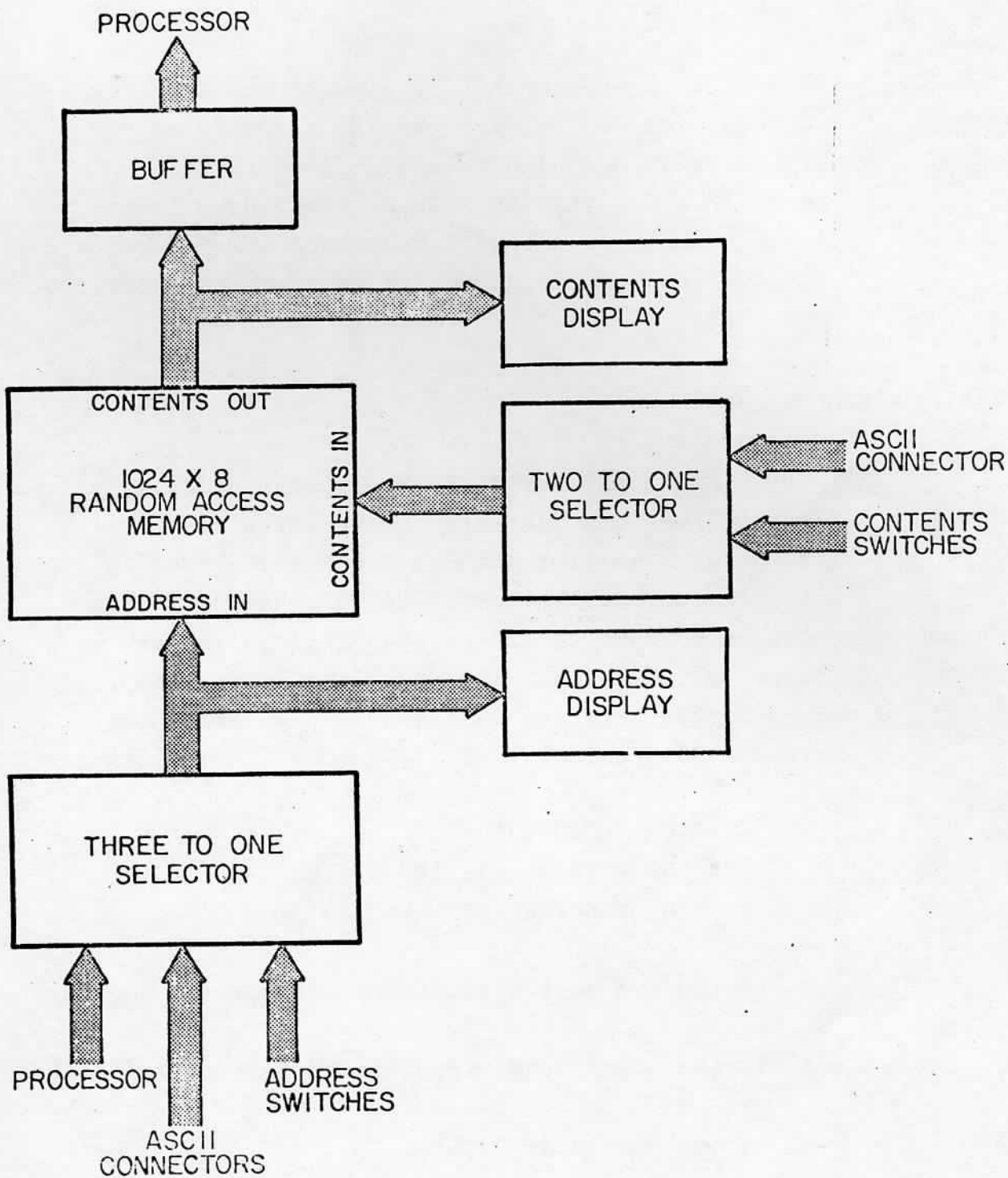
The contents of each memory location can be entered in two ways:

1. By the contents switches
2. Through the HPIB Connector

The following is a brief description of the above options.

### SWITCHES

Set the I/O Selector on "SW". Set the address switches to desired address location. The designated location and the contents of that location will be shown on the displays. If change of contents is desired, set the contents switches to the new contents and press the "WRITE" button. The new contents will be displayed on the contents display.



## OUTPUT

To output the contents of the memory to other devices such as a processor, set the I/O Selector on "OUT". Provide a 10 bit high true signal on the edge connectors PA0 through PA9 address lines. A high enable signal will cause the ROM-RAM to output the contents of the addressed location on the output bus.

## HPIB CONNECTOR

ROM - RAM Simulator is HPIB compatible, that is: the address and contents can be given to the ROM - RAM Simulator through the HPIB. Normally the listen address is set to "3", however; this can be changed to "2" by changing the jumper wire on the board. The following is an example program for writing in the ROM - RAM Simulator from a 9830 calculator.

```
10  CMD"?U3"  
20  WRITE (13,30) A,B,C;  
30  Format 3B
```

A is the two most significant bits of the address.

B is the eight least significant bits of the address.

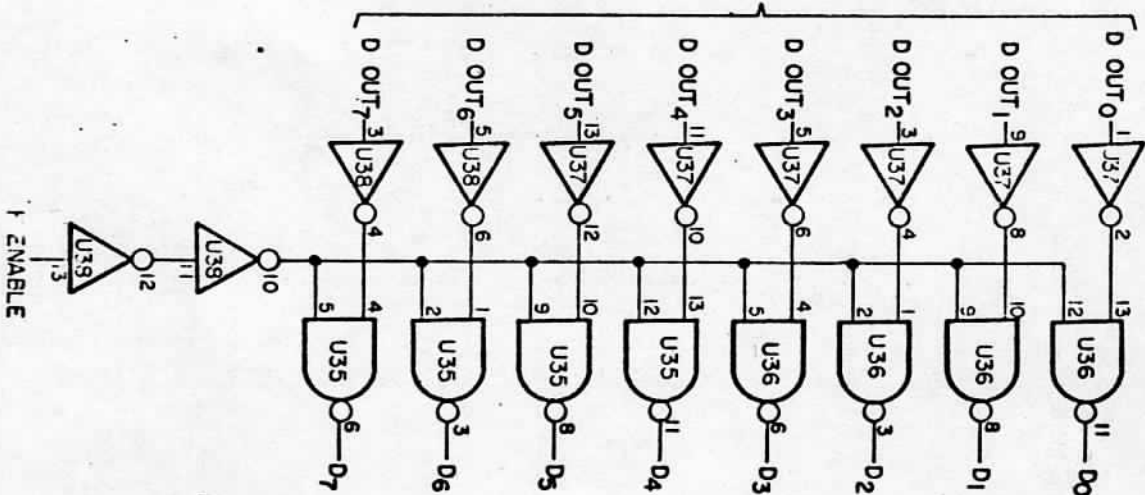
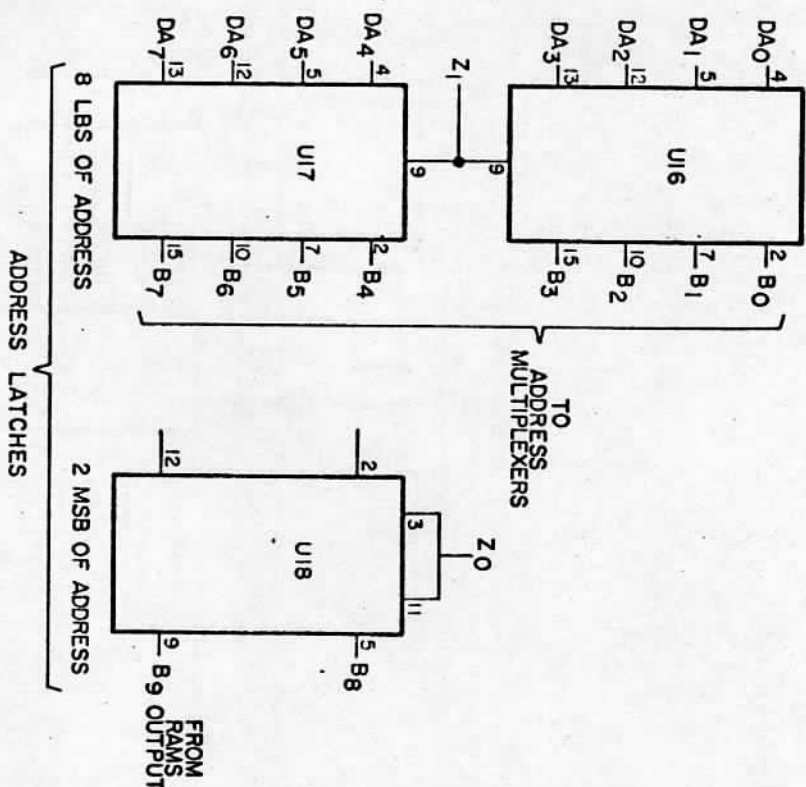
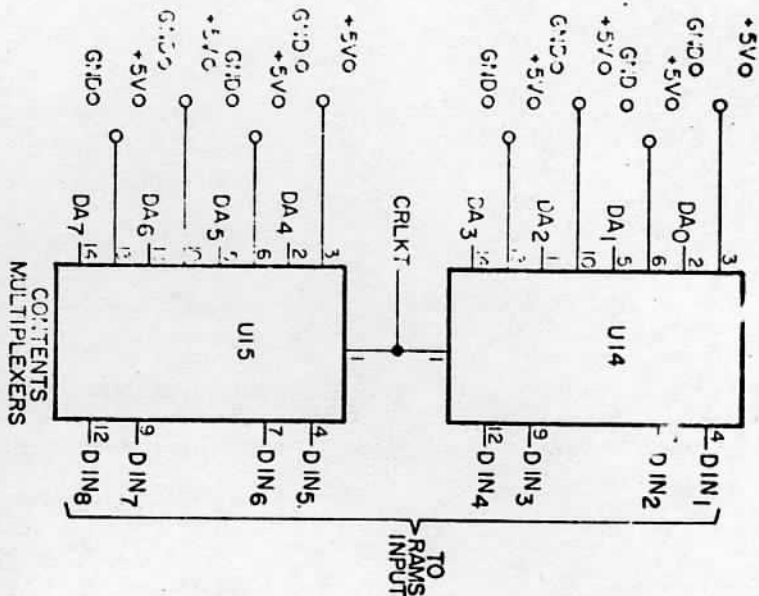
C is the contents.

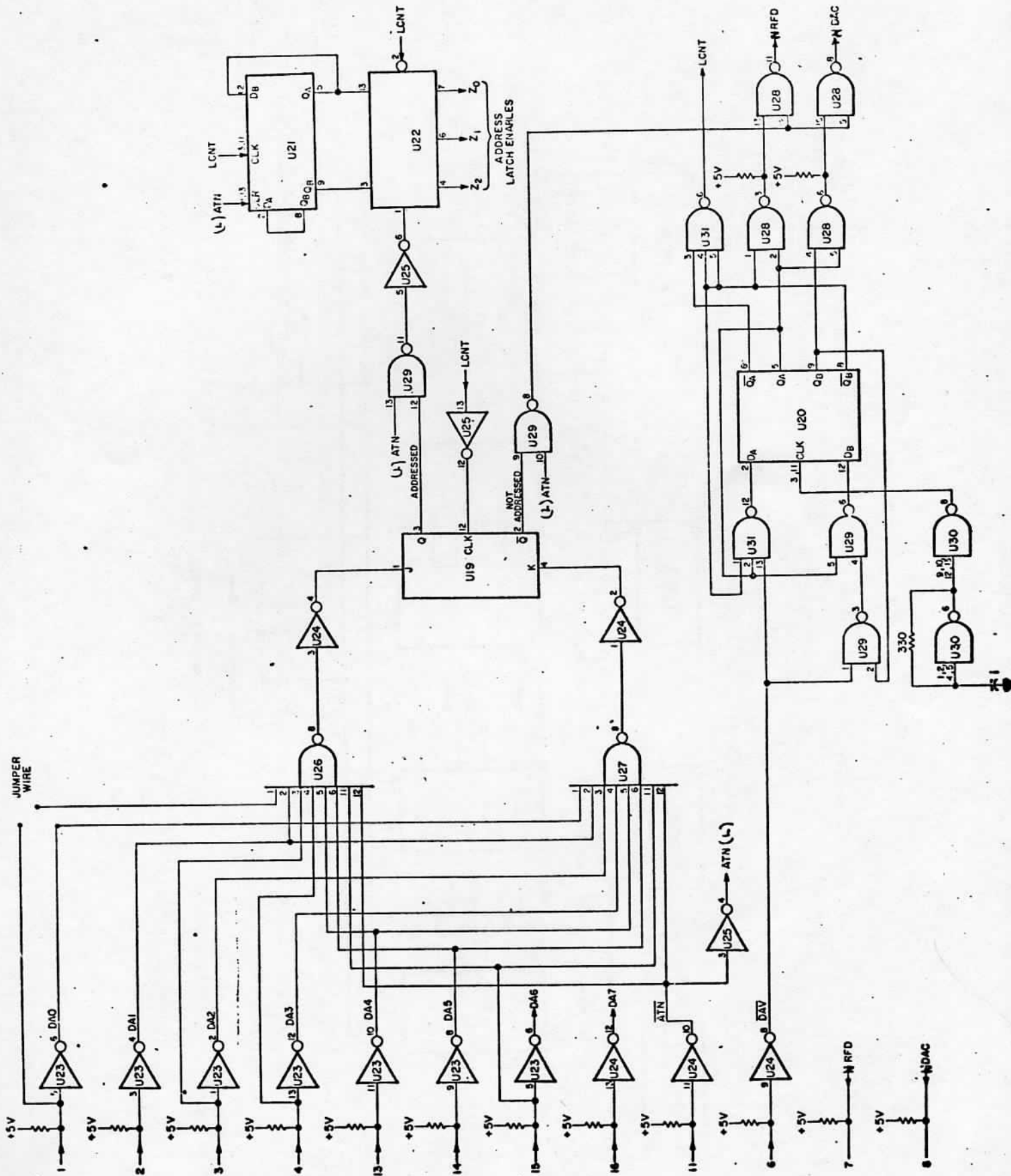
A, B, and C are the decimal equivalent of the binary code.

Example:

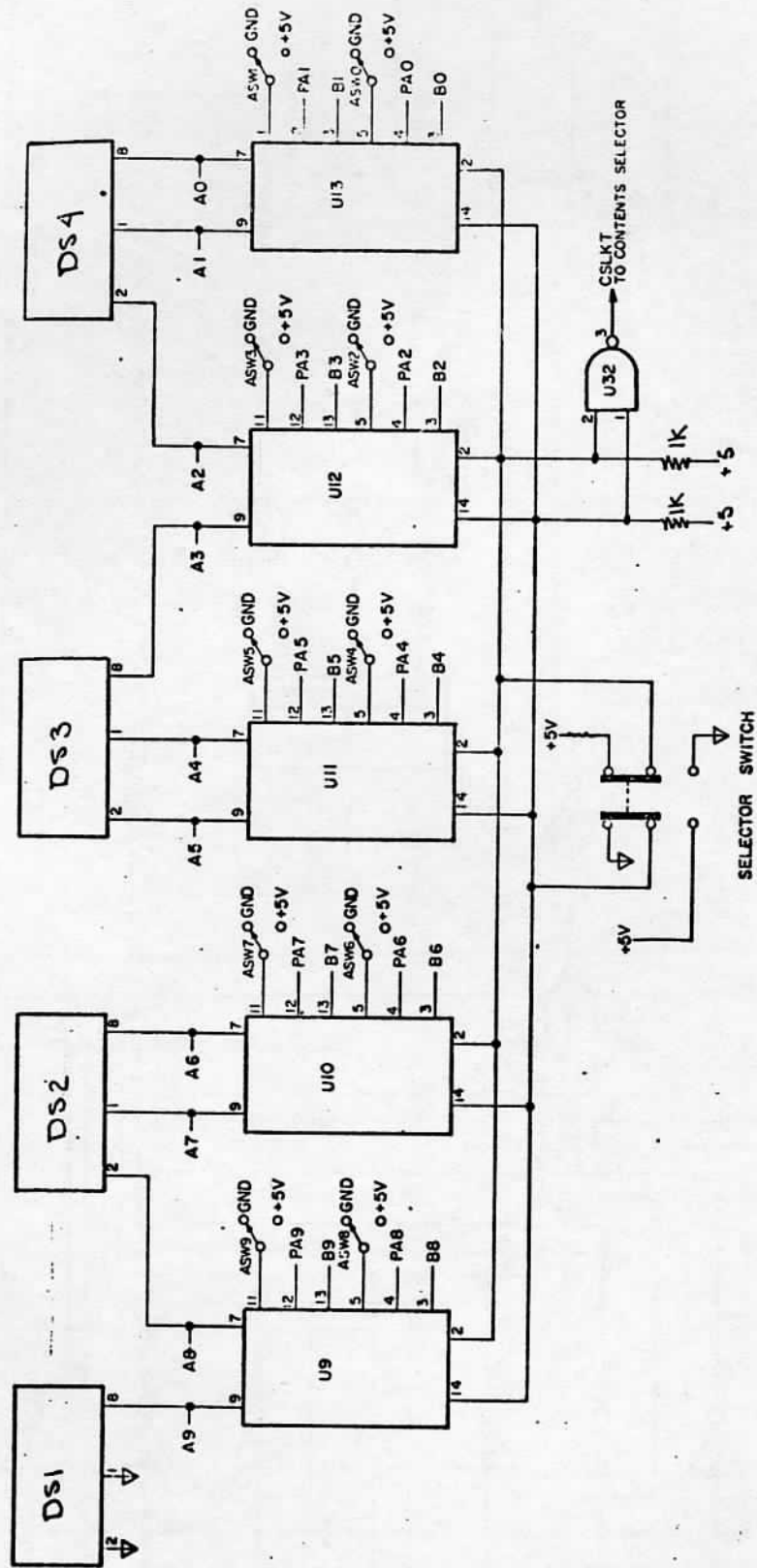
Utilizing a 9830 calculator, the following program could be used to write individual codes on the ROM - RAM Simulator.

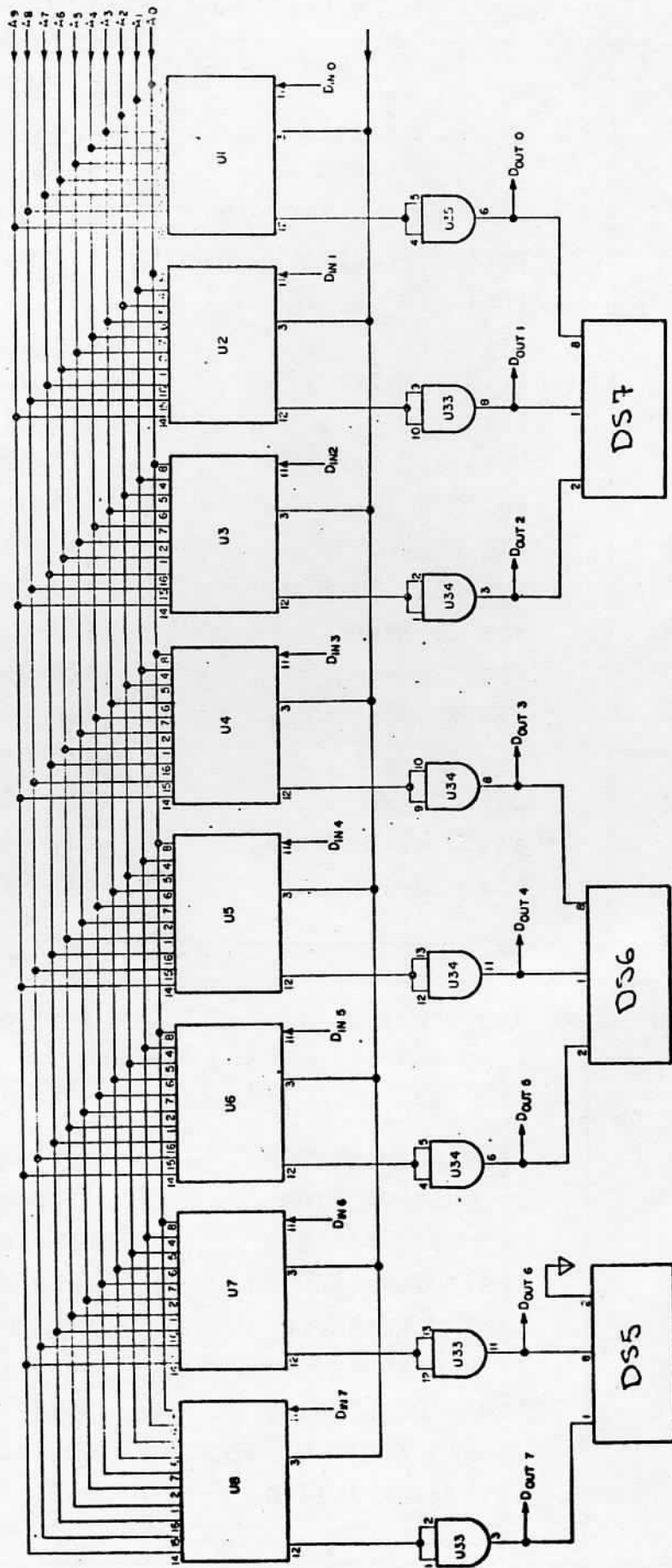
```
10     DISP "ENTER ADDRESS";
20     INPUT I
30     DISP "ENTER CONTENTS";
40     INPUT C
50     A = B IAND (ROT (I,8),3)
60     B = B IAND (I, 255)
70     CMD "?U3"
80     WRITE (13,90)A,B,C;
90     FORMAT 3B
100    GO TO 10
```











ROM-RAM SIMULATOR COMPONENTS

IC#	DESCRIPTION	HP PART NO
1-8	2102 INTEL RAM	1820-1078
9-13	74153 4-1 MULTIPLEXER	1820-0620
14,15	74157 2-1 MULTIPLEXER	1820-0839
16,17	74175 HEX D, FF	1820-0839
18	7474 D-FF	1826-0077
19	74107 J-K FF	1820-0281
20,21	7474 D-FF	1826-0077
22	74155 2-4 DECODER	1820-0738
23-25	7404 HEX INVERTOR	1820-0174
26,27	7430 8INPUT NAND	1820-0070
28	7438 O.C. 2I NAND	1820-0621
29	7400 2I NAND	1820-0054
30	7413	1820-0537
31	7410 3 INPUT NAND	1820-0068
32	7400 2I NAND	1820-0054
33,34	7408 2I AND	1820-0511
35,36	7438 O.C. 2I NAND	1820-0621
37,38	7404 HEX INV	1820-0174

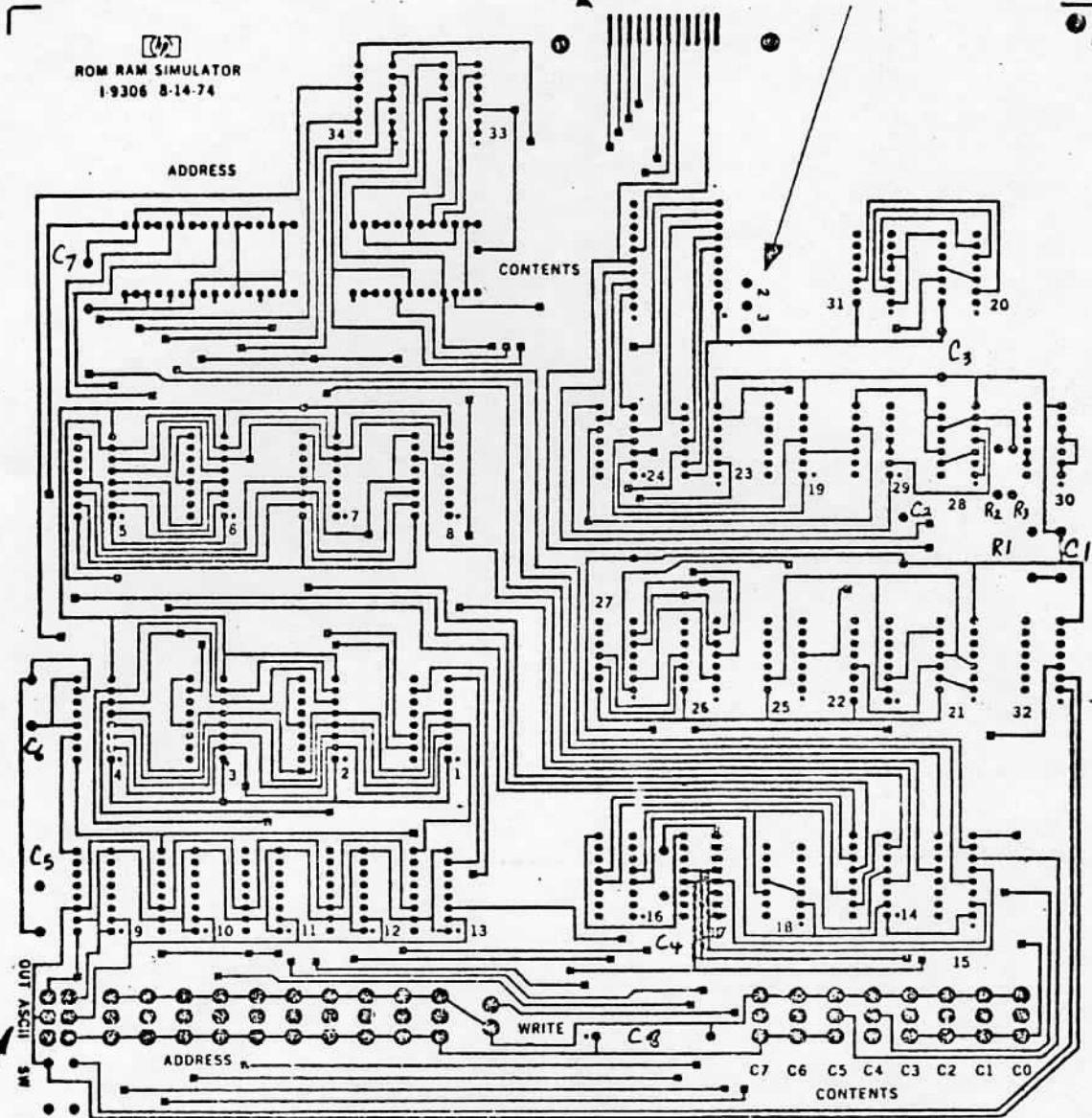
I/O SW	DPDT SWITCH 7203	3101-0939
ADD CONT. SW	SPDT SWITCH 7101	3101-1258
WRITE SW	SPST PUSH BUTTON	3101-0063
C <sub>8</sub>	.47 UF CAPACITOR	0180-0097
C <sub>2</sub> -C <sub>7</sub>	.47 UF CAPACITOR	0160-0174
	RESITOR-PACK	1810-0136
R <sub>1</sub>	330 Ω RESISTOR	0683-3315
R <sub>2</sub> -R <sub>4</sub>	1K RESISTORS	0683-1025
	7300 LED DISPLAY	5082- 7300
	24 PIN ASCII CONNECTOR	1251-3283
C <sub>1</sub>	.01 UF CAPACITOR	0150-0093

# ROM - RAM SIMULATOR BOARD

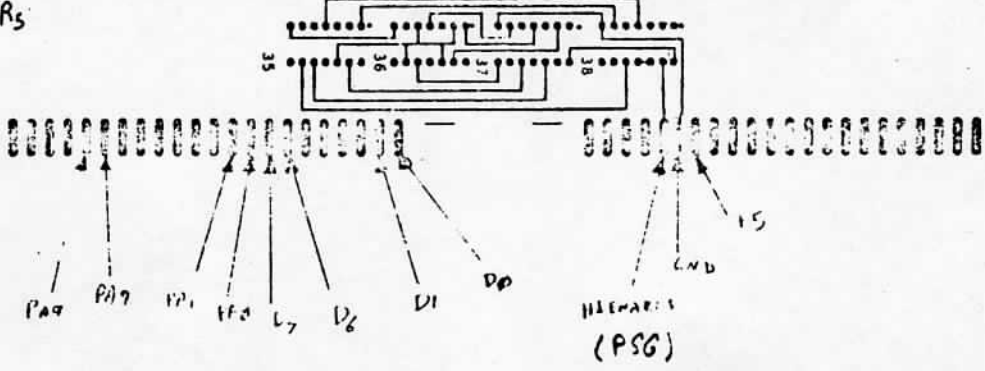
HP IB CONNECTOR

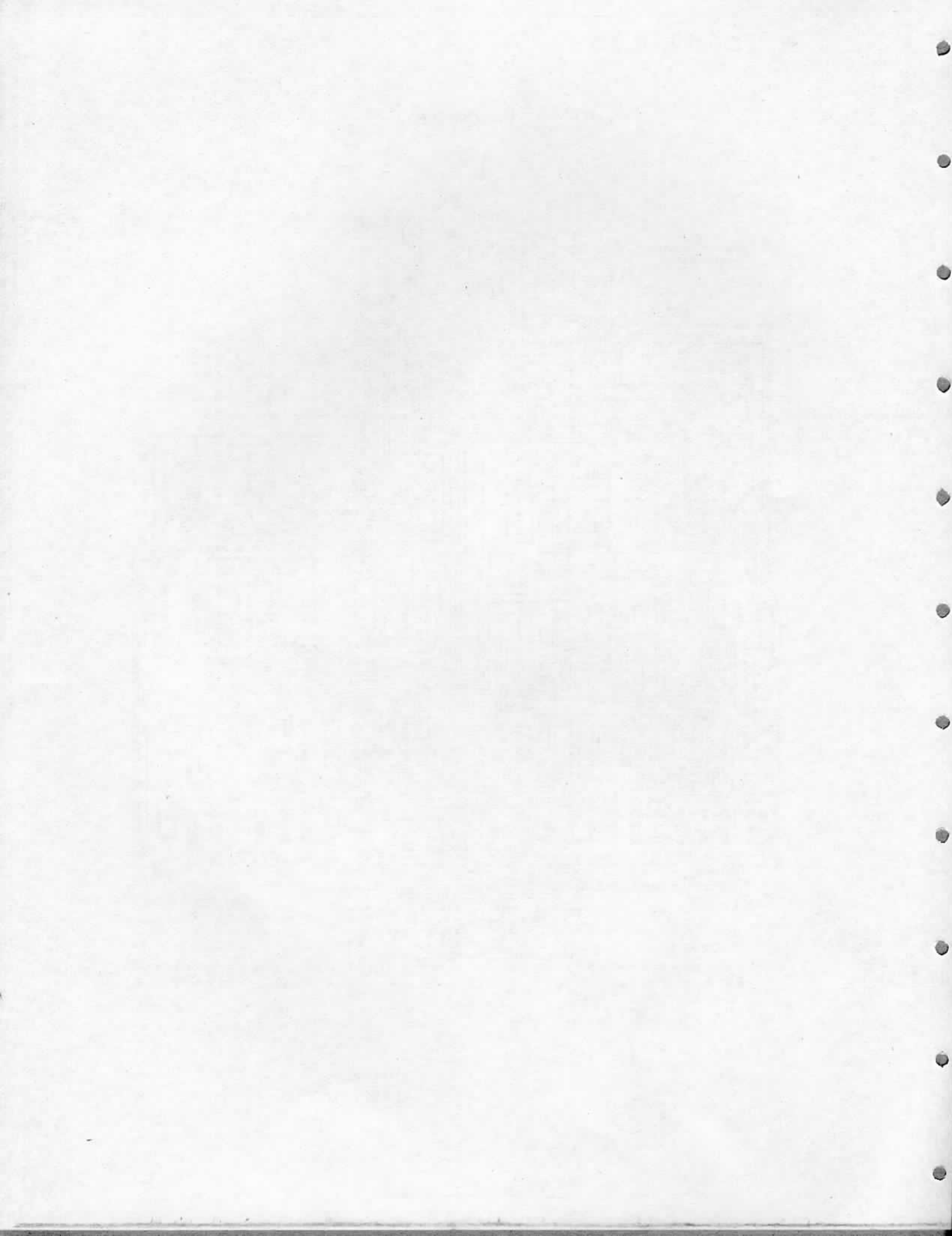
JUMPER WIRE

ROM RAM SIMULATOR  
1-9306 8-14-74



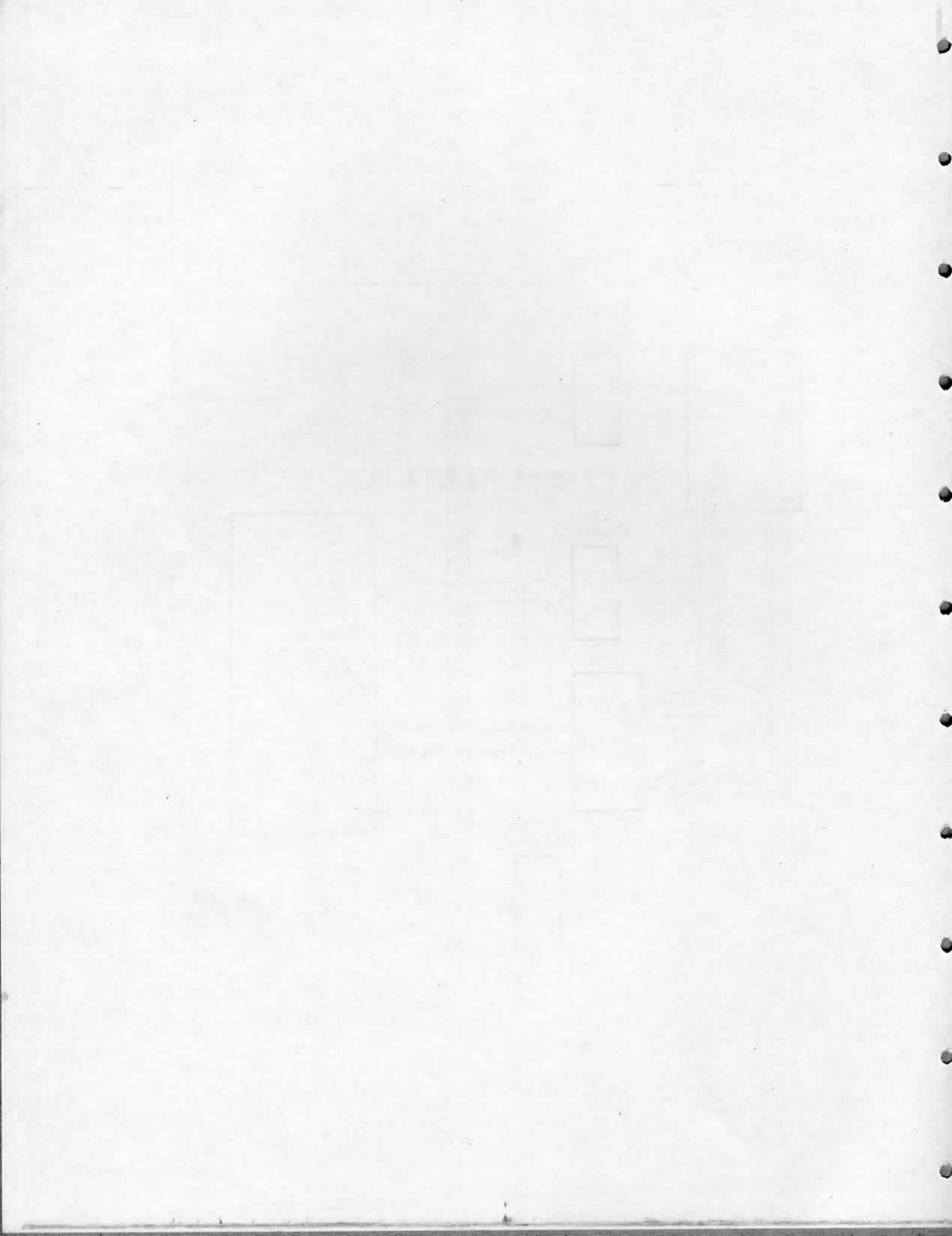
I/O  
SELECTOR  
SWITCH





**APPENDIX**

**C**



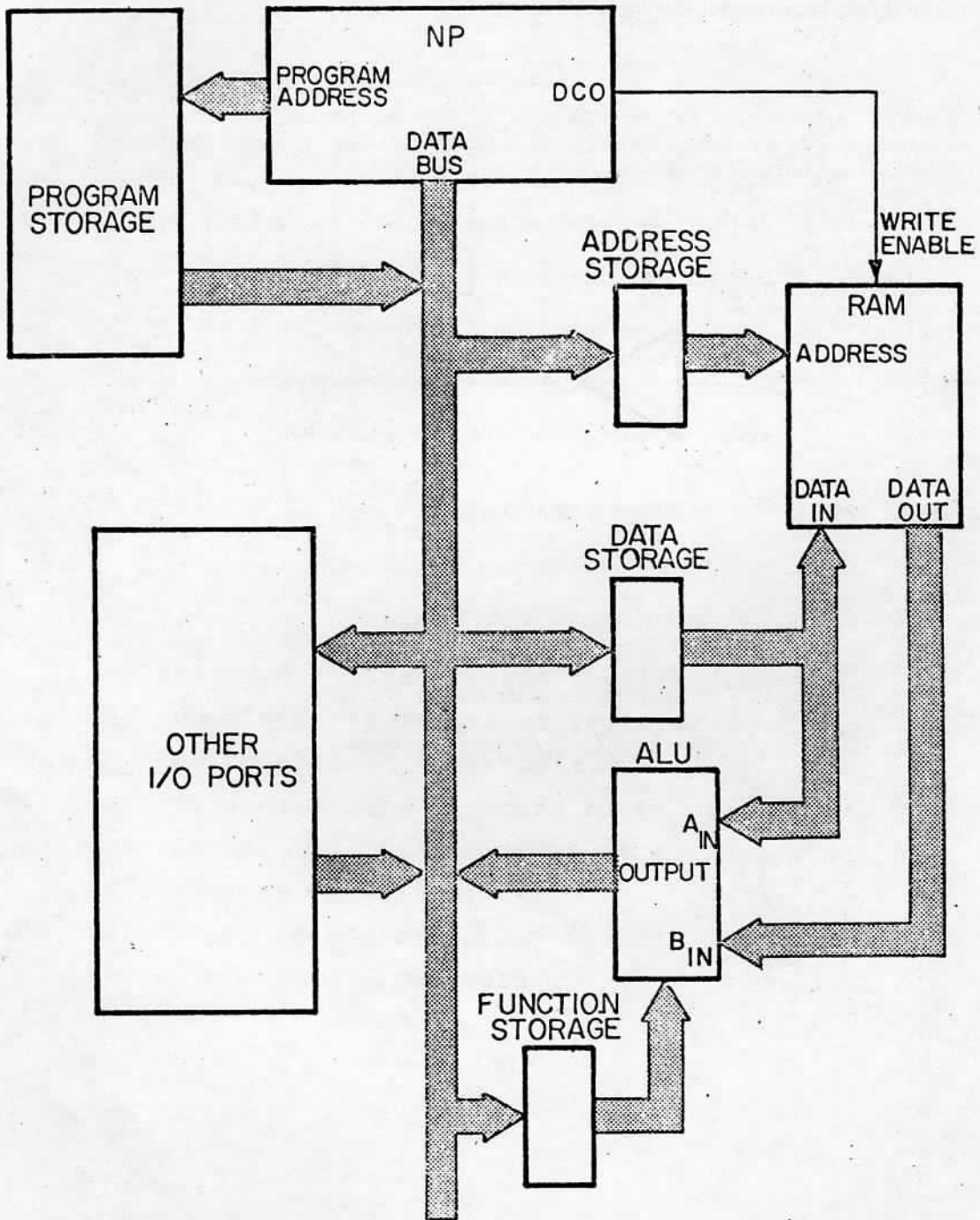
## AN ARITHMETIC CAPABILITY FOR THE NANO PROCESSOR.

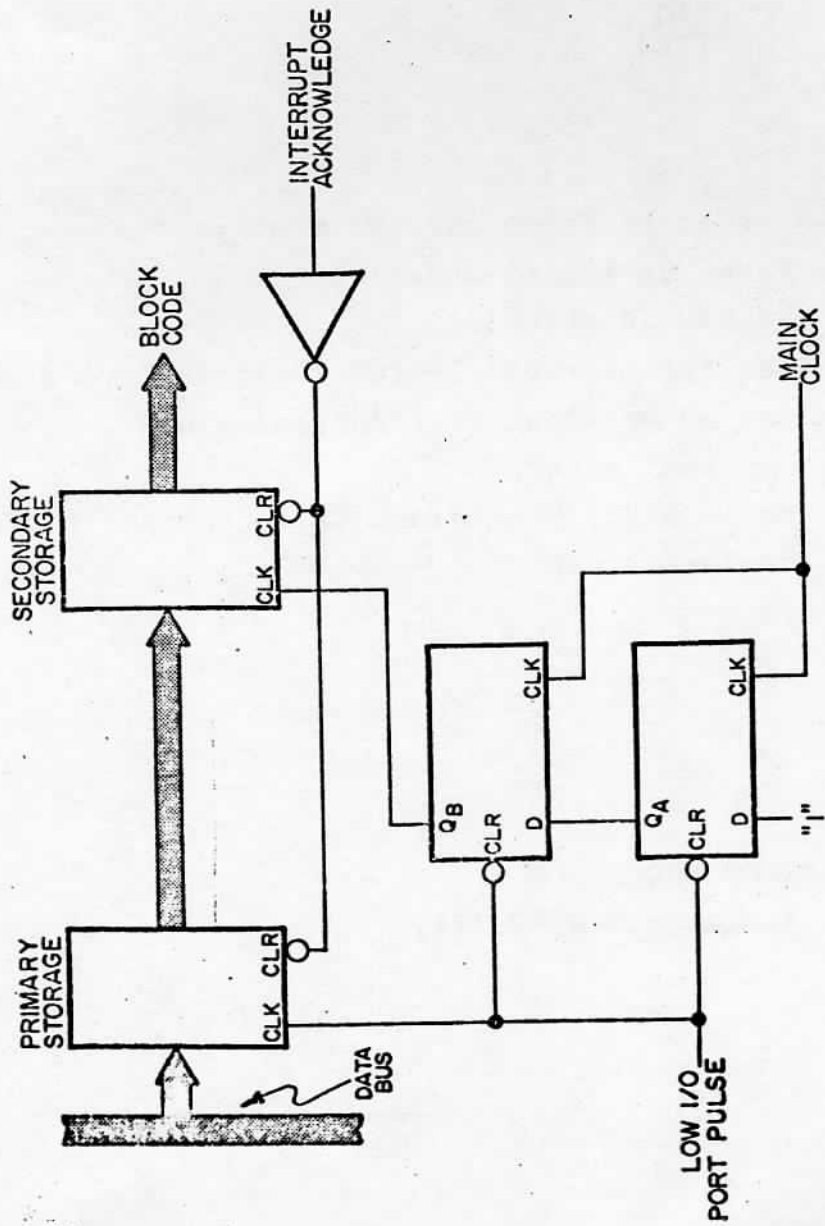
By using 4 or 5 I/O parts and minimum external hardware, the capacity of the Nano Processor for data manipulation and storage is greatly increased. Choose a RAM size (minimum one word) and an ALU capability to suit your needs.

Let ADDR be the select code of the Address Storage Latch.  
DATA be the select code of the Data Storage.  
ALU be the select code of the Arithmetic Unit.  
FN be the select code of the Function Storage.  
and RAM be the WRITE ENABLE line of the RAM.

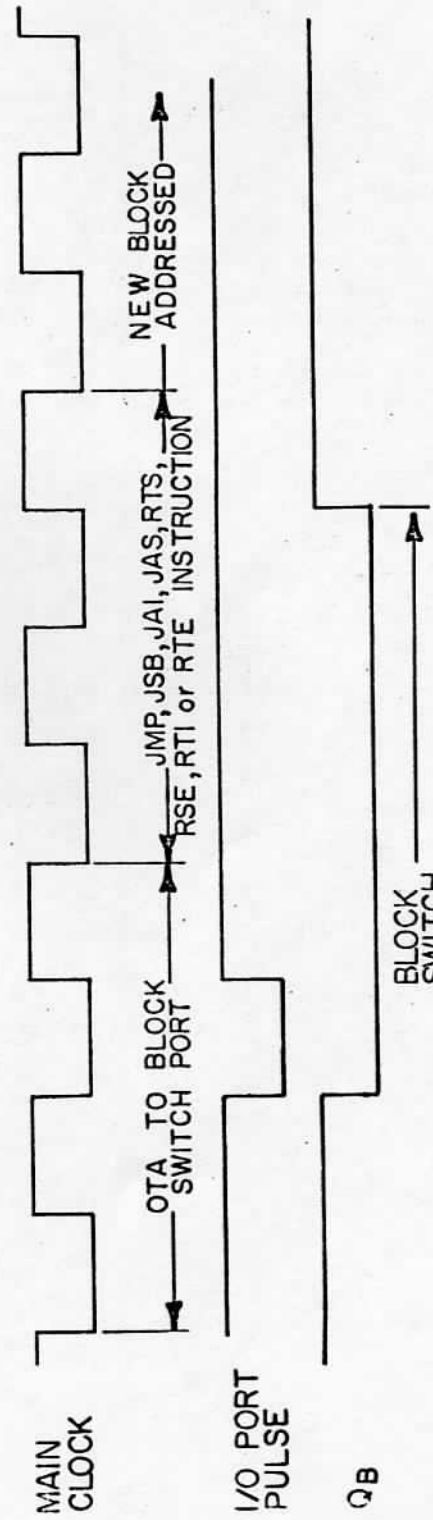
OTA ADDR \*ADDRESSES RAM LOCATION A  
OTA DATA \*Lets A be one argument of the Arith/Logic Unit  
OTR FN, "+" \*Selects function "+" for the ALU  
INA ALU \*Puts the result in A  
\*Maybe it also puts the result in DATA  
OTA RAM \*Puts DATA in RAM Location (ADDR)



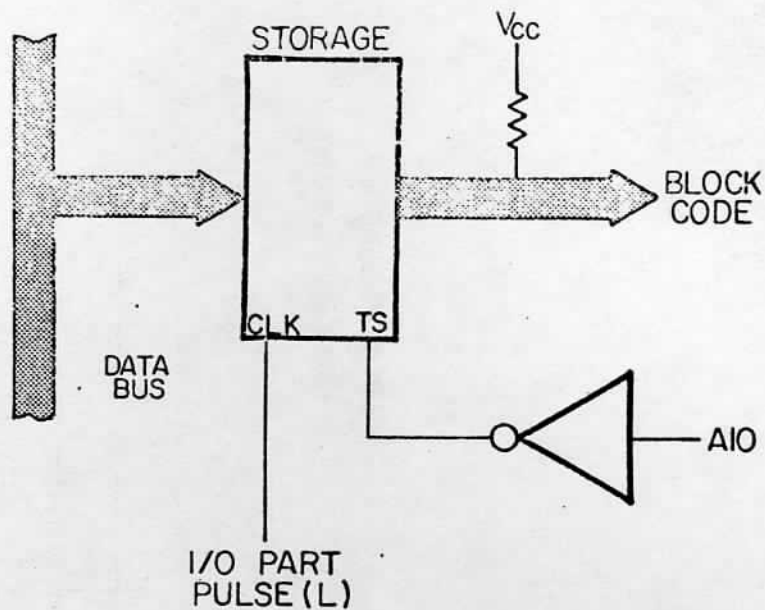




NANO PROCESSOR  
BLOCK SWITCHING (I)



NOTE  
INTERRUPTS MUST BE DISABLED  
DURING BLOCK SWITCHING

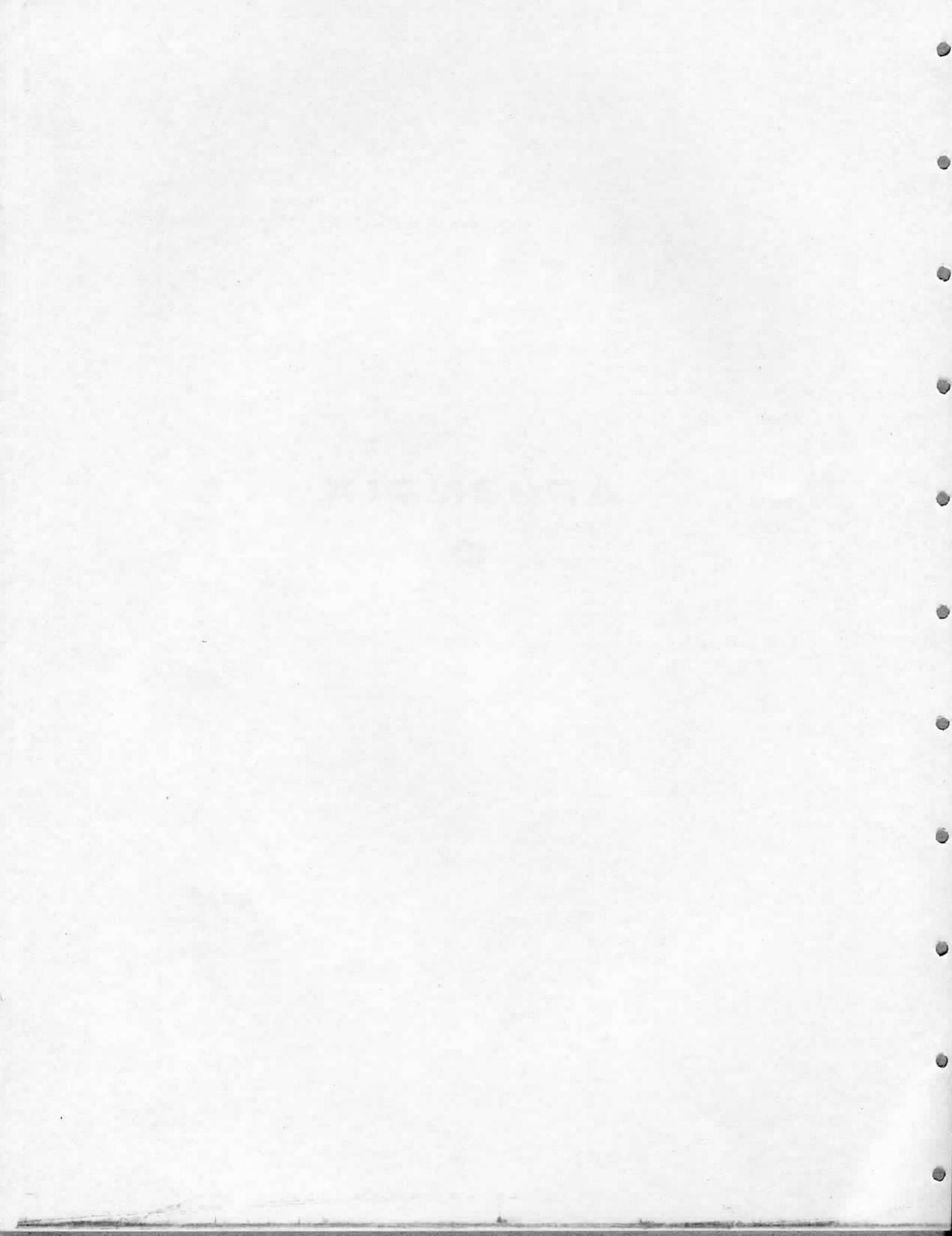


1. Use one quad tristate latch for 16K memory.
2. The new block may be set at any time.
3. No need to disable interrupts.
4. A10 is used as the "current block" indicator.  $A10 = \phi$ .
5. Subroutines may be in block H or current block.  
Return is to current block.
6. Interrrups are to block H. Return is to current block.
7. Blocks are 1024 words each.

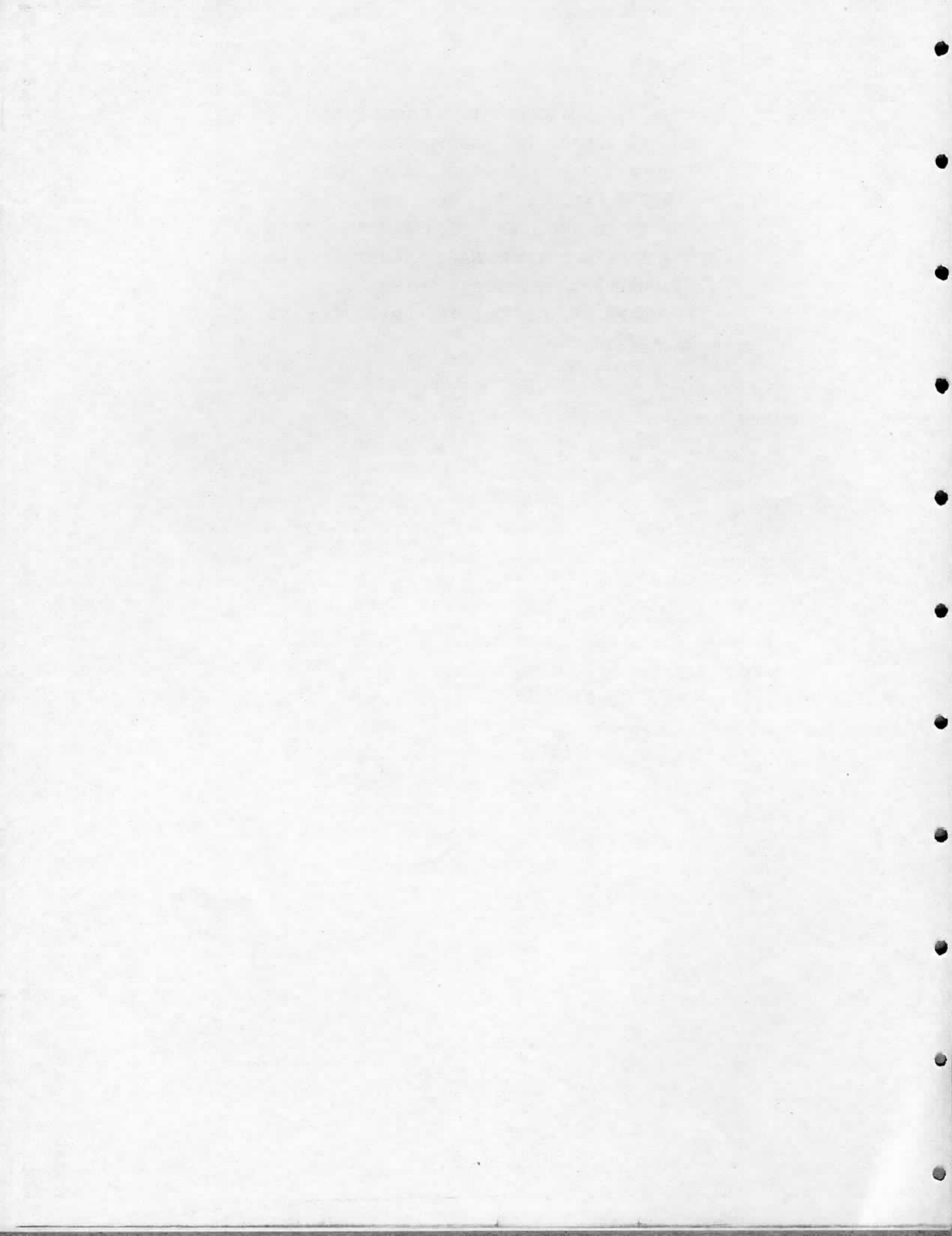
**NANO PROCESSOR  
BLOCK SWITCHING (II)**

**APPENDIX**

**D**



THIS APPENDIX CONTAINS INFORMATION  
ON THE USE OF THE NANO PROCESSOR  
EDITOR, ASSEMBLER, AND LOADER FOR  
A 9830A CALCULATOR. ALSO THE  
ASSEMBLER AND LOADER FOR A 2100 DOS  
III SYSTEM FOR THE NANO PROCESSOR  
IS COVERED. TO OBTAIN THESE  
PROGRAMS SEE GENERAL INFORMATION  
IN APPENDIX E.



H P PRIVATE

H P PRIVATE

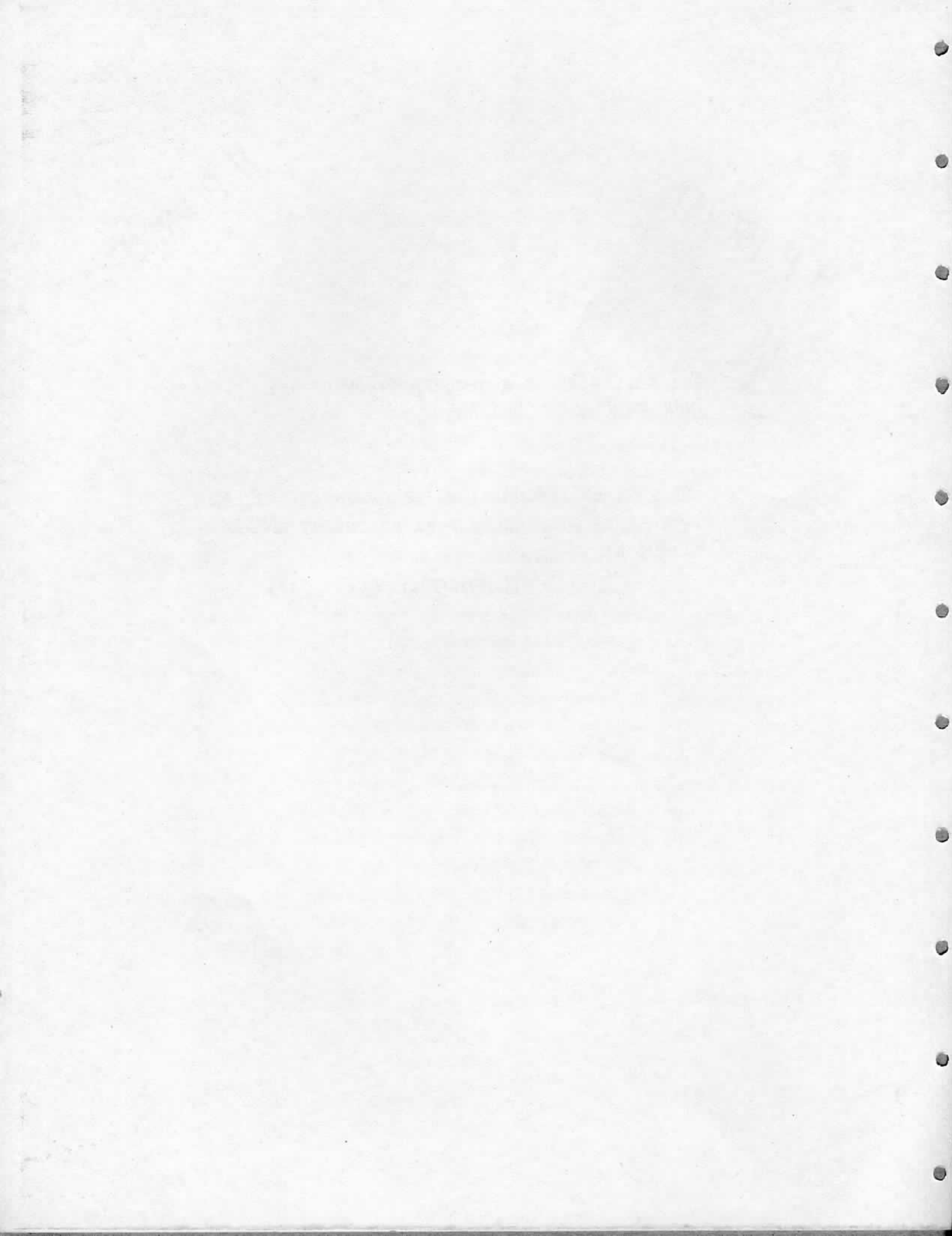
THE FOLLOWING IS A DOCUMENTATION OF THE  
NANO PROCESSOR SOFTWARE

FOR FURTHER INFORMATION, RECOMMENDATIONS, OR  
IN CASE OF DIFFICULTY PLEASE CONTACT KAMRAN  
FIR00Z AT

303-667-5000 EXT. 2873

OCTOBER/1974





NANO PROCESSOR EDITOR  
A NEW EDITOR FOR THE 9830 CALCULATORS

PURPOSE: To generate or edit a source program on the  
9830 calculators.

ROM REQUIREMENTS: ADVANCE PROG. #11279B,  
STRING VARIABLE #11274B

MEMORY REQUIREMENTS: 8K

OBJECTIVE: At the present time the programs stored on  
the cassette tapes of the 9830 calculators  
cannot be accessed by any program as a data  
file. In many applications it is quite  
desirable to be able to edit a program in  
whatever programming language desired, and  
store the generated source program as a  
data file on the cassette tapes. These  
programs can then be 'called in' by an  
assembler or a similar processor to be  
assembled.

The primary objective of this editor is to  
generate source programs for the Nano Pro-  
cessor Assembler; however, the program is  
versatile enough so that it can produce source  
program for other Micro Processor Assemblers  
as well.

Kamran Firooz  
August 1974

SYNTAX: The editor operates in two separate modes "FILE EDIT" and "LINE EDIT". The "FILE EDIT" is used to edit the entire file; the "LINE EDIT" is used to edit individual lines.

The commands shown on the lower portion of the "SPECIAL FUNCTION CARD" are executed during the "FILE EDIT" mode, and the commands shown on the upper portion are executed during the "LINE EDIT" mode. All of the commands shown on the card are immediately executable. The shift key is not needed to execute any of the commands in either mode.

A maximum of 140 lines can be edited on each file. The program will notify the user if this limit is exceeded. The maximum length of each line is 32 characters. Any character past this limit will be ignored. a "/" is used to designate the physical end of a line. If "/" is not found in each line, the editor will insert a "/" in the 32nd character. For a more efficient use of line length, a ":" is used as the "PSEUDO-END of LINE" character. This character will be recognized by the Nano Processor Assembler.

```
CLA /  
TAG STA 12 /  
JMP FAST /  
is equivalent to:  
CLA :TAG STA 12 : JMP FAST /
```

The physical file length is determined by the "EOF" statement. If such a statement is not found, the file length is taken as 140 lines.

The LOAD and STORE commands require a cassette file length of at least 2240 words.

After each command is entered, any negative number that is typed will cause the program to go back to the "FILE EDIT" mode without executing the requested command.

LIST OF COMMANDS OF "FILE EDIT" MODE

- INPUT: To generate a new file or replace certain lines of a previously generated program. "EXIT" command will terminate the INPUT command.
- STORE: To store the generated source program in a desired file on cassette tapes.
- LOAD: To load a previously stored program into the editor.
- DELETE: To delete line no.  $N_1$  to  $N_2$  of the generated program.
- INSERT: To insert  $N$  lines after line  $N_1$  of the generated program.
- XREF: To list a cross reference table of a character string.
- LIST: To list the entire program.
- LIST  $N_1$  : To list line no.  $N_1$  to the end of the program.
- LIST  $N_1, N_2$ : To list line no.  $N_1$  to  $N_2$  of the program.
- LINE EDIT: To edit an individual line.
- HELP: To help the user with the different commands and their syntax.

- LISTP; LISTP  $N_1$ ; LISTP  $N_1, N_2$ : To perform the same function as LIST; LIST  $N_1$ ; and LIST  $N_1, N_2$ , except that all the statements separated by the "PSEUDO-END OF LINE" character (:) will be listed on separate lines.
- LABEL: To list all of the labels used throughout the source program.
- CHANGE: To change a character string throughout the source program.

NOTE 1: The line numbers are only generated during the print period; they are not stored with the program.

NOTE 2: To delete or list a single line set  $N_1 = N_2$ .

NOTE 3: The total length of the file will be printed after each LIST or LISTP command.

NOTE 4: If as a result of an unacceptable command such as wrong file length an error occurs which stops the calculator, the program can be restarted without losing the current file by "CONT 100".

LIST OF COMMANDS OF "LINE EDIT" MODE

- FORWARD: To move the visible pointer one character space to the right each time it is pressed.
- BACK: To move the visible pointer one character to the left each time it is pressed.
- INSERT: To open up a character space immediately at the visible pointer.
- DELETE: To delete the character space where the visible pointer is located.
- ↓ : To store the edited line and edit the next line.
- ↑ : To store the edited line and edit the line immediately before the present line.
- EXIT: To store the edited line and return to the "FILE EDIT" mode.
- NO EDIT: To return to the "FILE EDIT" mode. All of the changes on the edited line will be ignored.

NOTE 5: To replace the strings followed by the visible pointer, just enter the new string.

NOTE 6: If line one is being edited and command "↑" is executed the program will return to the "FILE EDIT" mode.

NOTE 7: The visible pointer cannot pass the end of line character (/).

The following example is provided in an attempt to familiarize the user with some of the features of the Nano Processor Editor. All of the commands given by the user are underlined

EXAMPLE:

Load the editor program in the 9830 calculator and press RUN. EXECUTE. At the beginning of the execution the program loads the special function keys from the file following the editor program. Then the calculator will display

"FILE EDIT?"

Press INPUT

"STARTING LINE?"

1

"ENTER LINE 1 ?"



Then the following program is typed in:

```
*   EXAMPLE PROGRAM/
*THIS PROGRAM READS A NUMBER IN/
* BCD. AND CONVERTS IT TO BIN./
*
*
START INA DS0 *INPUT THE BCD #/
  STA R5 *STORE THE # IN R5/
  STR 0,0 *CLEAR R0:LOOP LDA 5/
  SAN *SKIP IF ACC#0: JMP OUT/
  DED *DECREMENT IN BCD: STA 5/
  LDA 0: INB *INCREMENT IN BIN./
  STA 0: JMP LOOP:OUT LDA 0/
  OTA DS1 *OUTPUT THE BIN. #/
  JMP START *READ ANOTHER NUMBER/
DS0 OCT 0:R5 OCT 5: END: EOF/
```

After the last line is entered press

EXIT

"FILE EDIT?"

LIST

```
1 * EXAMPLE PROGRAM/
2 * THIS PROGRAM READS A NUMBER IN/
3 * BCD, AND CONVERTS IT TO BIN./
4 *
5 *
6 START JNA DS0 *INPUT THE BCD #/
7 STA R5 *STORE THE # IN R5/
8 STR 0:0 *CLEAR R0:LOOP LDA 3/
9 SAN *SKIP IF ACC#0: JMP OUT/
10 DED *DECREMENT IN BCD: STA 5/
11 LDA 0: INB *INCREMENT IN BIN./
12 STA 0: JMP LOOP:OUT LDA 0/
13 CTA DS1 *OUTPUT THE BIN. #/
14 JMP START *READ ANOTHER NUMBER/
15 DCB OCT 0:R5 OCT 5: END: EOF/
TOTAL LINE NUMBER= 15
```

To list the lines separated by ":" individually type:

LISTP

PAGE 1

\*\*\* \*\*\*\*\*:\*\*\*\*\*

```
1      *   EXAMPLE PROGRAM/
2      *THIS PROGRAM READS A NUMBER IN/
3      * BCD, AND CONVERTS IT TO BIN./
4      *
5      *
6      START INA DS0 *INPUT THE BCD #/
7      STA R5 *STORE THE # IN R5/
8      STR 0,0 *CLEAR R0
9      LOOP LDA 5/
10     SAM *SKIP IF ACC#0
11     JMP OUT/
12     DED *DECREMENT IN BCD
13     STA 5/
14     LDA 0
15     INB *INCREMENT IN BIN./
16     STA 0
17     JMP LOOP
18     OUT LDA 0/
19     OTA DS1 *OUTPUT THE BIN. #/
20     JMP START *READ ANOTHER NUMBER/
21     DSB OCT 0
22     RS OCT 5
23     END
24     EOF/
```

TOTAL LINE NUMBER= 15

Suppose it is desired to output the binary equivalent number to some other devices besides device (DS1).

Then press

INSERT

"INSERT AFTER LINE?"

13

"HOW MANY LINES BE INSERTED?"

1

"ENTER LINE 14 ?"

OTA 2/

"FILE EDIT?"

Note that operand DS1 used in line 13 is not defined. This definition could be achieved by inserting a new line or addition to one of the existing lines.

LINE EDIT

"WHICH LINE?"

14

"OTA 2 /"

Press the "BACK" on Special Function Keys, the pointer will be pointing at;

"OTA 2 / "

Then type:

DS1 OCT 1 /

"OTA 2 :DS1 OCT 1 / "

Since no further changes are required press

EXIT

"FILE EDIT?"

To observe the changes type;

-----  
LISTP 10, 15

PAGE 1

\*\*\*\*\*

```
10      DED *DECREMENT IN BCD
        STA 5/
11      LDA 0
        INS *INCREMENT IN BIN./
12      STA 0
        JMP LOOP
        OUT LDA 0/
13      OTA DS1 *OUTPUT THE BIN. #/
14      OTA 2
        DS1 OCT 1/
15      JMP START *READ ANOTHER NUMBER/
TOTAL LINE NUMBER= 16
```

If no further changes are necessary and you wish to store the program on a cassette file press;

STORE

"STORE THE PROGRAM ON FILE?"

1

After the program is stored on the tape, calculator will print;

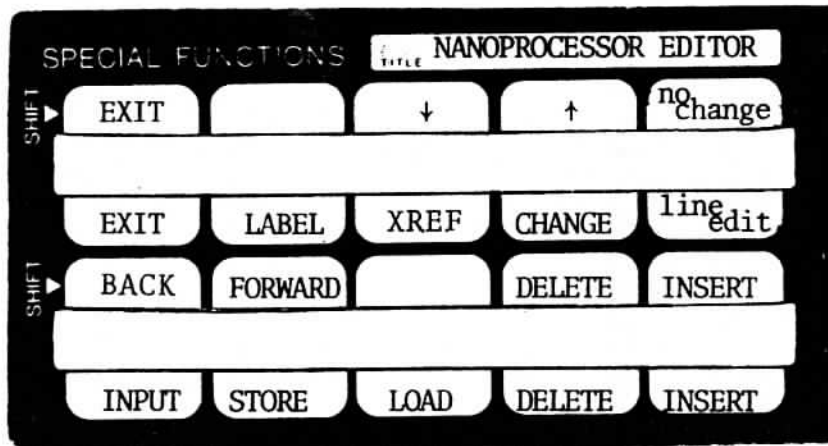
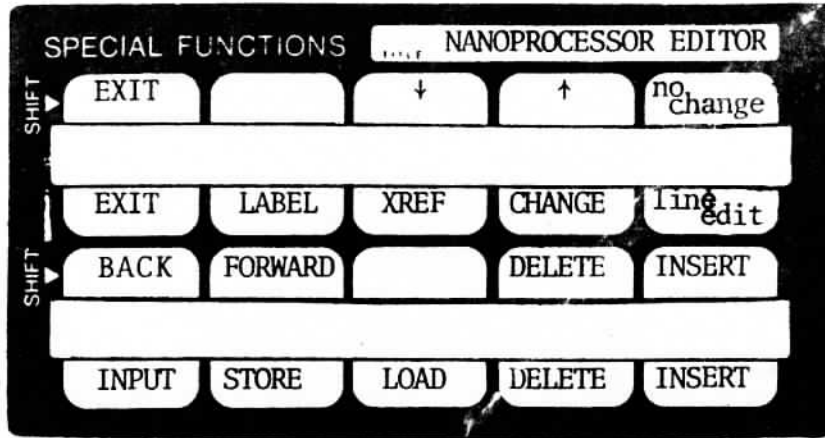
PROGRAM WAS STORED ON FILE 1

"FILE EDIT?"

**Note that if a negative number was entered as the file number, the editor would ignore the STORE command and return to the "FILE EDIT?" mode.**

To terminate program press;

END



USER DEFINABLE KEY OVER LAYER

# NANO PROCESSOR ASSEMBLER

## SECTION I

### USER GUIDE TO THE NANO PROCESSOR ASSEMBLER

PURPOSE: To assemble a source program for the  
NANO PROCESSOR using a 9830 calculator.

MEMORY REQUIREMENTS: 8K

ROM REQUIREMENTS: Advanced programming #11279B.  
String Variables #11274B, and Extended  
I/O #11272B

#### SOFTWARE REQUIREMENTS:

The Nanoprocessor EDITOR must be used to  
generate the source files.

Kamran Firooz  
August, 1974



DESCRIPTION: The Nano Processor Assembler is an absolute assembler designed to assemble source programs (generated by the Nano Processor Editor) stored on cassette tapes and to generate equivalent object code files. A loader program can then be used to load these binary files into a ROM-RAM Simulator, or a PROM. The assembling is performed in two passes. Pass one searches for user defined symbols, and pass two translates the mnemonic source program statements to their equivalent binary codes.

These binary codes are stored in an array called object file. At the end of pass 2 the object file is stored on the cassette tape. The file number where this array will be stored is requested at the beginning of the program.

The assembler program is written as a "conversational" program: that is, the different options of the assembler are asked at the beginning of the program. If the answer "Y" is not encountered the option will be voided. The following is a brief description of these options.

OPTION 1 "SYMBOL TABLE"

For this option the calculator will ask:

"PRINT THE SYMBOL TABLE?"

If the reply is "Y" the symbol table will be printed

OPTION 2 "PROGRAM PRINTOUT"

The second option provides a listing of the source program and its equivalent code. In regard to this option the calculator will ask;

"PRINT THE PROGRAM?"

If the answer is "Y" each line of the assembled program will be printed during pass two.

PERMANENT SYMBOL TABLE:

The permanent symbol table is an array consisting of all the op-codes and their binary equivalents. Permanent symbol table is stored on a file following the assembler file. At the beginning of execution this file is loaded into the calculator.

USER DEFINED TABLE:

User defined table is an array that holds the numerical value or the address of the labels. During the pass 1 all the labels are stored in this array. At the end of pass 1 this array is sorted in alphabetical order. The alphabetical arrangement of the labels make it possible to perform algorithmic search instead of a linear search. During pass 2 everytime an alphabetical operand is found, the assembler performs a logarithmic search into the user defined table to find the value or the address of the operand.

Maximum length of user defined table is 140 labels. Exceeding this limit would cause an error which stops the program.

OBJECT FILE:

Object file is an array that holds the binary codes of the assembled source program. At the end of pass II this file is stored on a cassette Tape. A loader program can then be used to load the object file to a ROM-RAM Simulator, or a ROM.

Object file is a 1024X1 array. Each location of this file will hold the object code for that location. For example; location 16 will hold the code that must be stored on location 1<sup>7</sup> of the ROM. (Due to the fact that array starts from 1 and not 0. All locations are decremented by one by the "LOADER")

Since object file has only 1024 location, caution must be taken not to exceed location 1777 octal. For example; the code that must be stored on location 2150 octal will be stored on location 150 octal. (11'th bit is truncated).

At the beginning of the assembling all of the locations of the object file are initialized to -|. During the assembling -| is over written by other codes, however the locations not used will remain as -|. This feature is used by the loader for "PATCH ASSEMBLING". For further information refer to "NANO PROCESSOR LOADER".

PROGRAM SOURCE FILES:

Program source files are cassette files that contain the source program. These files are generated through the Nano Processor Editor. Up to 10 files can be assembled at one time. If more than one file is used, an EOF statement must designate the termination of each file.

The maximum length of each file is 140 lines, and Each line is 32 character spaces wide.

A "/" is used to designate the end of line,

For example:

```
LOOP LDA REG5 * LOAD ACC from R5 /
```

For more efficient use of the source files, another character called the PSEUDO END OF LINE CHARACTER (":") is used to tell the assembler that the statement has terminated and that more statements follow on the same physical line

For example:

```
CLA: LDA REG5 :BACK STA R16 /
```

This line will occupy only one physical line of the program source file: However, it will be accepted as three individual lines by the assembler. i.e. This one physical line as far as the assembler is concerned is equivalent to the following lines:

```
CLA /  
LDA REG5 /  
BACK STA R16 /
```

GENERAL FORMAT:

Each line of the program consists of one or more separate fields. These fields are: Label, Opcode, Operand, and Comments. For the convenience of the user these fields are separated by one or more blank spaces. The following is a brief description of each one of these fields.

LABEL:

Label is a symbolic name that provides the ability to refer to the instruction or the value generated by the instruction, for example; in the instruction:

```
START LDA REG17 /
```

START is the label, and it holds the address of the location where this instruction is stored on the ROM.

But in the instruction:

```
REG17 OCT 17 /
```

REG17 is a label that holds the numerical value assigned to it by the OCT instruction.

The first letter of a label must be alphabetical, and the total length of the label cannot exceed 6 characters. If the first character of an instruction is blank the assembler assumes that there is no label present. Repeated labels cause the assembler to print an error message.

OPCODE:

Opcodes are mnemonic operation codes stored in the permanent symbol table that are recognized by the assembler and translated as machine instructions or Pseudo-instructions.

MACHINE INSTRUCTIONS:

Machine instructions are those instructions that the Nano Processor can execute to perform a specific task. The assembler translates these instructions to their binary codes.

There are three types of machine instructions:

Type 1:

Single byte instructions that are self-defined and do not require an operand.

For example:

CLA	*	CLEAR ACC
STE	*	Set extend register
RTS	*	Return from Subroutine
ENI	*	Enable the Inrupt
INB	*	Increment the ACC in Binary
SLE	*	Skip if ACC $\leq$ to register 0

Type 2:

Single byte instructions that require an Operand.

For example:

SBS 5	*	Skip if Bit 5 of the ACC is set
CBN BIT4	*	Clear BIT4 of the ACC
INA DS5	*	Input to ACC from Device 5

Type 3:

Double byte instructions that must be accompanied by an Operand -

For example:

OTR 2,DATA	*	Output ROM Data to Device 2
STR R5,FOUR	*	Store FOUR Into Register 5
JMP GOOD	*	Jump to Location GOOD
JSB ADD	*	Jump to Subroutine ADD



PSEUDO INSTRUCTION:

Pseudo instructions performs two types of tasks,

Type 1:

They provide information to the assembler about the program being assembled, such as ORG, EOF, END

Type 2:

They allow the definition of constants, such as OCT, DEC, BCD. Obviously type 2 of the Pseudo Instruction must be accompanied by a label and an Operand, since it is assigning the numerical value of the Operand to the label.

OPERAND:

Some instructions require the designation of an Operand. This Operand could be a destination address in a JUMP instruction or the numerical value of a label in an assign instruction. There are three types of Operands, they are:

Type 1 - NUMERICAL VALUE:

This type of Operand is used in a type two instruction code, or in a Constant Define Pseudo-instruction.

(Type 2 Pseudo instruction).

NOTE: ALL NUMERIC VALUES ARE TAKEN AS OCTAL EXCEPT IN BCD OR DEC. PSEUDO INSTR.

For example:

LDA 5		*	LOAD ACC FROM REGISTER 5
SFZ 4		*	SKIP IF FLAG 4 IS ZERO
REG14	OCT 14	*	ASSIGN VALUE OF 14 TO THE LABEL REG14
JMP	377	*	JUMP TO LOCATION 377
LDR	20	*	LOAD ACC FROM ROM DATA 20

This type of Operand has to be numerical. If they are being used in a type two instruction they cannot exceed 7 or 17 (OCTAL), if they are being used in a constant instruction their octal value should not exceed 377.

The following Operands are acceptable:

CBN 5		*	CLEAR BIT 5 OF ACC
STA 16		*	STORE ACC IN REGISTER 16
AA	OCT 167	*	
BB	DEC 250		
CC	BCD 89		

However the following Operands will cause error messages.

SBN 20		SET BIT 20 OF ACC	(Accumulator has only 8 bits.)
SFS 14		SKIP IF FLAG 14 IS SET	(There are only 8 flags.)
DD	OCT 19		(Unacceptable octal numbers.)
EE	DEC 340		(Exceed 377 octal.)
EE	BCD 140		(Exceed 377 octal.)

Type 2 - SYMBOLIC ADDRESS OR SYMBOLIC VALUE:

This type of Operand is used in jump and jump to subroutine instructions or in a type two opcode instruction.

Example:

```
JMP LOOP
JSB ADDING
JBN BIT4
LDA RIZ
STA R6
JAI INDI
```

This type of Operand follows the same Syntax rules as the label, that is; it must begin with an alphabetical character and must be less than or equal to 6 characters long. These Operands must be defined somewhere in the program as an address or a constant.

Type 3 - SYMBOLIC OR NUMERICAL VALUE

This type of Operand is a mixture of type 1 and type 2 Operands, and it is used in type 3 instructions.

For example:

```
STR R4,FORTY
STR 4,FORTY
STR R4,40
STR 4,40
```

As the above example indicate, this type of Operand consists of two separate fields. Either one of these fields are separated from each other by a ",", and there should be no blank space anywhere in the Operand Field. The symbolic portion of Operand follows the same rules as type one of the Operands.

#### COMMENTS

The comment field allow the user to transcribe comments on the list output produced by the assembler. The comments field must begin with an asterisk. This field could start at the beginning of a line such as:

```
* THIS IS ONLY A COMMENT /
```

```
or after a type one Opcode
```

```
AGAIN CLE * CLEAR EXTEND REGISTER /
```

Comments are ignored during pass one.

If an \* occurs at the beginning of a line, the entire line is assumed to be a comment.

PSEUDO OPCODES:

ORG:                   ORG is a Pseudo Opcode that provides absolute program origin or starting address of a segment of a program. The operand of the ORG must be an octal number. If no ORG is encountered the assembler assumes the starting address to be zero.

EOF:                   An EOF statement notifies the assembler that the physical end of file has reached which causes the assembler to load the next source file.

END:                   Terminates the source language program.

Note that ORG, EOF, and END are not executable statements; therefore any jump or jump subroutine to these instructions would cause an error.

OCT:                   OCT is a defining opcode that equates the numerical value of the operand to the label. Obviously the operand needs to be an octal number.

DEC:                   DEC Pseudo Opcode is another defining statement that converts the numerical value of the operand to octal and equates the converted number to the label.

BCD:

BCD is a pseudo opcode that converts the numerical value of the operand from BCD to octal equivalent. Each digit of the operand is taken as a 4 bit BCD number.

For example in the following statement:

TAG	BCD	38
-----	-----	----

The assembler separates the number 38 to 3 and 8 as 0011 1000.

This number is then converted to octal 00 111 000 (070). Note that the operand cannot exceed two digits.

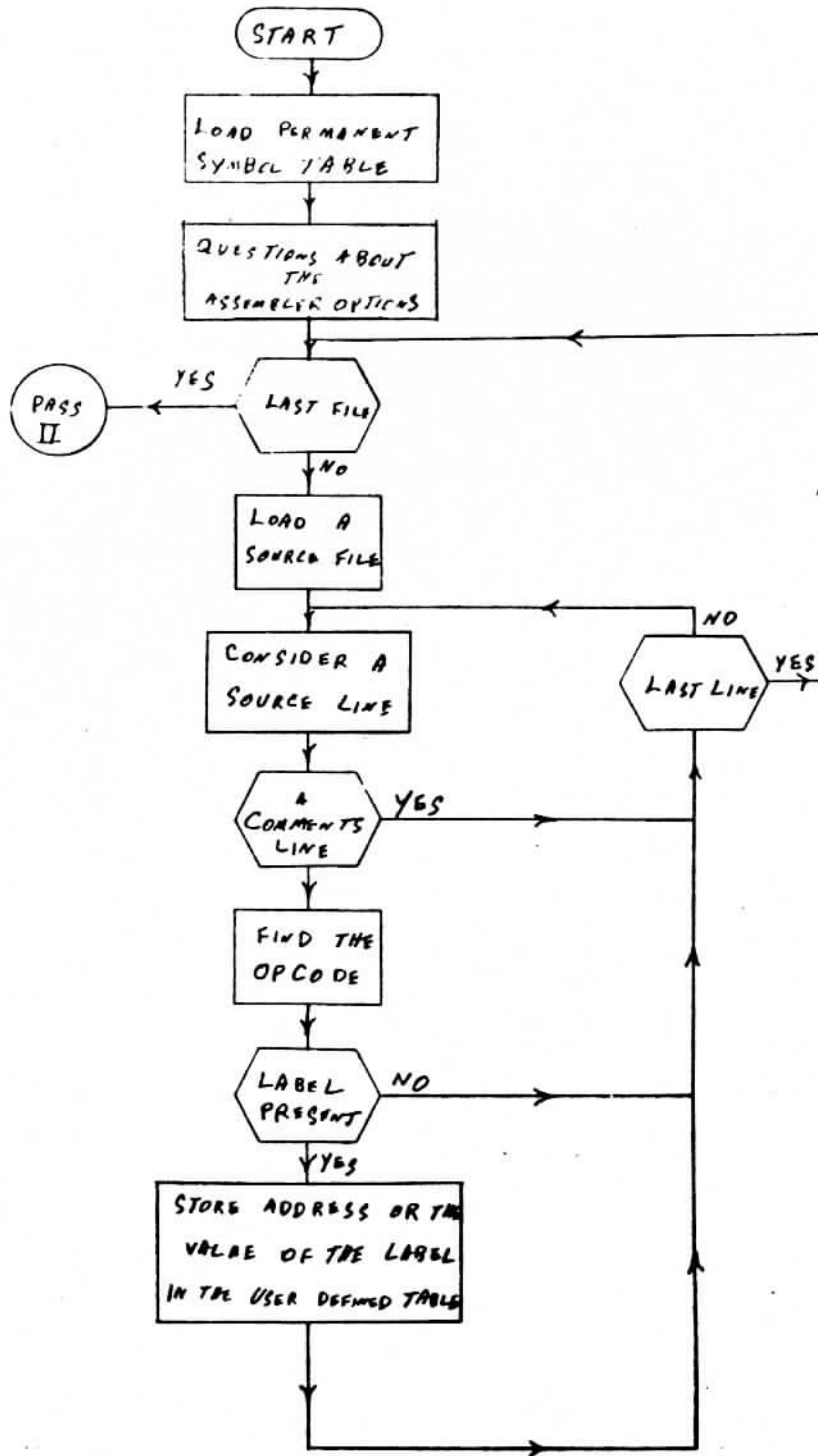
SECTION II

A BRIEF DESCRIPTION OF THE ASSEMBLER PROGRAM  
AND A FLOW CHART FOR BOTH PASSES

THE PROGRAM: NANO PROCESSOR ASSEMBLER program is written in the 9830 BASIC language. The source files are stored in an integer array and converted to string variable by the use of "TRANSFER" statement for assembling. The program consists of two passes, in pass one the assembler searches for labels and checks the syntax of opcodes. Labels or the addresses associated with them are stored in an array called "USER DEFINED TABLE". At the end of the pass one, this file is sorted in alphabetical order. This arrangement makes it possible to perform a logarithmic search for the labels rather than a linear search.

In pass 2 the assembler converts all of the statements to their equivalent binary codes, and stores the converted codes in an array called "OBJECT FILE". At the end of the assembling, the "OBJECT FILE" will be stored on a cassette tape.

The following pages include a simplified flowchart of both passes.



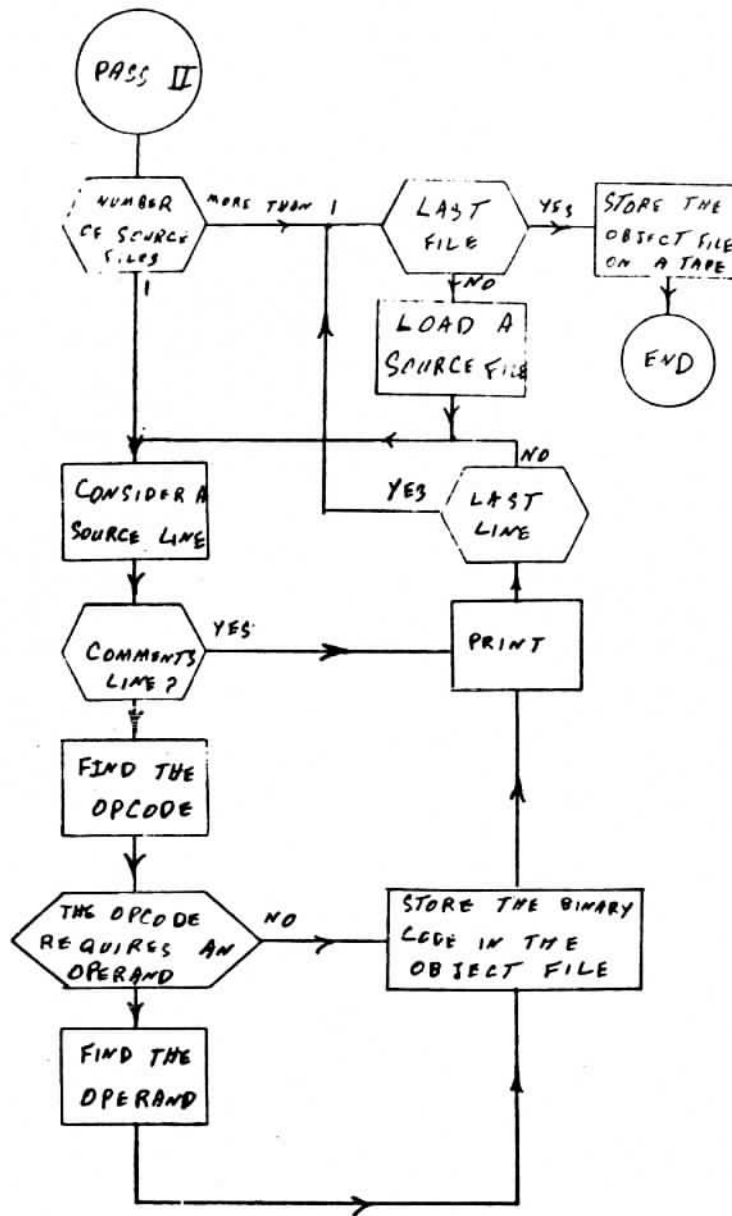
FLOW CHART OF PASS I

NANO PROCESSOR ASSEMBLER

KAMRAN FIROOZ

SEPT. 1974





Flow CHART OF PASS II

NANO PROCESSOR ASSEMBLER

KAMRAN FIROOZ

SEPT 1974

EXAMPLES

The following examples are given in an attempt to familiarize the user with the Nono Processor ASSEMBLER.

EXAMPLE I

The following program will add the contents of Register 5 and Register 6 and store the result on Register 6. Source program was generated by the "NANO PROCESSOR EDITOR", and stored on file 2 of a cassette tape.

PAGE 1

```
*****  
1 * NANO PROCESSOR ASSEMBLER /  
2 * EXPMPLE ONE /  
3 *:::*/  
4 * ADD THE CONTENTS OF REG. 5/  
5 * TO THE CONTENTS OF REG. 6 /  
6 * AND STORE THE RESULT IN REG.6/  
7 *:::*/  
8 LOOP LDA R5 *LOAD ACC FROM R5/  
9 DED *DECREMENT IN DECIMAL/  
10 SAN *SKIP IF ACC #0/  
11 JMP OUT: STA R5/  
12 LDA R6 *LOAD ACC FROM R6/  
13 IND *INCREMENT IN DECIMAL/  
14 STA R6 *STORE ACC AT R6 /  
15 JMP LOOP:OUT LDA R6/  
16 IND: STA R6 *R6 HAS THE RESULT/  
17 R5 OCT 5 *R5 IS OCTAL 5/  
18 R6 OCT 6: END/  
19 EOF/
```

TOTAL LINE NUMBER= 19

Load the assembler into the 9830 and press RUN, EXECUTE.  
After the Permanent Symbol Table is loaded into the Calculator,  
Calculator will display;

PRINT SYMBOL TABLE ?

Y

PRINT THE PROGRAM ?

Y

STORE THE OBJECT FILE ONFILE NO. ?

3

HOWMANY SOURCE FILES ?

1

FILE NO. ?

2

At this point source program stored on file 2 is loaded and the  
following pages are printed on the printer.

\*SYMBOL TABLE\*  
SYMBOL ADDRESS (VALUE)

---

LOOP	0
OUT	13
R5	5
R6	6

NUMBER OF ERRORS FOR PASS 1= 0

```

1 * HAND PROCESSOR ASSEMBLER
2 *   EXAMPLE ONE
3 *
4 * ADD THE CONTENTS OF REG. 5
5 * TO THE CONTENTS OF REG. 6
6 * AND STORE THE RESULT IN REG.6
7 *
8 *
9 *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 *
25 *
26 *
27 *
28 *
29 *
30 *
31 *
32 *
33 *
34 *
35 *
36 *
37 *
38 *
39 *
40 *
41 *
42 *
43 *
44 *
45 *
46 *
47 *
48 *
49 *
50 *
51 *
52 *
53 *
54 *
55 *
56 *
57 *
58 *
59 *
60 *
61 *
62 *
63 *
64 *
65 *
66 *
67 *
68 *
69 *
70 *
71 *
72 *
73 *
74 *
75 *
76 *
77 *
78 *
79 *
80 *
81 *
82 *
83 *
84 *
85 *
86 *
87 *
88 *
89 *
90 *
91 *
92 *
93 *
94 *
95 *
96 *
97 *
98 *
99 *
100 *

```

8	0	145	LOOP	LDA	R5	*LOAD ACC FROM R5
9	1	000		DED		*DECREMENT IN DECIMAL
10	2	017		SAN		*SKIP IF ACC#0
11	3	200		JMP	OUT	
11	4	013				
11	5	165		STA	R5	
12	6	146		LDA	R6	*LOAD ACC FROM R6
13	7	002		IND		*INCREMENT IN DECIMAL
14	10	166		STA	R6	*STORE ACC AT R6
15	11	200		JMP	LOOP	
15	12	000				
15	13	146	OUT	LDA	R6	
16	14	002		IND		
16	15	166		STA	R6	*R6 HAS THE RESULT
17			R5	OCT	5	*R5 IS OCTAL 5
18			R6	OCT	6	
				END		

NUMBER OF ERRORS FOR PASS 2= 0

EXAMPLE II

The following program examines 2 Direct Control lines (DC0 -DC1) and based on their conditions displays a different message on an external display.

The editor listing and the assembler listings are provided in the following pages.

\*\*\*\*\*

```

1  * DISPLAY ROUTINE:****/
2  STR R1,0 *CLEAR REGISTER 1/
3  READ STR R0,0 *CLEAR R0/
4  SFS FLAG0 *SKIP IF DC0 IS SET/
5  JMP DISPA *DISPLAY MESSAGE A/
6  SFS FLAG1 *SKIP IF DC1 IS SET/
7  JMP DISPB *DISPLAY MESSAGE B/
8  ENI *ENABLE THE INTERRUPT/
9  JMP READ *BOTH FLAGS ARE CLEAR/
10 DISPA STR R0,40 *REG0=40/
11 LDA R1 *LOAD ACC FROM POINTER/
12 SLT *SKIP IF ACC < 40/
13 STR 1,0 *CLEAR THE PONTER/
14 ENI : JMP READ /
15 DISPB STR R0,40 * REG0=40 /
16 LDA R1 *LOAD ACC FROM POINTER/
17 SGE *SKIP IF ACC>=40/
18 STR R1,40 *POINTER=40/
19 ENI: JMP READ/
20 ORG 377 *INTERRUPT ROUTINE/
21 STA R2 *STORE THE ACC DURING/
22 * THE INTERRUPT PERIOD:*/
23 LDA R1 *LOAD ACC FROM POINTER/
24 OTR DS0 *OUTPUT ACC TO ADDRESS/
25 * LATCH (DEVICE SELECT 0):***/
26 JAI 2 *JUMP INDIRECT TO LOC. /
27 * 010, AND 8 BITS OF ACC/
28 BL OTR DS2,40 *OUTPUT BLANK /
29 JMP DISP/
30 A OTR DS2,101 *OUTPUT 'A' CODE /
31 JMP DISP/
32 D OTR DS2,104 * OUTPUT 'D' CODE/
33 JMP DISP/
34 E OTR DS2,105 *OUTPUT 'E' CODE/
35 JMP DISP/
36 I OTR DS2,111 *OUTPUT 'I' CODE/
37 JMP DISP/
38 K OTR DS2,113 *OUTPUT K /
39 JMP DISP/
40 L OTR 2,114 *OUTPUT L: JMP DISP/
41 N OTR 2,116 *OUTPUT N: JMP DISP/
42 O OTR 2,117 *OUTPUT O: JMP DISP/
43 P OTR 2,120 *OUTPUT P: JMP DISP/
44 R OTR 2,122 *OUTPUT R: JMP DISP/
45 S OTR 2,123 *OUTPUT S: JMP DISP/
46 Y OTR 2,131 *OUTPUT Y: JMP DISP/
47 DISP OTR DS1,40 *OUTPUT CODE /
48 * FOR BLANK TO DEVICE 1:*/
49 OTR DS3 *START THE DISPLAY/
50 INB/
51 INB *DOUPLE INCREMENT THE /
52 * POINTER:***/
53 STA 1: LDA 2 *RELOAD THE ACC/
54 * BY ITS VALUE BEFORE THE /
55 * INTERRUPT OCCURED:***/
56 MIDD ENI *ENABLE THE INTERRUPT/

```

\*\*\*\*\*

```

57 *BEFORE RETURN:*** /
58 RIT *RETURN FROM INTERRUPT /
59 RESETA STR R1,0 *RESET POINTER /
60 JMP MIDD /
61 RESETB STR R1,40 *RESET POINTER /
62 JMP MIDD /
63 ORG 1000 *DISPLAY:*** /
64 * DISPLAY IS OK:*** /
65 JMP BL *DISPLAY BLANK /
66 JMP D *DISPLAY D /
67 JMP I *DISPLAY I /
68 JMP S *DISPLAY S /
69 JMP P: JMP L: JMP A: JMP Y /
70 JMP BL: JMP I: JMP S: JMP BL /
71 JMP O: JMP K: JMP RESETA /
72 ORG 1040 *DISPLAY:*** /
73 * ERROR IN DISPLAY:*** /
74 JMP E: JMP R: JMP R /
75 JMP O: JMP R: JMP BL: JMP I /
76 JMP N: JMP BL: JMP D: JMP I /
77 JMP S: JMP P: JMP L: JMP A /
78 JMP Y: JMP RESETB /
79 R0 OCT 0:R1 OCT 1:R2 OCT 2 /
80 DS0 OCT 0:DS1 OCT 1 /
81 DS2 OCT 2:DS3 OCT 3 /
82 FLAG0 OCT 0:FLAG1 OCT 1: END /
83 EOF /

```

TOTAL LINE NUMBER= 83



\*SYMBOL TABLE\*

SYMBOL ADDRESS (VALUE)

READ	2
DISPA	15
DISPB	26
BL	403
A	407
D	413
E	417
I	423
F	427
L	433
N	437
O	443
P	447
R	453
S	457
Y	463
DISP	467
MIDD	476
RESETA	500
RESETB	504
R0	0
R1	1
R2	2
DS0	0
DS1	1
DS2	2
DS3	3
FLAG0	0
FLAG1	1

NUMBER OF ERRORS FOR PASS 1= 0

DISPLAY ROUTINE

```

1  000  321      STR  R1,0      +CLEAR REGISTER 1
2  001  000
3  002  320      READ  STR  R0,0      +CLEAR R0
4  003  000
5  004  030      SFS  FLAG0      +SKIP IF DC0 IS SET
6  005  200      JMP  DISPA      +DISPLAY MESSAGE A
7  006  015
8  007  031      SFS  FLAG1      +SKIP IF DC1 IS SET
9  008  250      JMP  DISPB      +DISPLAY MESSAGE B
10 009  026
11 010  057      ENI
12 011  200      JMP  READ      +ENABLE THE INTERRUPT
13 012  002      +BOTH FLAGS ARE CLEAR
14 013  320      DISPA STR  R0,40      +REG0=40
15 014  040
16 015  141      LDA  R1      +LOAD ACC FROM POINTER
17 016  011      SLT
18 017  321      STR  1,0      +SKIP IF ACC < 40
19 018  000      +CLEAR THE POINTER
20 019  057      ENI
21 020  200      JMP  READ
22 021  002
23 022  320      DISPB STR  R0,40      * REG0=40
24 023  040
25 024  141      LDA  R1      +LOAD ACC FROM POINTER
26 025  015      SGE
27 026  321      STR  R1,40      +SKIP IF ACC >=40
28 027  040      +POINTER=40
29 028  057      ENI
30 029  200      JMP  READ
31 030  002
32 031  377      ORG  377      +INTERRUPT ROUTINE
33 032  162      STA  R2      +STORE THE ACC DURING
34 033  + THE INTERRUPT PERIOD
35 034  141      LDA  R1      +LOAD ACC FROM POINTER
36 035  120      OTR  DS0      +OUTPUT ACC TO ADDRESS
37 036  + LATCH (DEVICE SELECT 0)
38 037  222      JAI  2      +JUMP INDIRECT TO LOC.
39 038  302      + 010, AND 8 BITS OF ACC
40 039  040      OTR  DS2,40      +OUTPUT BLANK
41 040  201      JMP  DISP
42 041  067
43 042  302      H      OTR  DS2,101      +OUTPUT 'A' CODE
44 043  101
45 044  261      JMP  DISP
46 045  067
47 046  302      D      OTR  DS2,104      * OUTPUT 'D' CODE
48 047  104
49 048  201      JMP  DISP

```

```

33 416 067
34 417 302 E OTR DS2,105 *OUTPUT 'E' CODE
34 420 105
35 421 201 JMP DISP
35 422 067
36 423 302 I OTR DS2,111 *OUTPUT 'I' CODE
36 424 111
37 425 201 JMP DISP
37 426 067
38 427 302 K OTR DS2,113 *OUTPUT K
38 430 113
39 431 201 JMP DISP
39 432 067
40 433 302 L OTR 2,114 *OUTPUT L
40 434 114
40 435 201 JMP DISP
40 436 067
41 437 302 N OTR 2,116 *OUTPUT N
41 440 116
41 441 201 JMP DISP
41 442 067
42 443 302 O OTR 2,117 *OUTPUT O
42 444 117
42 445 201 JMP DISP
42 446 067
43 447 302 P OTR 2,120 *OUTPUT P
43 450 120
43 451 201 JMP DISP
43 452 067
44 453 302 R OTR 2,122 *OUTPUT R
44 454 122
44 455 201 JMP DISP
44 456 067
45 457 302 S OTR 2,123 *OUTPUT S
45 460 123
45 461 201 JMP DISP
45 462 067
46 463 302 Y OTR 2,131 *OUTPUT Y
46 464 131
46 465 201 JMP DISP
46 466 067
47 467 301 DISP OTR DS1,40 *OUTPUT CODE
47 470 040
48 + FOR BLANK TO DEVICE 1
48 *
49 471 123 OTR DS3 *START THE DISPLAY
50 472 000 INB
51 473 000 INB *DOUBLE INCREMENT THE
52 + POINTER
52 *
52 *
53 474 161 STA 1
53 475 142 LDA 2 *RELOAD THE ACC
54 * BY ITS VALUE BEFORE THE
54 * INTERRUPT OCCURED
55 *
55 *

```

```

56      478      057      MIDD  ENI      *ENABLE THE INTERRUPT
57      *BEFORE RETURN
57      *
57      *
58      477      260      RTI      *RETURN FROM INTERRUPT
59      500      321      RESETA STR R1,0  *RESET POINTER
59      501      000
60      502      201      JMP  MIDD
60      503      076
61      504      321      RESETB STR R1,40 *RESET POINTER
61      505      040
62      506      201      JMP  MIDD
62      507      076
63
63      *
63      *
64      * DISPLAY IS OK
64      *
64      *
65      1000      201      JMP  BL      *DISPLAY BLANK
65      1001      003
66      1002      201      JMP  D      *DISPLAY D
66      1003      013
67      1004      201      JMP  I      *DISPLAY I
67      1005      023
68      1006      201      JMP  S      *DISPLAY S
68      1007      057
69      1008      201      JMP  P
69      1009      047
70      1010      201      JMP  L
70      1011      033
70      1012      201      JMP  A
70      1013      007
70      1014      201      JMP  Y
70      1015      053
70      1016      201      JMP  BL
70      1017      003
70      1018      201      JMP  I
70      1019      023
70      1020      201      JMP  S
70      1021      057
70      1022      201      JMP  BL
70      1023      003
70      1024      201      JMP  O
70      1025      043
70      1026      201      JMP  K
70      1027      027
70      1028      201      JMP  RESETA
70      1029      100
71
71      *
71      *
72      * ERROR IN DISPLAY
72      *
72      *
73      1040      201      JMP  E
73      1041      017

```

```

74 1042 201 JMP R
74 1043 053
74 1044 201 JMP R
74 1045 053
75 1046 201 JMP O
75 1047 043
75 1050 201 JMP R
75 1051 053
75 1052 201 JMP BL
75 1053 003
75 1054 201 JMP I
75 1055 023
76 1056 201 JMP N
76 1057 037
76 1060 201 JMP BL
76 1061 003
76 1062 201 JMP D
76 1063 013
76 1064 201 JMP I
76 1065 023
77 1066 201 JMP S
77 1067 057
77 1070 201 JMP P
77 1071 047
77 1072 201 JMP L
77 1073 033
77 1074 201 JMP A
77 1075 007
78 1076 201 JMP Y
78 1077 063
78 1100 201 JMP RESETB
78 1101 104
79 R0 OCT 0
79 R1 OCT 1
79 R2 OCT 2
80 DS0 OCT 0
80 DS1 OCT 1
81 DS2 OCT 2
81 DS3 OCT 3
82 FLAG0 OCT 0
82 FLAG1 OCT 1
82 END

```

NUMBER OF ERRORS FOR PASS 2= 0

EXAMPLE III

The following example will demonstrate the "PATCH ASSEMBLING" feature of the NANO PROCESSOR ASSEMBLER and LOADER.

Suppose it is desirable to change the message B of the example 2

"ERROR IN DISPLAY"

TO:

"ERROR IN DEVICE"

A short program such as the one following could accomplish the desired change.

\*\*\*\*\*

```
1 * PATCH PROGRAM FOR DISPLAY/
2   ORG 510/
3   C OTR 2,103 *OUTPUT 103 TO 2/
4   JMP 467 *JUMP TO DISP/
5   V OTR 2,126 *OUTPUT 127 TO 2/
6   JMP 467 *JUMP TO DISP/
7   ORG 1060/
8   JMP 403 *JUMP TO BL/
9   JMP 413 *JUMP TO D/
10  JMP 417 *JUMP TO E/
11  JMP V/
12  JMP 423 *JUMP TO I/
13  JMP C/
14  JMP 417 *JUMP TO E/
15  JMP 504 * JUMP TO RESETB/
16  END: EOF/
```

TOTAL LINE NUMBER= 16

\*SYMBOL TABLE\*  
SYMBOL ADDRESS (VALUE)

C	510
V	514

NUMBER OF ERRORS FOR PASS 1= 0



```
1 * PATCH PROGRAM FOR DISPLAY
2
3     510      003      C      ORG  510
4     511      103      OTR  2,103      *OUTPUT 103 TO 2
5     512      001      JMP  467      *JUMP TO DISP
6     513      067      V      OTR  2,126      *OUTPUT 127 TO 2
7     514      002      JMP  467      *JUMP TO DISP
8     515      126
9     516      201
10    517      067
11
12    1060      201      ORG  1060
13    1061      003      JMP  403      *JUMP TO BL
14    1062      201      JMP  413      *JUMP TO D
15    1063      013
16    1064      201      JMP  417      *JUMP TO E
17    1065      017
18    1066      201      JMP  V
19    1067      114
20    1070      201      JMP  423      *JUMP TO I
21    1071      023
22    1072      201      JMP  C
23    1073      110
24    1074      201      JMP  417      *JUMP TO E
25    1075      017
26    1076      201      JMP  504      * JUMP TO RESETB
27    1077      104
28
29    END
```

NUMBER OF ERRORS FOR PASS 2= 0

ASSEMBLY LANGUAGE INSTRUCTIONS

ACCUMULATOR INSTRUCTIONS:

Skip on Bit N=1	SBS	N	00 010 .N .
Skip on Bit N=0	SBZ	N	00 110 .N .
Set Bit N	SBN	N	00 100 .N .
Clear Bit N	CBN	N	10 100 .N .
Increment ACC (Binary)	INB		00 000 000
Increment ACC (Decimal)	IND		00 000 010
Decrement ACC (Binary)	DEB		00 000 001
Decrement ACC (Decimal)	DED		00 000 011
Clear ACC	CLA		00 000 100
Complement ACC	CMA		00 000 101
Left Shift ACC	LSA		00 000 110
Right Shift ACC	RSA		00 000 111
Load ACC with ROM DATA *	LDR	DATA	11 001 111 DATA
Skip on E=1	SES		00 011 111
Skip on E=0	SEZ		00 111 111
Set E	STE		10 110 100
Clear E	CLE		10 110 101

REGISTER AND I/O INSTRUCTIONS:

Load ACC From Register R	LDA	R	01 10. R .
Load ACC Indexed	LDI	Z	11 10. Z .
Store ACC At Register R	STA	R	01 11. R .
Store ACC Indexed	STI	Z	11 11. R .
Input To ACC From DS	INA	DS	01 00. DS .
Output ACC To DS	OTA	DS	01 01. DS .
Store ROM DATA At Register R*	STR	R,DATA	11 01. R . DATA
Output ROM DATA To DS*	OTR	DS,DATA	11 00. DS . DATA

Set Control K	STC K	00 101 K
Clear Control K	<del>CLE</del> K	10 101 K
Skip On Flag J=1	SFS J	00 011 J
Skip On Flag J=0	SFZ J	00 111 J

COMPARATOR INSTRUCTIONS:

Skip On ACC >RO	SGT	00 001 000
Skip On ACC <RO	SLT	00 001 001
Skip on ACC = RO	SEQ	00 001 010
Skip On ACC >=RO	SGE	00 001 101
Skip On ACC <=RO	SLE	00 001 100
Skip On ACC #RO	SNE	00 001 110
Skip On ACC = 0	SAZ	00 001 011
Skip On ACC # 0	SAN	00 001 111

PROGRAM CONTROL INSTRUCTIONS:

Jump To Address	JMP ADDRESS	10 000 PN <sup>+</sup> OFFSET
Jump Indirect To Address	JAI U	10 010 U
Jump Sub T Address *	JSB ADDRESS	10 001 PN <sup>+</sup> OFFSET
Jump Indirect Sub To Address	JAS U	10 011 U
Return From Subroutine	RTS	10 111 000
Return From Interrupt and Enable Interrupt	RTE	10 110 001
Return From Interrupt	RTI	10 110 000
No Operation	NOP	01 011 111
Disable The Interrupt	DSI	10 101 111
Enable The Interrupt	ENI	00 101 111

\* Double Byte Instructions

+ PN = Page no.

Kamran Firooz  
October 21, 1974

## NANO PROCESSOR LOADER

Nano Processor loader is a program that loads the object files produced by the Nano Processor Assembler and stored on cassette tapes into the ROM-RAM Simulator. At the beginning of the execution the calculator questions the file number where the object file is stored. Then it asks;

"PATCH LOADING?"

If the reply is negative all of the unused locations of the object file will be loaded by the code for instruction NOP (137). If patch loading was requested; only the assembled codes will be loaded.

Kamran Firooz  
Sept. 1974

Example:

Consider the program given on the following page. In order to load the object file of this program into ROM-RAM Simulator, load the Loader program into the calculator and RUN EXECUTE.

"FILE NO.?"

6

"PATCH LOADING"

N

Since the answer to the latter question was negative locations 0 to 507, and 520 to 1057 and 1100 to 1777 will be filled by the code for NOP. (137 oct.)

However, if the reply was "Y" the only locations effected in the ROM-RAM would be 510 to 517 and 1060 to 1077. Rest of the memory would remain unchanged.

This feature of the loader can be used to combine (PATCH) object files of different source programs.

```

1  A SIMPLE PROGRAM FOR DISPLAY
2
3      510      302      C      ORG 510
4      511      103      OTR 2+103      *OUTPUT 103 TO 2
5      512      201      BRP 467      *JUMP TO DISP
6      513      067
7      514      302      V      OTR 2+126      *OUTPUT 127 TO 2
8      515      126
9      516      201      JMP 467      *JUMP TO DISP
10     517      067
11
12     ORG 1060
13     1061      201      JMP 403      *JUMP TO BL
14     1062      000
15     1063      201      JMP 413      *JUMP TO D
16     1064      013
17     1065      201      JMP 417      *JUMP TO E
18     1066      017
19     1067      201      JMP V
20     1068      114
21     1069      201      JMP 423      *JUMP TO I
22     1070      023
23     1071      201      JMP C
24     1072      110
25     1073      201      JMP 417      *JUMP TO E
26     1074      017
27     1075      201      JMP 504      * JUMP TO RESETB
28     1076      104
29
30     END

```

NUMBER OF JUMPS FOR PASS 2= 0



H. P. PRIVATE

H. P. PRIVATE

THE FOLLOWING IS A DOCUMENTATION OF THE  
NANO PROCESSOR ASSEMBLER AND LOADER WRITTEN  
FOR H.P. 2100 COMPUTERS. TWO PROGRAMS FOR  
TRANSFERRING SOURCE PROGRAMS FROM 9830 TO  
2100, AND TRANSFERRING OBJECT FILES FROM  
2100 TO 9830 ARE ALSO INCLUDED.

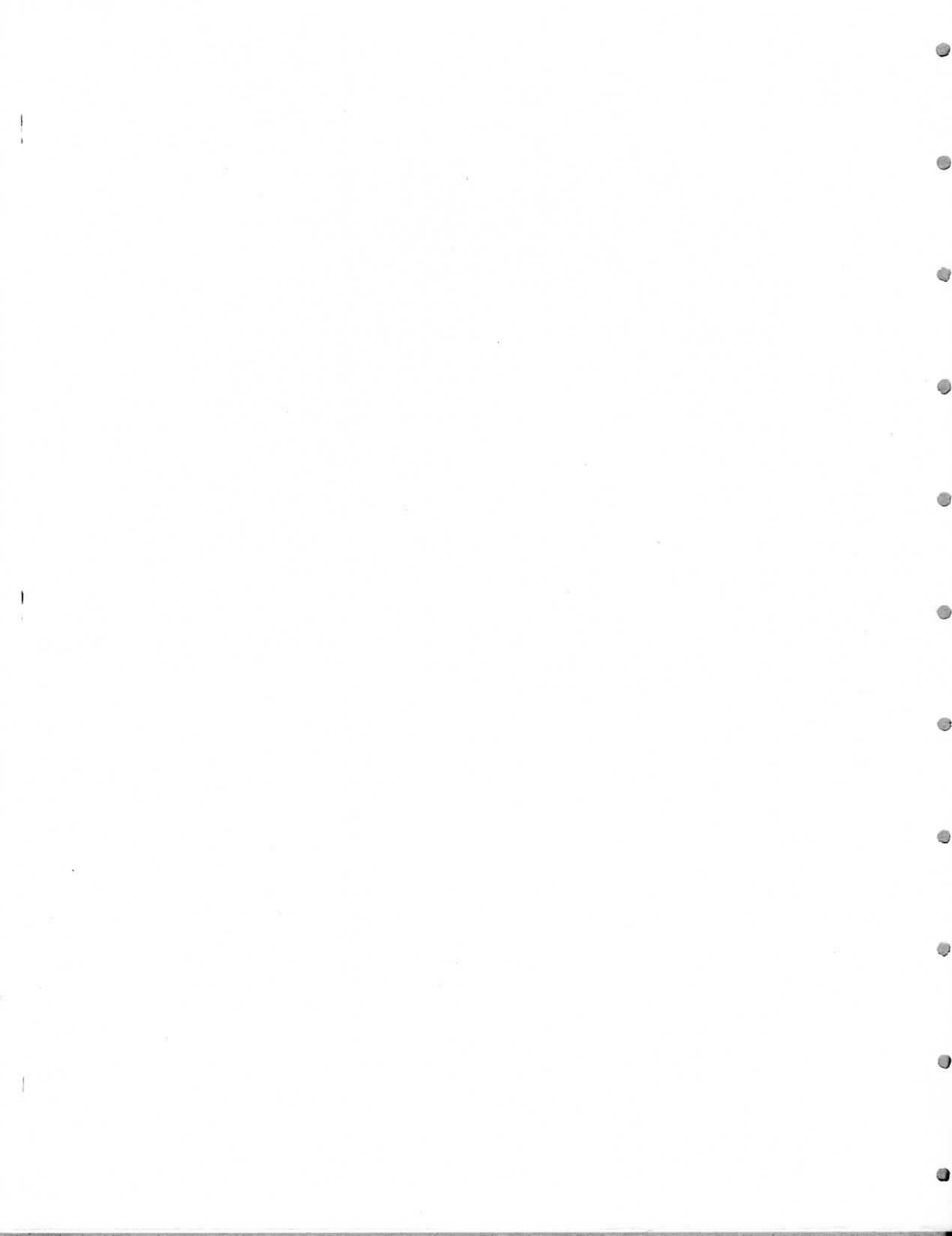
PLEASE BEAR IN MIND THAT NANO PROCESSOR  
MATERIALS ARE H.P. PRIVATE.

FOR FURTHER INFORMATION, RECOMMENDATIONS  
OR IN CASE OF DIFFICULTY PLEASE CONTACT  
KAMRAN FIROOZ AT

303-667-5000 EXT. 2873

DECEMBER/1974





NANO PROCESSOR ASSEMBLER

PURPOSE: To assemble a source program for the  
Nano Processor using an HP 2100 Computer

MEMORY REQUIREMENTS: 16K

SYSTEM REQUIREMENT: DOS III

Kamran Firooz

December, 1974

DESCRIPTION: The Nano Processor Assembler is an absolute assembler designed to assemble source programs stored on a disk and to generate equivalent object code files. A loader program can then be used to load these binary files into a ROM-RAM Simulator, or a PROM. The assembling is performed in two passes. Pass one searches for user defined symbols, and pass two translates the mnemonic source program statements to their equivalent binary codes.

These binary codes are stored in an array called the object file. At the end of pass 2 the object file is stored on the disk. The file name where this array will be stored is requested at the beginning of the program.

ASSEMBLER OPTIONS: The following questions are asked at the beginning of execution:

"LISTING?"

If the response to this option is "YES" the Logical Unit where the listing must be done is requested; otherwise, listing of both passes will be suppressed. In this case all of the assembly and error messages will be listed on Logical Unit 1 (CRT).

Next question is:

"OBJECT FILE NAME?"

Object file must be a binary file of at least 8 sectors. These files can be created prior to the execution of the assembler by using the following command:

```
:ST,B,name,8
```

If the number of sectors in the file is less than 8, the computer will display:

```
"FILE name IS TOO SMALL"
```

```
"OBJECT FILE'S NAME?"
```

Note that the type of file is not searched by the assembler, any type of file other than Binary File will result in an error at the time of storing the object codes into the object file (at the end of Pass 11).

The following questions are asked next:

"HOW MANY SOURCE FILES?"

"ENTER SOURCE FILE'S NAME?"

Up to five source files can be given. The assembler will assemble the source files in the order that file names are entered. Only one file name should be given at a time. If more files are needed the computer will display:

"ENTER SOURCE FILE'S NAME?"

If any of the Source Files are not found on the disk, the computer will display:

"FILE name NOT FOUND"

"ENTER SOURCE FILE'S NAME?"

After all of the Source Files are entered, the assembler starts to assemble the given Source Files.

OBJECT FILE: Object file is an array that holds the binary codes of the assembled source program. At the end of pass 2 this file is stored in a binary file of the disk. A loader program can then be used to load the object file to a ROM-RAM Simulator, or a ROM.

Object file is a 1024 X 1 array. Each location of this file will hold the object code for that location. For example; location 16 will hold the code that must be stored on location 15 of the ROM.

(Due to the fact that the array starts from 1 and not 0, all locations are decremented by one by the "LOADER").

Since object file has only 1024 locations, caution must be taken not to exceed location 1777 octal. For example; the code that must be stored on location 2150 octal will be stored on location 150 octal. (11'th bit is truncated); however the address would appear as 2150 in the assembler listing.

At the beginning of the assembling all of the locations of the object file are initialized to ~~137~~ (code for NOP). During the assembling ~~137~~ is overwritten by other codes; however, the locations not used will remain as ~~137~~. This feature is used by the loader for "PATCH ASSEMBLING". For further information refer to "NANO PROCESSOR LOADER".

PROGRAM SOURCE FILES: As the name implies, program source files are files stored on the disk that contain the source programs. These files can be generated or edited using standard HP 2100 editor or any other available editors (CRTED for example).

Up to five files can be assembled at one time. If more than one file is used, an EOF statement must designate the termination of each file.

USER DEFINED TABLE: User defined table is an array that holds the numerical value or the address of the labels. During pass 1 all the labels are stored in this array. In pass 2, everytime an alphabetical operand is found, the assembler performs a linear search into the user defined table to find the value or the address of the operand.

Maximum length of the user defined table is 256 labels. Exceeding this limit would cause the assembler to print error messages, and any label encountered will be ignored.

During the second pass, if any of the ignored labels are referenced, the assembler will print "Undefined Label" error message.

GENERAL FORMAT: Each line of the program consists of one or more separate fields. These fields are: Label, Opcode, Operand, and Comments. For the convenience of the user these fields are separated by one or more blank spaces. The following is a brief description of each one of these fields.

LABEL: Label is a symbolic name that provides the ability to refer to the instruction or the value generated by the instruction. For example, in the instruction:

```
START LDA REG17
```

START is the label, and it holds the address of the location where this instruction is stored on the ROM.

But in the instruction:

```
REG17 OCT 17
```

REG17 is a label that holds the numerical value assigned to it by the OCT instruction.

The first letter of a label must be alphabetical, and the total length of the label cannot exceed 5 characters. If the first character of an instruction is blank the assembler assumes that there is no label present. Repeated labels cause the assembler to print an error message.



OPCODE: Opcodes are mnemonic operation codes stored in the permanent symbol table that are recognized by the assembler and translated as machine instructions or Pseudo-instructions.

MACHINE INSTRUCTIONS: Machine instructions are those instructions that the Nano Processor can execute to perform a specific task. The assembler translates these instructions to their binary codes. There are three types of machine instructions:

Type 1:

Single byte instructions that are self-defined and do not require an operand.

For example:

CLA	*	CLEAR ACC
STE	*	Set extend register
RTS	*	Return from Subroutine
ENI	*	Enable the intrupt
INB	*	Increment the ACC in Binary
SLE	*	Skip if ACC $\leq$ to register 0

Type 2:

Single byte instructions that require an Operand.

For example:

SBS 5	*	Skip if Bit 5 of the ACC is set
CBN BIT4	*	Clear Bit4 of the ACC
INA DS5	*	Input to ACC from Device 5

Type 3:

Double byte instructions that must be accompanied by an Operand -

For example:

OTR 2,DATA	*	Output ROM Data to Device 2
STR R5,FOUR	*	Store FOUR Into Register 5
JMP GOOD	*	Jump to Location GOOD
JSB ADD	*	Jump to Subroutine ADD

PSEUDO INSTRUCTION: Pseudo instructions perform two types of tasks:

Type 1:

They provide information to the assembler about the program being assembled, such as  
ORG, EOF, END

Type 2:

They allow the definition of constants, such as OCT, DEC, BCD. Obviously, type 2 of the Pseudo Instruction must be accompanied by a label and an Operand, since it is assigning the numerical value of the Operand to the label.

OPERAND: Some instructions require the designation of an Operand. This Operand could be a destination address in a JMP instruction or the numerical value of a Label in an assign instruction. There are three types of Operands:

Type 1 - NUMERICAL VALUE:

This type of Operand is used in a type 2 instruction code, or in a Constant Define Pseudo instruction.

(Type 2 Pseudo instruction)

*Note: all numeric values are taken as OCTAL except in BCD or DEC pseudo-instr.*

For example:

LDA 5		* LOAD ACC FROM REGISTER 5
SFZ 4		* SKIP IF FLAG 4 IS ZERO
REG14	OCT 14	* ASSIGN VALUE OF 14 TO
		* THE LABEL REG14
JMP 377		* JUMP TO LOCATION 377
LDR 20		* LOAD ACC FROM ROM DATA 20

This type of Operand has to be numerical. If they are being used in a type 2 instruction they cannot exceed 7 or 17 (OCTAL); if they are being used in a define constant instruction their octal value should not exceed 377.

The following Operands are acceptable:

CBN 5		* CLEAR BIT 5 OF ACC
STA 16		* STORE ACC IN REGISTER 16
AA	OCT 167	* OCTAL 167
BB	DEC 250	* OCTAL 372
CC	BCD 89	* OCTAL 251

However the following Operands will cause error messages:

SBN 20	SET BIT 20 OF ACC
	(Accumulator has only 8 bits.)

SFS	14		SKIP IF FLAG 14 IS SET
			(There are only 8 flags.)
DD	OCT	19	(Unacceptable octal numbers.)
EE	DEC	340	(Exceed 377 octal.)
FF	BCD	140	(Exceed 377 octal.)

Type 2. SYMBOLIC ADDRESS OR SYMBOLIC VALUE:

This type of Operand is used in jump to subroutine instructions or in a type 2 opcode instruction.

For example:

```
JMP LOOP
JSB ADDNG
JBN BIT4
LDA RIZ
STA R6
JAI INDI
```

This type of Operand follows the same Syntax rules as the Label; that is, it must begin with an alphabetical character and must be less than or equal to 5 characters long. These Operands must be defined somewhere in the program as addresses or constants.

Type 3 - SYMBOLIC OR NUMERICAL VALUE:

This type of Operand is a mixture of type 1 and type 2 Operands, and it is used in type 3 instructions.

For example:

```
STR R4,FORTY
STR 4,FORTY
STR R4,4Ø
STR 4,4Ø
```

As the above examples indicate, this type of Operand consists of two separate fields. These fields are separated from each other by a ",", and there should be no blank space anywhere in the Operand Field. The symbolic portion of Operand follows the same rules as type 1 of the Operands.

COMMENTS: The comment field allows the user to transcribe comments on the list output produced by the assembler. The comments field must begin with an asterisk. This field could start at the beginning of a line, such as:

\* THIS IS ONLY A COMMENT

or after the Opcode or Operand

AGAIN CLE \* CLEAR EXTEND REGISTER

Comments are ignored during pass one.

If an "\*" occurs at the beginning of a line, the entire line is assumed to be a comment.

If a comment starts at the beginning of a line, up to 64 characters can be used in each line. If a comment begins after an Opcode or Operand, up to 28 characters will be printed and remainder will be truncated.

ERROR MESSAGES: For the convenience of the user, the assembler will print error messages if any error are encountered. Along with the message, the line number where the error occurred is printed.

PSEUDO OPCODES:

- ORG:      ORG is a Pseudo Opcode that provides absolute program origin or starting address of a segment of a program. The operand of the ORG must be an octal number. If no ORG is encountered the assembler assumes the starting address to be zero.
- EOF:      An EOF statement notifies the assembler that the physical end of file has been reached. This causes the assembler to load the next source file.
- END:      End terminates the source language program. Note that ORG, EOF, and END are not executable statements; therefore, any reference to these instructions would cause an error.
- OCT:      OCT is a defining opcode that equates the numerical value of the operand to the label. Obviously, the operand needs to be an octal number.
- DEC:      DEC Pseudo Opcode is another defining



statement that converts the numerical value of the operand to octal and equates the converted number to the label.

BCD: BCD is a pseudo opcode that converts the numerical value of the operand from BCD to its octal equivalent. Each digit of the operand is taken as a 4 bit BCD number.

For example, in the following statement:

TAG	BCD	38
-----	-----	----

The assembler separates the number 38 to 3 and 8 as 0011 1000.

This number is then converted to octal 00 111 000 (070). Note that the operand cannot exceed two digits.

EXAMPLES

The following examples are given in an attempt to familiarize the user with the NANO PROCESSOR ASSEMBLER.

EXAMPLE I

The following program will add the contents of Register 5 and Register 6 and store the result on Register 6. The source program was generated by the "CRTED EDITOR" and stored on File NPEX1 of a disk.

PAGE 0001

```
0001 *      NANO PROCESSOR ASSEMBLER
0002 *      EXAMPLE I
0003 *
0004 *
0005 *
0006 *      THIS PROGRAM ADDS THE CONTENTS OF REGISTER 5 TO THE
0007 *      CONTENTS OF REGISTER 6 AND STORE THE RESULT IN REGISTER 6.
0008 *
0009 *
0010 *
0011 LOOP LDA R5 *LOAD ACC FROM REG. 5
0012 DED *DECREMENT IN DECIMAL
0013 SAN *SKIP IF ACC #0
0014 JMP OUT *(END OF ROUTINE)
0015 STA R5 *STORE THE ACC IN R5
0016 LDA R6 *LOAD ACC FROM REG. 6
0017 IND *INCREMENT IN DECIMAL
0018 STA R6 *STORE ACC AT REGISTER 6
0019 JMP LOOP
0020 *      REPEAT THE DECREMENT AND INCREMENT ROUTINE
0021 OUT LDA R6
0022 IND
0023 STA R6 *R6 HAS THE SUM
0024 R5 0015 *DEFINE R5 AS OCTAL 5
0025 R6 0016
0026 END
0027 EOF
```

Load the assembler into the 2100 as follow;

:PR,NPA

After the program is loaded the computer will display;

YES

WANT FOR OUTPUT ?

6

FILE NAME ?

OBJ1

SOURCE FILES ?

1

SOURCE FILE'S NAME

NPEX1

At this point the source program stored on File NPEX1 is loaded and the following pages are printed on the printer (Logical Unit 6).

SYMBOL TABLE\*

SYMBOL ADDRESS OF VALUE

LOAD	0000
OUT	0013
IN	0001
RR	0005

NO OF ERRORS FOR PASS 1 = 0

```

1 *      HAND PROCESSOR ASSEMBLER
2 *      ROUTINE 1
3 *
4 *
5 *
6 *      THIS PROGRAM ADDS THE CONTENTS OF REGISTER 5 TO THE
7 *      CONTENTS OF REGISTER 6 AND STORES THE RESULT IN REGISTER 6.
8 *
9 *
10 *
11 0000 145  LOOP  LDA  R5          *LOAD ACC FROM REG. 5
12 0001 420      DEB              *DECREMENT IN DECIMAL
13 0002 017      SNA              *SKIP IF ACC #0
14 0003 200      JMP  OUT          *(END OF ROUTINE)
14 0004 013
15 0005 165      STA  R5          *STORE THE ACC IN R5
16 0006 146      LDA  R6          *LOAD ACC FROM REG. 6
17 0007 002      INC              *INCREMENT IN DECIMAL
18 0010 166      STA  R6          *STORE ACC AT REGISTER 6
19 0011 200      JMP  LOOP
19 0012 000
20 *      REPEAT THE DECREMENT AND INCREMENT ROUTINE
21 0013 146  OUT  LDA  R6
22 0014 002      INC
23 0015 166      STA  R6          *R6 HAS THE SUM
24      R5      OCT  5          *DEFINE R5 AS OCTAL 5
25      R6      OCT  6
26      END

```

NO OF ERRORS FOR PAGE 2 = 0

EXAMPLE II

The following program examines 2 Direct Control lines (DC0 - DC1) and based on their conditions displays a different message on an external display.

The editor listing and the assembler listings are provided on the following pages.

```
0001 *      NAND PROCESSOR ASSEMBLER
0002 *      EXAMPLE 2
0003 *
0004 *
0005 *      DISPLAY ROUTINE
0006 *
0007 *
0008 *      STR R1,0 *CLEAR REGISTER 1
0009 *      READ STR R0,0 *CLEAR R0
0010 *      SFC FLAG0 *SKIP IF DC0 IS SET
0011 *      JMP DISPA *DISPLAY MESSAGE A
0012 *      SFC FLAG1 *SKIP IF DC1 IS SET
0013 *      JMP DISPB *DISPLAY MESSAGE B
0014 *      FHI *ENABLE THE INTERRUPT
0015 *      JMP READ *NEITHER FLAG IS SET
0016 *      DISPA STR R0,40 *STORE 40 IN R0
0017 *      LDA R1 *LOAD ACC FROM R1
0018 *      SLT *SKIP IF ACC <40
0019 *      STR 1,0 *CLEAR THE POINTER
0020 *      ENI
0021 *      JMP READ
0022 *      DISPB STR R0,40
0023 *      LDA R1
0024 *      SGE *SKIP IF ACC > POINTER
0025 *      STR R1,40 *SET THE POINTER FOR B
0026 *      ENI
0027 *      JMP READ
0028 *      OTR DS7 *INTERRUPT ROUTINE
0029 *      STR R2
0030 *      *K. HOLDS THE CONTENTS OF ACC DURING INTERRUPT ROUTINE
0031 *      LDA R1 *LOAD ACC FROM POINTER
0032 *      OTR DS0
0033 *      *OUTPUT ACC TO ADDRESS LATCH (DEVICE SELECT 0)
0034 *      JAI 2
0035 *      BL OTR DS2,40 *OUTPUT A BLANK
0036 *      JMP DISP
0037 *      A OTR DS2,101 *OUTPUT "A" CODE
0038 *      JMP DISP
0039 *      D OTR DS2,104 *OUTPUT "D" CODE
0040 *      JMP DISP
0041 *      E OTR DS2,105 *OUTPUT "E" CODE
0042 *      JMP DISP
0043 *      I OTR DS2,111 *OUTPUT "I" CODE
0044 *      JMP DISP
0045 *      K OTR DS2,114 *OUTPUT "K" CODE
0046 *      JMP DISP
0047 *      L OTR DS2,115 *OUTPUT "L" CODE
0048 *      JMP DISP
0049 *      N OTR DS2,116 *OUTPUT "N" CODE
0050 *      JMP DISP
0051 *      O OTR 2,117 *OUTPUT "O" CODE
0052 *      JMP DISP
0053 *      P OTR 2,120 *OUTPUT "P" CODE
0054 *      JMP DISP
0055 *      R OTR 2,122 *OUTPUT "R" CODE
0056 *      JMP DISP
0057 *      S OTR DS2,123 *OUTPUT "S" CODE
0058 *      JMP DISP
0059 *      Y OTR 2,131 *OUTPUT "Y" CODE
0060 *      DISP OTR DS1,40 *OUTPUT A BLANK
```

```
0061  OIA D53 *START THE DISPLAY
0062  LMB
0063  INB *DOUBLE INCREMENT THE POINTER
0064  *
0065  *
0066  STA I *UPDATE THE POINTER
0067  LDA Z
0068  *LOAD HL WITH ITS VALUE BEFORE THE INTERRUPT OCCURED
0069  HLD CHI *ENABLE THE INTERRUPT
0070  RTI *RETURN FROM INTERRUPT
0071  SETA CIR R1,0 *RESET POINTER FOR A
0072  JMP MID0
0073  SETB CIR R1,40 *RESET POINTER FOR B
0074  JMP MID0
0075  ORG 1000 *MESSAGE A
0076  JMP BL *DISPLAY A BLANK
0077  JMP D *DISPLAY D
0078  JMP I *DISPLAY I
0079  JMP S *DISPLAY S
0080  JMP P *DISPLAY P
0081  JMP L *DISPLAY L
0082  JMP A *DISPLAY A
0083  JMP Y *DISPLAY Y
0084  JMP BL
0085  JMP J
0086  JMP S
0087  JMP BL
0088  JMP D
0089  JMP K
0090  JMP SETA *RESET THE POINTER
0091  ORG 1040 *MESSAGE B
0092  JMP E
0093  JMP P
0094  JMP R
0095  JMP D
0096  JMP R
0097  JMP BL
0098  JMP I
0099  JMP H
0100  JMP BL
0101  JMP D
0102  JMP I
0103  JMP S
0104  JMP P
0105  JMP L
0106  JMP A
0107  JMP Y
0108  JMP SETB
0109  RA OCT 0
0110  R1 OCT 1
0111  R2 OCT 2
0112  D50 OCT 0
0113  D51 OCT 1
0114  D52 OCT 2
0115  D53 OCT 3
0116  FLAG0 OCT 0
0117  FLAG1 OCT 1
0118  END
```



SYMBOL TABLE

SYMBOL ADDRESS OR VALUE

FLAG	0001
DISP0	0015
DISP0	0015
LL	0400
H	0400
B	0410
E	0417
I	0420
K	0427
L	0430
H	0437
A	0440
P	0447
K	0450
S	0457
T	0460
DISP	0465
MIDB	0474
SETA	0476
SETB	0502
R0	0000
R1	0001
R2	0002
DS0	0000
DS1	0001
DS2	0002
DS3	0003
FLAG0	0000
FLAG1	0001

NO OF ERRORS FOR PASS 1 = 0

```

1 *      MINI PROCESSOR ASSEMBLER
2 *      EXAMPLE 2
3 *
4 *
5 *      DISPLAY ROUTINE
6 *
7 *
8 0000 021      STR  R1,0      *CLEAR REGISTER 1
9 0001 000
9 0002 020  READ  STR  R0,0      *CLEAR R0
9 0003 000
10 0004 030      SFS  FLAG0      *SKIP IF DC0 IS SET
11 0005 200      JMP  DISPA      *DISPLAY MESSAGE A
11 0006 015
12 0007 031      SFS  FLAG1      *SKIP IF DC1 IS SET
13 0010 200      JMP  DISPB      *DISPLAY MESSAGE B
13 0011 026
14 0012 057      ENI
15 0013 200      JMP  READ      *NEITHER FLAG IS SET
15 0014 002
16 0015 020  DISPA STR  R0,40
16 0016 040
17 0017 141      LDA  R1      *LOAD ACC FROM R1
18 0020 011      SLT
19 0021 021      STR  1,0      *SKIP IF ACC <40
19 0022 000      *CLEAR THE POINTER
20 0023 057      ENI
21 0024 200      JMP  READ
21 0025 002
22 0026 020  DISPB STR  R0,40
22 0027 040
23 0030 141      LDA  R1
24 0031 015      SGE      *SKIP IF ACC > POINTER
25 0032 021      STR  R1,40
25 0033 040
26 0034 057      ENI
27 0035 200      JMP  READ
27 0036 002
28          ORG  377      *INTERRUPT ROUTINE
29 0377 162      STA  R2
30 *R2 HOLDS THE CONTENTS OF ACC DURING INTERRUPT ROUTINE
31 0400 141      LDA  R1      *LOAD ACC FROM POINTER
32 0401 120      OTA  DS0
33 *OUTPUT ACC TO ADDRESS LATCH (DEVICE SELECT 0)
34 0402 222      JAI  2
35 0403 002  BL   OTR  DS2,40
35 0404 040
36 0405 201      JMP  DISP
36 0406 065
37 0407 002  A    OTR  DS2,101      *OUTPUT "A" CODE
37 0410 101
38 0411 201      JMP  DISP
38 0412 065
39 0413 002  D    OTR  DS2,104      *OUTPUT "D" CODE
39 0414 104
40 0415 201      JMP  DISP
40 0416 065

```

41	0417	001	E	OTR	DS2,105	*OUTPUT "E" CODE
41	0420	105				
42	0421	001		IMP	DISP	
42	0422	065				
43	0423	001	I	OTR	DS2,111	*OUTPUT "I" CODE
43	0424	111				
44	0425	101		JMP	DISP	
44	0426	065				
45	0427	002	K	OTR	DS2,114	*OUTPUT "K" CODE
45	0430	114				
46	0431	001		JMP	DISP	
46	0432	065				
47	0433	001	L	OTR	DS2,115	*OUTPUT "L" CODE
47	0434	115				
48	0435	001		JMP	DISP	
48	0436	065				
49	0437	002	H	OTR	DS2,116	*OUTPUT "H" CODE
49	0440	116				
50	0441	001		JMP	DISP	
50	0442	065				
51	0443	002	O	OTR	2,117	*OUTPUT "O" CODE
51	0444	117				
52	0445	001		JMP	DISP	
52	0446	065				
53	0447	002	P	OTR	2,120	*OUTPUT "P" CODE
53	0450	120				
54	0451	001		JMP	DISP	
54	0452	065				
55	0453	002	R	OTR	2,122	*OUTPUT "R" CODE
55	0454	122				
56	0455	001		JMP	DISP	
56	0456	065				
57	0457	002	S	OTR	DS2,123	*OUTPUT "S" CODE
57	0460	123				
58	0461	001		JMP	DISP	
58	0462	065				
59	0463	002	Y	OTR	2,131	*OUTPUT "Y" CODE
59	0464	131				
60	0465	001	DISP	OTR	DS1,40	
60	0466	040				
61	0467	123		OTR	DS3	*START THE DISPLAY
62	0470	000		INH		
63	0471	000		INH		*DOUBLE INCREMENT THE POINTER
64						
65						
66	0472	101		STA	J	*UPDATE THE POINTER
67	0473	142		LDA	J	
68	0474	000		AND	DISP	*UPDATE THE POINTER WITH ITS VALUE BEFORE THE INTERRUPT OCCURED
69	0475	007	MIDD	ENI		*ENABLE THE INTERRUPT
70	0476	000		RTI		*RETURN FROM INTERRUPT
71	0476	001	SETR	STR	R1,0	*RESET POINTER FOR R
71	0477	000				
72	0500	001		JMP	MIDD	
72	0501	074				
73	0502	001	SETD	STR	R1,40	
73	0503	040				
74	0504	001		JMP	MIDD	

74	0505	011			
75			OP 1000		*MESSAGE A
76	1006	201	JMP D1		*DISPLAY A BLANK
76	1001	000			
77	1007	201	JMP D		*DISPLAY D
77	1003	013			
78	1004	201	JMP I		*DISPLAY I
78	1001	023			
79	1006	201	JMP S		*DISPLAY S
79	1007	057			
80	1010	201	JMP P		*DISPLAY P
80	1011	047			
81	1012	201	JMP L		*DISPLAY L
81	1013	033			
82	1014	201	JMP R		*DISPLAY R
82	1014	007			
83	1015	201	JMP Y		*DISPLAY Y
83	1011	063			
84	1020	201	JMP BL		
84	1021	003			
85	1022	201	JMP I		
85	1023	023			
86	1024	201	JMP S		
86	1025	057			
87	1026	201	JMP BL		
87	1027	002			
88	1028	201	JMP G		
88	1031	043			
89	1032	201	JMP K		
89	1033	027			
90	1034	201	JMP SETA		*RESET THE POINTER
90	1035	070			
91			ORG 1040		*MESSAGE B
92	1040	201	JMP E		
92	1041	017			
93	1042	201	JMP R		
93	1043	053			
94	1044	201	JMP R		
94	1045	053			
95	1046	201	JMP O		
95	1047	043			
96	1050	201	JMP R		
96	1051	053			
97	1052	201	JMP BL		
97	1053	003			
98	1054	201	JMP J		
98	1055	023			
99	1056	201	JMP N		
99	1057	027			
100	1060	201	JMP BL		
100	1061	003			
101	1062	201	JMP D		
101	1063	013			
102	1064	201	JMP I		
102	1065	023			
103	1066	201	JMP S		
103	1067	057			

```
104 1070 201      JMP  P
104 1071 007
105 1072 201      JMP  L
105 1073 003
106 1074 200      JMP  A
106 1075 007
107 1076 201      JMP  Y
107 1077 003
108 1100 201      JMP  SETB
108 1101 102
109          R0    OCT  0
110          R1    OCT  1
111          R2    OCT  2
112          DS0   OCT  0
113          DS1   OCT  1
114          DS2   OCT  2
115          DS3   OCT  3
116          FLAG0 OCT  0
117          FLAG1 OCT  1
118          END
```

NO OF ERRORS FOR PASS 2 = 0

EXAMPLE III

The following example will demonstrate the "PATCH ASSEMBLING" feature of the NANO PROCESSOR ASSEMBLER and LOADER.

Suppose it is desirable to change the message B of the example 2

"ERROR IN DISPLAY"

TO:

"ERROR IN DEVICE"

A short program such as the one following could accomplish the desired change.

```

1 *      MAIN PROCESSOR ASSEMBLER
2 *      EXAMPLE 5
3 *
4 *
5 *      PART B PROGRAM FOR DISPLAY PROGRAM GIVEN ON EXAMPLE 2
6 *
7 *
8          ORG 510
9 0510 302 C      OTR 2*103      *OUTPUT CODE FOR "C" TO DS2
9 0511 103
10 0512 201      JMP 467      *JUMP TO DISP
10 0513 067
11 0514 302 V      OTR 2*126      *OUTPUT CODE FOR "V"
11 0515 126
12 0516 201      JMP 467
12 0517 067
13          ORG 1060
14 1060 201      JMP 403      *JMP TO BL
14 1061 003
15 1062 201      JMP 413      *JMP TO D
15 1063 013
16 1064 201      JMP 417      *JMP TO E
16 1065 017
17 1066 201      JMP V
17 1067 114
18 1070 201      JMP 423      *JMP TO I
18 1071 023
19 1072 201      JMP C
19 1073 110
20 1074 201      JMP 417      *JMP TO E
20 1075 017
21 1076 201      JMP 504      *JMP TO SETB
21 1077 104
22          END

```

NO OF ERRORS FOR PASS 2 = 0

ASSEMBLY LANGUAGE INSTRUCTIONS

ACCUMULATOR INSTRUCTIONS:

Skip on Bit N=1	SBS	N	00 010 N
Skip on Bit N=0	SBZ	N	00 110 N
Set Bit N	SBN	N	00 100 N
Clear Bit N	CBN	N	10 100 N
Increment ACC (Binary)	INB		00 000 000
Increment ACC (Decimal)	IND		00 000 010
Decrement ACC (Binary)	DEB		00 000 001
Decrement ACC (Decimal)	DED		00 000 011
Clear ACC	CLA		00 000 100
Complement ACC	CMA		00 000 101
Left Shift ACC	LSA		00 000 110
Right Shift ACC	RSA		00 000 111
Load ACC with ROM DATA *	LDR	DATA	11 001 111 DATA
Skip on E=1	SES		00 011 111
Skip on E=0	SEZ		00 111 111
Set E	STE		10 110 100
Clear E	CLE		10 110 101

REGISTER AND I/O INSTRUCTIONS:

Load ACC From Register R	LDA	R	01 10 R
Load ACC Indexed	LDI	Z	11 10 Z
Store ACC At Register R	STA	R	01 11 R
Store ACC Indexed	STI	Z	11 11 Z
Input To ACC From DS	INA	DS	01 00 DS
Output ACC To DS	OTA	DS	01 01 DS
Store ROM DATA At Register R*	STR	R, DATA	11 01 R DATA
Output ROM DATA To DS*	OTR	DS, DATA	11 00 DS DATA



Set Control K	STC	K	PP
Clear Control K	CLC	K	14
Skip On Flag J=1	SFS	J	PP
Skip On Flag J=0	SFZ	J	PP

COMPARATOR INSTRUCTIONS:

Skip On ACC >R0	SGT		PP
Skip On ACC <R0	SLT		PP
Skip on ACC = R0	SEQ		PP
Skip On ACC >R0	SGE		PP
Skip On ACC <R0	SLE		PP
Skip On ACC #R0	SNE		PP
Skip On ACC = 0	SAZ		PP
Skip On ACC # 0	SAN		PP

PROGRAM CONTROL INSTRUCTIONS:

Jump To Address	JMP	ADDRESS	14
Jump Indirect To Address	JAI	U	14
Jump Sub To Address*	JSB	ADDRESS	14
Jump Indirect Sub To Address	JAS	U	14
Return From Subroutine	RTS		14
Return From Interrupt and Enable Interrupt	RTE		14
Return From Interrupt	RTI		14
No Operation	NOP		14
Dissable The Interrupt	DSI		14
Enable The Interrupt	ENI		14

\* Double Byte Instructions

+ PN = Page no.

Kamran Firooz  
October 21, 1987

NANO PROCESSOR LOADER (NPL)  
(Using H.P. 2100 Computer)

Nano Processor Loader is a program that loads the object codes produced by the Nano Processor Assembler and stored on a Binary File of a disk into a ROM-RAM Simulator. At the beginning of the execution the computer questions the I/O slot where the ASCII Card is placed, and then the file name where the object file is stored is asked:

Next question is;

"PATCH LOADING?"

If the reply is negative, all of the unused locations of the object file will be loaded by the code for instruction NOP (137 octal). If patch loading was requested; only the assembled codes will be loaded and all unused locations will remain unchanged.

Kamran Firooz  
December/1974

Example:

Consider the program given on the following page. In order to load the object file of this program into ROM-RAM Simulator, load the Loader program into the computer and RUN EXECUTE.

"OBJECT FILES'S NAME"

OBJ1

"PATCH LOADING?"

NO

Since the answer to the latter question was negative, locations 0 to 507, and 520 to 1057 and 1100 to 1777 will be filled by the code for NOP.

However, if the reply were "Y" the only locations affected in the ROM-RAM would be 510 to 517 and 1060 to 1077. The rest of the memory would remain unchanged.

This feature of the loader can be used to combine (PATCH) object files of different source programs.

Kamran Firooz  
December/1974





2100 to 9830

This program enables the user to transfer his binary files from 2100 to a 9830 calculator. The object files produced by 2100 Nano Processor Assembler are first punched on paper tape as follows:

:D0,4,name

The punched binary tape can then be read by this program using a paper tape reader and stored on cassette tapes. The files stored by this program are compatible to the files produced by the 9830 Assembler, therefore, they can be loaded into a ROM-RAM Simulator using the 9830 Nano Processor loader program.

Kamran Firooz  
December/1974

```
30 RLI ----- 2100 TO 9830 -----
10 RLI 2100-9830 F1R002 DECEMBER 1974
20 END
30 CLR
40 CLR 2100 PROGRAM READS AN OBJECT FILE PRODUCED BY THE 2100 NANO PROCESSOR
50 AND ASSEMBLER AND STORES THE FILE ON A CASSETTE TAPE.
60 FOR COMPATIBILITY FILES ARE COMPATIBLE TO THE FILES PRODUCED BY THE 9830 ASSEMBLER
70 FOR MICROPROCESSORS THEY CAN BE LOADED INTO A ROM-RAM SIMULATOR USING THE LOADER
80 SEM 100EAM.
90 DIM A(256)
100 P=0
10 DTPR "OBJECT CODE NUMBER:"
20 T=0
30 D OF "PROGRAM FILE NO.:"
40 INPUT L
50 FOR I=1 TO 500
60 D-RBYTE
70 IF M=0 THEN 300
80 NEXT I
90 GOTO 200
100 IF M=255 THEN 320
110 M=255
120 FOR J=1 TO 4
130 FOR L=1 TO 256
140 P=K+M
150 K=RP*100
160 M=H*100
170 IF P=256 THEN 390
180 P=P-1
190 A(L)=J+P
200 NEXT L
10 NEXT J
10 NEXT I
30 STORE DATA L:A
30 END
```

## 9830 to 2100

This program punches a source file generated by the Nano Processor Editor on paper tape using a paper tape punch. The paper tape can then be stored on a disk of 2100 computer as follows:

```
:ST,S,S, name
```

The stored program can be assembled using the 2100 Nano Processor Assembler (NPA).

In addition to this program, one may use the Terminal mode of a 9830 calculator to create or edit his programs, and then by using:

```
LISTX #select code number for punch.
```

Transfer the source program onto paper tape and then into a 2100.

Transfer of Source Programs from 9830 to 2100 can also be done by use of 9830 ASCII card interfaced to an HP1B card of a 2100 or the teletype plug.

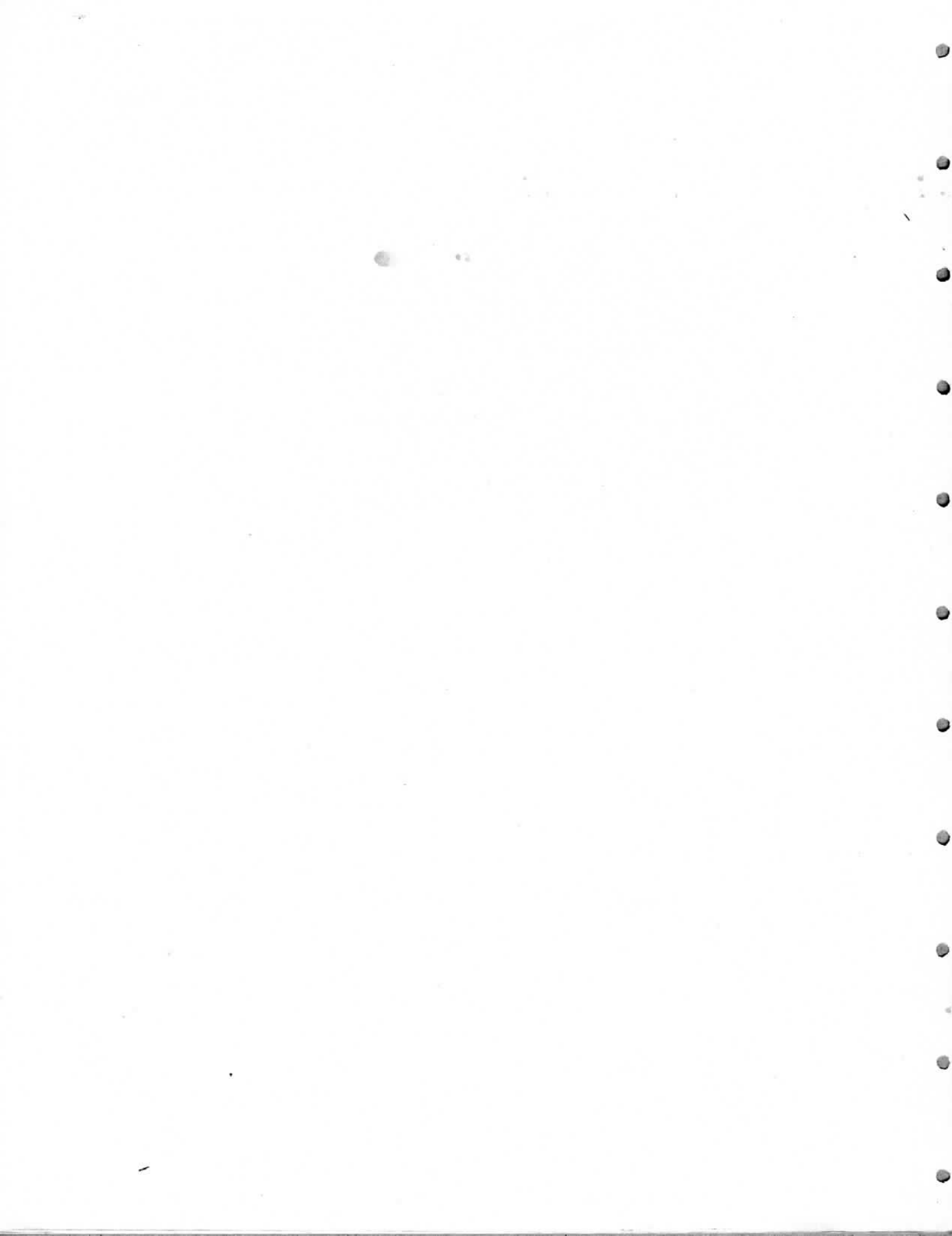
Kamran Firooz  
December/1974



```
100 REM ----- 9830 TO 2100 -----
110 REM 2100 NANO FT500Z DECEMBER 1974.
120 REM
130 REM
140 REM THE 2100 NANO PUNCHES A SOURCE FILE GENERATED BY THE NANO PROCESSOR
150 REM FROM PAPER TAPE ON PAPER TAPE USING A PAPER TAPE PUNCH.
160 REM
170 REM THE PAPER TAPE CAN THEN BE STORED ON A DISK OF A 2100 COMPUTER.
180 REM THE STORED PROGRAM CAN BE ASSEMBLED USING THE 2100 NANO PROCESSOR
190 REM WITH ASSEMBLER (NPA).
200 REM
210 DIM A$(1+9),B$(32),N$(60)
220 DEF FN SELECT CODE NO. :
230 INPUT I
240 GOSUB 100
250 INPUT I
260 LOAD DATA 1,0
270 GOSUB 150
280 FOR I=0 TO 140
290 TRANSFER A(I+1) TO N#
300 P=POS(4+I)
310 IF P=0 THEN 370
320 N#=#
330 GOSUB 140
340 N#=#
350 N#=#+I+32
360 GOTO 340
370 P=POS(4+I)
380 IF P=0 THEN 410
390 N#=#
400 GOSUB 440
410 NEXT I
420 GOSUB 330
430 END
440 N=P-10+(P/2)*2
450 IF N#0 THEN 480
460 N#=#+P
470 P=P+1
480 WRITE (P+3)N#(1,P-1)
490 IF POS(4)="#EOF" THEN 420
500 IF N#0 THEN 530
510 P=P+1
520 RETURN
530 FOR I=1 TO 200
540 WRITE (I+3)N#(I,200)
550 NEXT I
560 RETURN
```

APPENDIX

E



GENERAL INFORMATION

Nano Processor support materials may be ordered from LID at no charge. The Nano Processor User's Guide may be obtained by asking for drawing number A-5955-0331-1. Also a limited quantity of software material, ie 9830A EDITOR, ASSEMBLER, LOADER CASSETTE, and 2100 DOS III paper tape are available at no charge.

When ordering, specify one of the following;

9830A NANO PROCESSOR SOFTWARE CASSETTE

P/N 5061-0768

2100 NANO PROCESSOR SOFTWARE PAPER TAPE

P/N 5061-0769

These materials are limited, and when the supply is exhausted no more will be available. You may be able to check around your division for others who have these software materials.



# NANO PROCESSOR

George Latham

7/29/75

## "DATA BUS APPLICATION HINTS"

- I. Data bus rise time equations where " $C_D$ " is the total data bus capacitance external to the Nano Processor package.
- A. "A" Chips  
Rise time =  $7.7(C_D + 7)ns$  ( $C_D$  in pF)
  - B. "B" Chips  
Rise Time =  $9.6(C_D + 7)ns$  ( $C_D$  in pF)
  - C. "C" Chips (NO LONGER AVAILABLE)  
Rise Time =  $11.5(C_D + 7)ns$  ( $C_D$  in pF)
  - D. We will define data bus rise time as: r.t.(data) (to a 4.0 Volt level)
  - E. A max. loading of one T<sup>2</sup>L input (1.6mA) is allowed.
  - F. The data bus rise time is measured from the last data output low ("0") or from the last moment the ROM applies a zero on the data bus. It is therefore important to watch the time between TPA2 valid and T<sub>PGH</sub> (program gate) to be sure that the ROM does not glitch the data bus low just before a valid high ("1") is output by the ROM. It is reasonable to use maximum values of both TPA2 and T<sub>PGH</sub> at the same time.
  - G. The equations for interrelating these parameters are:
    - 1.  $CLK\tau \geq r.t. (data) + T_{IP} + T_{DV} max.$
    - 2.  $CLK\tau \geq min. spec.$
    - 3.  $CLK\tau \geq T_{PA2} + T_{AA} + T_{IP}$   
Note: T<sub>AA</sub> include r.t.(data) to 4.0 Volts
    - 4.  $CLK\tau \geq T_{PGH} max. + T_{EA} + T_{IP}$
- Reasonable values to use for T<sub>DV</sub> max. (not speced) are:  
"A" Chip, 110ns; "B" Chips, 140ns; "C" Chips, 170ns.

## II. Other suggested Data Bus Pull-Up Methods.

- A. The most simple pull-up method for greater data bus speeds is to connect a 10K resistor from each data bus to 12V or 9V for "A" or "B" & "C" chips respectively. This is permissible providing that the circuits on the data bus have a maximum voltage rating of at least 7.0 volts. One LS input is assumed (0.36mA). The principle of operation is that then N.P. output pull-up FETS will self-clamp the data bus to approximately 6.5 Volts using 10K ohm resistors. Design constants for this method are:

"A" Chips: r.t.(data) = 2.4 (C<sub>D</sub> + 7)ns

"B" Chips: r.t.(data) = 3.6 (C<sub>D</sub> + 7)ns

"C" Chips: r.t.(data) = 3.9 (C<sub>D</sub> + 7)ns (C<sub>D</sub> IN pF)

- B. A faster yet method which yields a lower clamp voltage (≈5.5 Volts max.) uses 7.5K ohm resistors connected as in part A, but also connects clamping Schottky diodes between the data bus and the 5 volts supply. Design constants for this method are:

"A" Chips: r.t.(data) = 2.2 (C<sub>D</sub> + 7)ns

"B" Chips: r.t.(data) = 3.1 (C<sub>D</sub> + 7)ns

"C" Chips: r.t.(data) = 3.4 (C<sub>D</sub> + 7)ns (C<sub>D</sub> in pF).