LID
NANOPROCESSOR

Users' Guide

HEWLETT PACKARD
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.
TABLE OF CONTENTS

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

APPENDICES
A. SPEC. SHEET
B. ROM–RAM SIMULATOR
C. NANO PROCESSOR DESIGN EXAMPLES
D. SOFTWARE
E. GENERAL INFORMATION
I. INTRODUCTION.

The 64P 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 (DØ 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 (DCØ 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 (RØ – R17)
E. One 8-Bit Magnitude Comparator (CMP)
F. Seven Bidirection Direct Control I/O Lines (DCØ – 6)
G. One 11-Bit Program Counter (PC)
H. One 11-Bit Subroutine Return Register (SRR)
I. One 11-Bit Interrupt Return Register (IRR)
CONTROL SIGNALS

SET CMD 0
CLR CMD 0
FLAG 0
SET CMD 1
CLR CMD 1
FLAG 1
SET CMD 2
CLR CMD 2
FLAG 2
SET CMD 3
CLR CMD 3
FLAG 3
SET CMD 4
CLR CMD 4
FLAG 4
SET CMD 5
CLR CMD 5
FLAG 5
SET CMD 6
CLR CMD 6
FLAG 6

COMMAND FLIP FLOPS

INPUT/OUTPUT SIGNALS

DC0
DC1
DC2
DC3
DC4
DC5
DC6

DIRECT CONTROL
I/O STRUCTURE
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. R6 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 R6 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 listed in Appendix A.)
MAIN CLOCK INPUT

FETCH PHASE

EXECUTE PHASE

PROGRAM ADDRESS VALID (PAO-10)

VALID

TPA1

VALID

TPA2

PROGRAM GATE

TPGH

TPGL

TPGH

TPGL

PROGRAM GATE FOR TWO BYTE INSTRUCTIONS

PROGRAM GATE FOR OTA \& INA INSTRUCTIONS

PROGRAM ACCESS FROM ADDRESS TAA

\[ t_{IP} \]

FOR MAX. SPEED 90ns MAX.

PROGRAM ACCESS FROM PROGRAM GATE \( t_{EA} \)

INSTRUCTION POINT PROGRAM DATA MUST BE ON DATA BUS BY THIS POINT IN TIME.
DC/IO LINES

MAIN CLOCK INPUT

DEVICE SELECT AND R/W OUTPUTS

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 INPUT (R/W LOW)

DATA MUST BE ON BUS BY THIS POINT IN TIME.

OUTPUT DATA SHOULD BE STORED ON LEADING EDGE OF CLOCK.

DATA OUTPUT (R/W HIGH)

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

OUTPUT DATA NO LONGER VALID

\[ t_{bc}, t_{d1}, t_{d0}, t_{dv} \]
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 incrementated. 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 R0= 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:

<table>
<thead>
<tr>
<th>Program Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>After the skip instruction</td>
<td>N</td>
</tr>
<tr>
<td>This instruction is executed</td>
<td>N + 1</td>
</tr>
<tr>
<td>if Bit 3 is zero</td>
<td>N + 2</td>
</tr>
<tr>
<td>This instruction is executed</td>
<td>N + 3</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>SBS</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Mnemonic</td>
<td>Operand(s)</td>
</tr>
</tbody>
</table>

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 \(Rφ(Rφφ - Rφ3)\).

Example:

\[
\begin{align*}
&I_0 - I_3 & 1001 \\
&Rφφ - Rφ3 & 0101 \\
&Effective Register & 1101 \\
&Address
\end{align*}
\]

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

***Note: Since \(Rφ\) is used as the index, caution should be used so that \(Rφ\) is not the effective destination of a Store instruction.***
### V - A. Accumulator Group.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>SBS N</td>
<td>Skip on accumulator bit #N Set (1)</td>
<td>00010 N</td>
</tr>
<tr>
<td>SBZ N</td>
<td>Skip on accumulator bit #N zero (0)</td>
<td>00110 N</td>
</tr>
<tr>
<td>SBN N</td>
<td>Set accumulator bit #N</td>
<td>00100 N</td>
</tr>
<tr>
<td>CBN N</td>
<td>Clear accumulator bit #N</td>
<td>10100 N</td>
</tr>
<tr>
<td>INB</td>
<td>Increment accumulator as an 8-bit binary number. The extend register is set if overflow occurs.</td>
<td>00000000 N</td>
</tr>
<tr>
<td>IND</td>
<td>Increment accumulator as two BCD code decimal numbers (1(1)). Carry between digits is automatically handled. The extend register is set if overflow occurs.</td>
<td>000000010</td>
</tr>
<tr>
<td>DEB</td>
<td>Decrement accumulator as an 8-bit binary number. The extend register is set if underflow occurs.</td>
<td>000000001</td>
</tr>
<tr>
<td>DED</td>
<td>Decrement accumulator as two BCD coded decimal digits. Borrow between digits is automatically handled. The extend register is set if underflow occurs.</td>
<td>000000011</td>
</tr>
<tr>
<td>CLA</td>
<td>Clear accumulator. Does not affect the extend register.</td>
<td>00000100</td>
</tr>
<tr>
<td>CMA</td>
<td>Complement accumulator. The accumulator is treated as an 8-bit binary number and one's complement is performed.</td>
<td>00000101</td>
</tr>
<tr>
<td>LSA</td>
<td>Left shift accumulator 1-bit shift with zero (0) fill. Does not affect extend register.</td>
<td>00000111</td>
</tr>
<tr>
<td>RSA</td>
<td>Right shift accumulator 1-bit shift with zero (0) fill. Does not affect extend register.</td>
<td>00000110</td>
</tr>
<tr>
<td>SES</td>
<td>Skip on extend register set (1).</td>
<td>00011111</td>
</tr>
<tr>
<td>SEZ</td>
<td>Skip on extend register Zero (0).</td>
<td>00111111</td>
</tr>
<tr>
<td>LDR</td>
<td>ROM Data. Load accumulator with ROM data. (ROM data is the second byte of this instruction)</td>
<td>11001111</td>
</tr>
</tbody>
</table>
V - B. REGISTER TRANSFER GROUP.

**LDA R**
Load accumulator with data from register #R.

**STA R**
Store accumulator at register #R.

**LDI Z**
Load accumulator with data from register addressed by [Z]v(R0). (See description of indexing.)

**STI Z**
Store accumulator at register addressed by (z) v (R0).

**STR R,**
ROM Data.
Store ROM data at register #R.
ROM data is the second byte of this instruction.

V - C. EXTEND REGISTER GROUP.

**STE**
Set extend register.

**CLE**
Clear extend register.

V - D. INTERRUPT GROUP.

**DSI**
Disable the interrupt.

**ENI**
Enable the interrupt.
V – E. COMPARATOR GROUP.

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

SLT
Skip on accumulator less than \( R0 \).

SEQ
Skip on accumulator equal to \( R0 \).

SAZ
Skip on accumulator equal to zero (0).

SLE
Skip on accumulator equal to or equal to \( R0 \).

SGE
Skip on accumulator greater than or equal to \( R0 \).

SNE
Skip on accumulator not equal to \( R0 \).

SAN
Skip on accumulator not equal to zero (0).

SGT
Skip on accumulator greater than \( R0 \).

V – F. INPUT/OUTPUT GROUP.

INA DS
Input data from device \#DS to accumulator.

0 1 0 0  DS

OTA DS
Output accumulator data to device \#DS.

0 1 0 1  DS

OTR DS, ROM DATA
Output ROM data to device \#DS

1 1 0 0  DS

ROM Data.

ROM DATA

STC K
Set direct control.

0 0 1 0 1  K

Bit \#K

CLC K
Clear direct control.

1 0 1 0 1  K

Bit \#K

SFS J
Skip on direct control.

0 0 0 1 1  J

Flag \#J Set (1).

SFZ J
Skip on direct control flag \#J zero (0).

0 0 1 1 1  J
RTI
Return from interrupt.
An unconditional jump to the location stored in the interrupt stack register is performed.
The interrupt control bit is not affected.

RTE
Return from interrupt and enable interrupt.
Same as RTI instruction except that the interrupt control bit is set allowing future interrupt.

NOP
NO Operation.

JAI
Jump indirect (through accumulator) indexed.
The page number is the indexed value (Z) v (R0).
The page offset is the accumulator.
An unconditional jump to the address formed from the page number and page offset.

JAS
Jump indirect (through accumulator) indexed to subroutine.
Same as JAI with the addition that the location of the JAS instruction Plus 2 is stored in the subroutine stack register.

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 R0's 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.
V - G. PROGRAM CONTROL GROUP.

JMP ADDRESS

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

(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 cannot be nested.

RTS

Return from subroutine.
An unconditional jump to the location stored in the subroutine stack register is performed.
The location of the RTS instruction Plus 2 is stored in the subroutine stack register, thus co-routine linkages may be performed.

RSE

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 (PA0 – 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.
**Pin Out**

**Program Address Lines**
- PA0
- PA1
- PA2
- PA3
- PA4
- PA5
- PA6
- PA7
- PA8
- PA9
- PA10

**Device Select**
- DS3
- DS2
- DSI
- DSO

**I/O Read/Write**
- R/W
- GND

**I/O Lines**
- INT
- ENA
- REQ
- ACK

**Direct Control**
- VGG
- VDD
- VBG
- DCO
- DC1
- DC2
- DC3
- DC4
- DC5
- DC6

**+9 or +12 Volts**

**5 Volts**

**-2 to -5 Volts**

**Interrupt Lines**
- CLK
- PSG

**Main Clock Input**
- D7
- D6
- D5
- D4
- D3

**Data Bus**

*±5%, <1W Total*
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

MAIN CLOCK INPUT

EXECUTE PHASE → INTERRUPT PHASE → FETCH PHASE OF FIRST INSTRUCTION OF INTERRUPT SERVICE

INTERRUPT REQUEST INPUT

INT REQ STORED AT THIS POINT

INT REQ MUST BE REMOVED BY THIS POINT TO PREVENT SECOND INT PHASE

INTERRUPT ACKNOWLEDGE OUTPUT

T _IAH → T _IAL

MINIMUM VECTOR ACCESS TIME 100ns

VECTOR MUST BE ON DATA BUS BY THIS POINT IN TIME

T _VA → T _VI

INTERRUPT ENABLE OUTPUT

T _IE MAX → DISABE

THIS LINE MAY BE USED TO GATE OFF INT REQ

INTERRUPT RESPONSE TIME:
(MAX: ONE FULL INSTRUCTION TIME OVER MINIMUM)
Nano Processor Turn On
Valid Startup Sequence

$V_{DD}$

$V_{BG}$

$V_{GG}$

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

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

Sample Circuit (Schematic)

+9 or +12 $V_{GG}$ (pin 40)

All supplies are valid and clock is running

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

MCLK
APPENDIX A
## NANOPROCESSOR SPECIFICATIONS
Revision Date 11/10/75

<table>
<thead>
<tr>
<th></th>
<th>1820-1692 NP-A (500ns)</th>
<th>1820-1691 NP-B (750ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MIN</td>
<td>TYP</td>
</tr>
<tr>
<td>$V_{GG}$ (mA)</td>
<td>+12</td>
<td>30</td>
</tr>
<tr>
<td>$I_{GG}$ (mA)</td>
<td>4.75</td>
<td>5.5</td>
</tr>
<tr>
<td>$I_{DD}$ (mA)</td>
<td>110</td>
<td></td>
</tr>
<tr>
<td>$V_{BG}$ (mV)</td>
<td>-2</td>
<td>-3</td>
</tr>
<tr>
<td>$Pd$ (mW)</td>
<td>800</td>
<td>1000</td>
</tr>
<tr>
<td>$I_{BG}$ (mA)</td>
<td>500</td>
<td></td>
</tr>
<tr>
<td>$CLK$ (ns)</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>$T_{PA1}$</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>$T_{PA2}$</td>
<td>60</td>
<td>95</td>
</tr>
<tr>
<td>$T_{PGH}$</td>
<td>15</td>
<td>20</td>
</tr>
<tr>
<td>$T_{PGL}$</td>
<td>5</td>
<td>15</td>
</tr>
<tr>
<td>$T_{IP}$</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>$T_{AA}$</td>
<td>90</td>
<td></td>
</tr>
<tr>
<td>$T_{EA}$</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>$T_{DC}$</td>
<td>40</td>
<td>90</td>
</tr>
<tr>
<td>$T_{DS1}$</td>
<td>85</td>
<td></td>
</tr>
<tr>
<td>$T_{DS2}$</td>
<td>30</td>
<td>85</td>
</tr>
<tr>
<td>$T_{DI}$</td>
<td>40</td>
<td>60</td>
</tr>
<tr>
<td>$T_{DO}$</td>
<td>150</td>
<td></td>
</tr>
<tr>
<td>$T_{DV}$</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>$T_{IV}$</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>$T_{IRR}$</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>$T_{IAH}$</td>
<td>115</td>
<td>140</td>
</tr>
<tr>
<td>$T_{IAL}$</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>$T_{VA}$</td>
<td>95</td>
<td>155</td>
</tr>
<tr>
<td>$T_{V1}$</td>
<td>40</td>
<td>60</td>
</tr>
<tr>
<td>$T_{IE}$</td>
<td>250</td>
<td></td>
</tr>
<tr>
<td>AMBIENT TEMPERATURE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Moving air</td>
<td>80°C</td>
<td></td>
</tr>
<tr>
<td>Still air</td>
<td>70°C</td>
<td></td>
</tr>
</tbody>
</table>

I.C. will be marked

CLOCK AT TYP. VOLTAGES

PROG. ACCESS

DCIO

DS & R/W

DATA IN

DATA OUT

INTERRUPT

acknowledge

TIAH

TIAL

TVA

TV1

TIE
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 sunked when input low is 0.4 volts. See MCLK for an exception.

3. MCLK may require up to 3.0ma to be sunked 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 < 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.
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 = BIAND (ROT (I,8),3)
60   B = BIAND (I, 255)
70   CMD "?U3"
80   WRITE (13,90)A,B,C;
90   FORMAT 3B
100  GO TO 10
```
## ROM-RAM SIMULATOR COMPONENTS

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

<table>
<thead>
<tr>
<th>I/O SW</th>
<th>DESCRIPTION</th>
<th>PART NO</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>DPDT SWITCH 7203</td>
<td>3101-0939</td>
</tr>
<tr>
<td>CONT. SW</td>
<td>SPDT SWITCH 7101</td>
<td>3101-1258</td>
</tr>
<tr>
<td>WRITE SW</td>
<td>SPST PUSH BUTTON</td>
<td>3101-0063</td>
</tr>
<tr>
<td>C₄</td>
<td>47 UF CAPACITOR</td>
<td>0180-0097</td>
</tr>
<tr>
<td>C₂-C₇</td>
<td>.47 UF CAPACITOR</td>
<td>0160-0174</td>
</tr>
<tr>
<td>R₁</td>
<td>330 Ω RESISTOR</td>
<td>1810-0136</td>
</tr>
<tr>
<td>R₂-R₄</td>
<td>1K RESISTORS</td>
<td>0683-3315</td>
</tr>
<tr>
<td></td>
<td>7300 LED DISPLAY</td>
<td>0683-1025</td>
</tr>
<tr>
<td></td>
<td>24 PIN ASCII CONNECTOR</td>
<td>5082-7300</td>
</tr>
<tr>
<td></td>
<td>.01 UF CAPACITOR</td>
<td>1251-3283</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0150-0093</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Let</th>
<th>ADDR</th>
<th>be the select code of the Address Storage Latch.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DATA</td>
<td>be the select code of the Data Storage.</td>
</tr>
<tr>
<td></td>
<td>ALU</td>
<td>be the select code of the Arithmetic Unit.</td>
</tr>
<tr>
<td></td>
<td>FN</td>
<td>be the select code of the Function Storage.</td>
</tr>
<tr>
<td>and</td>
<td>RAM</td>
<td>be the WRITE ENABLE line of the RAM.</td>
</tr>
</tbody>
</table>

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)
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. $A1\bar{0}$ is used as the "current block" indicator. $A1\bar{0} = \phi$.
5. Subroutines may be in block H or current block. Return is to current block.
6. Interrrrups are to block H. Return is to current block.
7. Blocks are 1024 words each.

NANO PROCESSOR
BLOCK SWITCHING (II)
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.
The following is a documentation of the Nano Processor software.

For further information, recommendations, or in case of difficulty please contact Kamran Firooz 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 Processor 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.

\( \dagger \) : To store the edited line and edit the next line.

\( \dagger \dagger \) : 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 DSO *INPUT THE BCD #/
STA R5 *STORE THE # IN R5/
STR 0,0 *CLEAR R0:LOOP LDA 5/
SAN *SKIP IF ACCH0: JMP OUT/
DEC *DECREMENT IN BCD: STA 5/
LDA 0: INC *INCREMENT IN BIN. /
STA 0: JMP LOOP:OUT LDA 0/
OUT DSI *OUTPUT THE BIN. #/
JMP START *READ ANOTHER NUMBER/
DSO 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 INA D50 =INPUT THE BCD #/
7 STA R5 =STORE THE # IN R5:
8 STA 0,0 =CLEAR R0: LOOP LDA 1/
9 SNN *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 STA R51 *OUTPUT THE BIN, #/
14 JMP START *READ ANOTHER NUMBER/
15 D50 OCT 0: R5 OCT 5: END: ZOF/

TOTAL LINE NUMBER= 15
To list the lines separated by ":" individually type:

LISTP

```plaintext
* 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 /
SAM *SKIP IF ACC #0
JMP OUT /
DED * DECREMENT IN BCD
STA 5 /
LDA 0
INC * 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 /
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:

\texttt{DS1-OCT 1 /}

"OTA 2 :DS1 OCT 1 /"

Since no further changes are required press \texttt{EXIT}

"FILE EDIT?"
To observe the changes type:

\texttt{LISTP 10, 15}

\begin{verbatim}
10  DEK #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/
\end{verbatim}

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 I "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 17 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 –1. During the assembling –1 is overwritten by other codes, however the locations not used will remain as –1. 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 Interrupt
INB * Increment the ACC in Binary
SLE * Skip if ACC ≤ 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 RE,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 OP CODES:

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 (70). 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 arrange-
ment 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 Firdoz
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.

```
1    * NANO 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    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, FXECUTE. 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 ON FILE 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**

<table>
<thead>
<tr>
<th>SYMBOL</th>
<th>ADDRESS (VALUE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOOP</td>
<td>0</td>
</tr>
<tr>
<td>OUT</td>
<td>13</td>
</tr>
<tr>
<td>R5</td>
<td>5</td>
</tr>
<tr>
<td>R6</td>
<td>6</td>
</tr>
</tbody>
</table>

Number of errors for pass 1 = 0
ADD THE CONTENTS OF REG. 5
TO THE CONTENTS OF REG. 6
AND STORE THE RESULT IN REG. 6

| 0 | 145 | LOOP | LDA R5 | #LOAD ACC FROM R5
| 1 | 003 | DED | DED R5 | #DECREMENT IN DECIMAL
| 2 | 017 | SAN | SAN R5 | #SkiP IF ACC#A
| 3 | 200 | JMP | OUT | #STORE ACC AT R6
| 4 | 013 | STA | R5 | #LOAD ACC FROM R6
| 5 | 165 | | | #INCREMENrT IN DECIMAL
| 6 | 146 | LDA | R6 | #STORE ACC AT R6
| 7 | 002 | IND | R5 | #R6 HAS THE RESULT
| 8 | 166 | STA | R6 | #R6 IS OCTAL 5
| 9 | 200 | JMP | LOOP | #R6 IS OCTAL 5
| 10 | 000 | OUT | LDA | R6
| 11 | 146 | | | #R6 IS OCTAL 5
| 12 | 032 | IND | R5 | #R6 IS OCTAL 5
| 13 | 166 | STA | R6 | #R6 IS OCTAL 5
| 14 | | | | #R6 IS OCTAL 5
| 15 | | | | #R6 IS OCTAL 5
| 16 | | | | #R6 IS OCTAL 5
| 17 | | | | #R6 IS OCTAL 5
| 18 | | | | #R6 IS OCTAL 5

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.
DISPLAY ROUTINE:

1) CLR R1:0 - CLEAR REGISTER 1
2) READ PTR R0:0 = CLEAR R0
3) DCS FLAG = SKIP IF DC0 IS SET
4) JMP DISP = DISPLAY MESSAGE A
5) DCF FLAG = SKIP IF DC1 IS SET
6) JMP DISP = DISPLAY MESSAGE B
7) ENI = ENABLE THE INTERRUPT
8) LPF READ = BOTH FLAGS ARE CLEAR
9) DISP STR R0:40 * REG0 = 40
10) LDA R1 * LOAD ACC FROM POINTER
11) SLT = SKIP IF ACC < 40
12) STR 1:0 = CLEAR THE POINTER
13) ENI = JMP READ
14) LDA R1 * LOAD ACC FROM POINTER
15) JSR R1:40 = POINTER = 40
16) ENI = JMP READ
17) LDA R1 * LOAD ACC FROM POINTER
18) SGE = SKIP IF ACC >= 40
19) STA R2 * STORE THE ACC DURING
20) THE INTERRUPT PERIOD
21) LDA R1 * LOAD ACC FROM POINTER
22) OUT DS0 = OUTPUT ACC TO ADDRESS
23) * LATCH (DEVICE SELECT 0)
24) HAJ 2 = JUMP INDIRECT TO LOC
25) 010 AND 8 BITS OF ACC
26) BL OTR DS2:40 = OUTPUT BLANK
27) JMP DISP
28) OTR DS2:101 = OUTPUT 'A' CODE
29) JMP DISP
30) OTR DS2:104 = OUTPUT 'D' CODE
31) JMP DISP
32) OTR DS2:105 = OUTPUT 'E' CODE
33) JMP Disp
34) OTR DS2:111 = OUTPUT 'I' CODE
35) JMP Disp
36) OTR DS2:113 = OUTPUT K
37) JMP Disp
38) OTR 2:114 = OUTPUT L: JMP Disp
39) OTR 2:116 = OUTPUT H: JMP Disp
40) OTR 2:117 = OUTPUT O: JMP Disp
41) P OTR 2:120 = OUTPUT P: JMP Disp
42) P OTR 2:122 = OUTPUT R: JMP Disp
43) S OTR 2:123 = OUTPUT S: JMP Disp
44) T OTR 2:131 = OUTPUT Y: JMP Disp
45) DISP OTR DS1:40 = OUTPUT CODE
46) FOR BLANK TO DEVICE 1:
47) OTR DS3 = START THE DISPLAY
48) INB
49) INB = DOUBLE INCREMENT THE
50) * POINTER:
51) STA 1: LDA 2 = RELOAD THE ACC
52) * BY ITS VALUE BEFORE THE
53) INTERRUPT OCCURED:
54) MID ENI = ENABLE THE INTERRUPT
57  \texttt{BEFORE RETURN:*:*/*}
58  \texttt{RIT \*RETURN FROM INTERRUPT/}
59  \texttt{RESETA STR R1,0 \*RESET POINTER/}
60  \texttt{JMP MIDD/}
61  \texttt{RESETB STR R1,40 \*RESET POINTER/}
62  \texttt{JMP MIDD/}
63  \texttt{ORG 1000 \*DISPLAY:*:*/*}
64  \texttt{DISPLAY IS OK:*:*/*}
65  \texttt{JMP BL \*DISPLAY BLANK/}
66  \texttt{JMP D \*DISPLAY D/}
67  \texttt{JMP I \*DISPLAY I/}
68  \texttt{JMP S \*DISPLAY S/}
69  \texttt{JMP P: JMP L: JMP A: JMP Y/}
70  \texttt{JMP BL: JMP I: JMP S: JMP BL/}
71  \texttt{JMP O: JMP K: JMP RESETA/}
72  \texttt{ORG 1040 \*DISPLAY:*:*/*}
73  \texttt{+ ERROR IN DISPLAY:*:*/*}
74  \texttt{JMP E: JMP R: JMP R/}
75  \texttt{JMP O: JMP R: JMP BL: JMP I/}
76  \texttt{JMP N: JMP BL: JMP D: JMP I/}
77  \texttt{JMP S: JMP P: JMP L: JMP A/}
78  \texttt{JMP Y: JMP RESETB/}
79  \texttt{R0 OCT 0:R1 OCT 1:R2 OCT 2/}
80  \texttt{DS0 OCT 0:DS1 OCT 1/}
81  \texttt{DS3 OCT 2:DS3 OCT 3/}
82  \texttt{FLAG0 OCT 0:FLAG1 OCT 1: END/}
83  \texttt{EOF/}

\textbf{TOTAL LINE NUMBER= 83}
<table>
<thead>
<tr>
<th>SYMBOL</th>
<th>ADDRESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>PEAH0</td>
<td>2</td>
</tr>
<tr>
<td>DISP</td>
<td>15</td>
</tr>
<tr>
<td>DISP2</td>
<td>26</td>
</tr>
<tr>
<td>BL</td>
<td>483</td>
</tr>
<tr>
<td>D</td>
<td>407</td>
</tr>
<tr>
<td>E</td>
<td>413</td>
</tr>
<tr>
<td>T</td>
<td>417</td>
</tr>
<tr>
<td>K</td>
<td>423</td>
</tr>
<tr>
<td>L</td>
<td>427</td>
</tr>
<tr>
<td>M</td>
<td>433</td>
</tr>
<tr>
<td>N</td>
<td>437</td>
</tr>
<tr>
<td>O</td>
<td>443</td>
</tr>
<tr>
<td>P</td>
<td>447</td>
</tr>
<tr>
<td>R</td>
<td>453</td>
</tr>
<tr>
<td>S</td>
<td>457</td>
</tr>
<tr>
<td>Y</td>
<td>463</td>
</tr>
<tr>
<td>DISP</td>
<td>467</td>
</tr>
<tr>
<td>MID</td>
<td>476</td>
</tr>
<tr>
<td>RESETH</td>
<td>500</td>
</tr>
<tr>
<td>RESETB</td>
<td>504</td>
</tr>
<tr>
<td>RO</td>
<td>0</td>
</tr>
<tr>
<td>R1</td>
<td>1</td>
</tr>
<tr>
<td>R2</td>
<td>2</td>
</tr>
<tr>
<td>DS0</td>
<td>0</td>
</tr>
<tr>
<td>DS1</td>
<td>1</td>
</tr>
<tr>
<td>DS2</td>
<td>2</td>
</tr>
<tr>
<td>DS3</td>
<td>3</td>
</tr>
<tr>
<td>FLAG0</td>
<td>0</td>
</tr>
<tr>
<td>FLAG1</td>
<td>1</td>
</tr>
</tbody>
</table>

NUMBER OF ERRORS FOR PASS 1 = 0
DISPLAY ROUTINE

321 STR R1,0 *CLEAR REGISTER 1
000
320 READ STR R0,0 *CLEAR R0
000
300 SFS FLG0 *SKIP IF DC0 IS SET
015 JMP DISPA *DISPLAY MESSAGE A
400 200 SFS FLG1 *SKIP IF DC1 IS SET
015 JMP DISPB *DISPLAY MESSAGE B
10 002
10 057 ENI *ENABLE THE INTERRUPT
13 JMP READ *BOTH FLAGS ARE CLEAR
14 002
10 020 DISPA STR R0,40 *REGO=40
16 040
10 141 LDA R1 *LOAD ACC FROM POINTER
20 011 SLT *SKIP IF ACC < 40
20 321 STR 1,0 *CLEAR THE POINTER
22 000
23 057 ENI
24 JMP READ
25 002
20 020 DISPB STR R0,40 * REGO=40
22 040
10 141 LDA R1 *LOAD ACC FROM POINTER
18 015 SGE *SKIP IF ACC>=40
18 321 STR R1,40 *PONITER=40
25 000
26 057 ENI
26 JMP READ
26 002
377 ORG 377 *INTERRUPT ROUTINE
50 STA R2 *STORE THE ACC DURING
141
120 (LATCH DEVICE SELECT 0)
141

THE INTERRUPT PERIOD
402 JAI 2 *JUMP INDIRECT TO LOC.
222
400 AND 8 BITS OF ACC
380 BL OTR DS2,40 *OUTPUT BLANK
404 040
401 JMP DISP
405 201
406 067
407 302 H OTR DS2,101 *OUTPUT 'H' CODE
410 101
411 201 JMP DISP
412 067
413 302 D OTR DS2,104 * OUTPUT 'D' CODE
414 104
415 201 JMP DISP
32 415 201
42
48  479  040  DISP  OTR  DS1,40  *OUTPUT CODE
47 + FOR BLANK TO DEVICE 1
46  466  067  467  301  DISP
45  465  067  464  201  JMP DISP
44  463  067  462  131  JMP DISP
43  461  201  460  123  JMP DISP
42  459  067  458  302  OTR  2,123
41  457  123  456  201  OTR  2,122
40  455  067  454  201  OTR  2,122
39  453  302  452  067  OTR  2,120
38  451  201  450  120  OTR  2,117
37  449  067  448  302  OTR  2,116
36  447  120  446  201  OTR  2,114
35  445  067  444  117  OTR  2,114
34  443  302  442  067  441  201
33  440  116  439  302  438  067
32  437  302  436  114  OTR  DS2,113
31  435  201  434  114  OTR  DS2,111
30  433  302  432  067  OTR  DS2,105
29  431  201  430  185  OTR  DS1,40
28 + FOR BLANK TO DEVICE 1
27  474  161  473  000  STA  1
26  475  142  472  000  STA  1
25  471  123  470  000  STA  1
24 + POINTER
23 +
22 +
21 +
20 +
19 + BY ITS VALUE BEFORE THE INTERRUPT OCCURED
18 +
17 +
16 +
15 +
14 +
13 +
12 +
11 +
10 +
BETORE RETURN

RTI  +RETURN FROM INTERRUPT
RESETA STR R1,0  +RESET POINTER
JMP MIDD
RESETEB STR R1,40  +RESET POINTER
JMP MIDD
ORG 1000  +DISPLAY

DISPLAY IS ON

JMP BL  +DISPLAY BLANK
JMP D  +DISPLAY D
JMP I  +DISPLAY I
JMP S  +DISPLAY S
JMP P
JMP L
JMP A
JMP Y
JMP BL
JMP I
JMP S
JMP BL
JMP O
JMP K
JMP RESETA
ORG 1040  +DISPLAY

ERROR IN DISPLAY

JMP E
74 1042 201 JMP R
74 1043 053
74 1044 201 JMP R
74 1045 053
75 1046 201 JMP Q
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
76 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
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.
+ PATCH PROGRAM FOR DISPLAY/
ORG 510.

1) OTR 5:103 *OUTPUT 103 TO 2/
2) JMP 467 *JUMP TO DISP/
3) OTR 2:127 *OUTPUT 127 TO 2/
4) JMP 467 *JUMP TO DISP/
5) ORG 1060/
6) JMP 403 *JUMP TO BL/
7) JMP 413 *JUMP TO D/
8) JMP 417 *JUMP TO E/
9) JMP 423 *JUMP TO I/
10) JMP 427 *JUMP TO O/
11) JMP 417 *JUMP TO E/
12) JMP 564 *JUMP TO RESETB/
13) END: EDF/
TOTAL LINE NUMBER= 16
**Symbol Table**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>510</td>
<td></td>
</tr>
<tr>
<td>V</td>
<td>514</td>
<td></td>
</tr>
</tbody>
</table>

Number of ERRORS for PASS 1 = 0
* PATCH PROGRAM FOR DISPLAY

ORG 510
L OTR 2:103 *OUTPUT 103 TO C
JMP 467 *JUMP TO DISP
V OTR 2:106 *OUTPUT 127 TO C
JMP 467 *JUMP TO Disp
OGR 1060
JMP 403 *JUMP TO BL
1061 003
1062 201
1063 013
1064 201
1065 017
1066 201
1067 114
1068 201
1069 023
1070 201
1071 110
1072 201
1073 917
1074 201
1075 504 *JUMP TO RESETB
1076 204
1077 104

NUMBER OF ERRORS FOR PASS 2 = 0
## Assembly Language Instructions

### Accumulator Instructions:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Skip on Bit N=1</td>
<td>SBS N</td>
<td>00 010 100</td>
</tr>
<tr>
<td>Skip on Bit N=∅</td>
<td>SBZ N</td>
<td>00 110 100</td>
</tr>
<tr>
<td>Set Bit N</td>
<td>SBN N</td>
<td>00 100 100</td>
</tr>
<tr>
<td>Clear Bit N</td>
<td>CBN N</td>
<td>10 100 100</td>
</tr>
<tr>
<td>Increment ACC (Binary)</td>
<td>INB</td>
<td>00 000 000</td>
</tr>
<tr>
<td>Increment ACC (Decimal)</td>
<td>IND</td>
<td>00 000 010</td>
</tr>
<tr>
<td>Decrement ACC (Binary)</td>
<td>DEB</td>
<td>00 000 001</td>
</tr>
<tr>
<td>Decrement ACC (Decimal)</td>
<td>DED</td>
<td>00 000 011</td>
</tr>
<tr>
<td>Clear ACC</td>
<td>CLA</td>
<td>00 000 100</td>
</tr>
<tr>
<td>Complement ACC</td>
<td>CMA</td>
<td>00 000 101</td>
</tr>
<tr>
<td>Left Shift ACC</td>
<td>LSA</td>
<td>00 000 110</td>
</tr>
<tr>
<td>Right Shift ACC</td>
<td>RSA</td>
<td>00 000 111</td>
</tr>
<tr>
<td>Load ACC with ROM DATA *</td>
<td>LDR DATA</td>
<td>11 001 111</td>
</tr>
<tr>
<td>Skip on E=1</td>
<td>SES</td>
<td>00 011 111</td>
</tr>
<tr>
<td>Skip on E=∅</td>
<td>SEZ</td>
<td>00 111 111</td>
</tr>
<tr>
<td>Set E</td>
<td>STE</td>
<td>10 110 100</td>
</tr>
<tr>
<td>Clear E</td>
<td>CLE</td>
<td>10 110 101</td>
</tr>
</tbody>
</table>

### Register and I/O Instructions:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load ACC From Register R</td>
<td>LDA R</td>
<td>01 10 10 1</td>
</tr>
<tr>
<td>Load ACC Indexed</td>
<td>LDI Z</td>
<td>11 10 10 1</td>
</tr>
<tr>
<td>Store ACC At Register R</td>
<td>STA R</td>
<td>01 11 10 1</td>
</tr>
<tr>
<td>Store ACC Indexed</td>
<td>STI Z</td>
<td>11 11 10 1</td>
</tr>
<tr>
<td>Input To ACC From DS</td>
<td>INA DS</td>
<td>01 00 01 0</td>
</tr>
<tr>
<td>Output ACC To DS</td>
<td>OTA DS</td>
<td>01 01 01 0</td>
</tr>
<tr>
<td>Store ROM DATA At Register R*</td>
<td>STR R,DATA</td>
<td>11 01 01 1</td>
</tr>
<tr>
<td>Output ROM DATA To DS*</td>
<td>OTR DS,DATA</td>
<td>11 00 00 1</td>
</tr>
<tr>
<td>Instruction</td>
<td>Opcode</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>---------</td>
<td>------------------------------------------</td>
</tr>
<tr>
<td>Set Control K</td>
<td>STC K</td>
<td>00 101 K</td>
</tr>
<tr>
<td>Clear Control K</td>
<td>CLC K</td>
<td>10 101 K</td>
</tr>
<tr>
<td>Skip On Flag J=1</td>
<td>SFS J</td>
<td>00 011 J</td>
</tr>
<tr>
<td>Skip On Flag J=Ø</td>
<td>SFZ J</td>
<td>00 111 J</td>
</tr>
<tr>
<td><strong>COMPARATOR INSTRUCTIONS:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Skip On ACC &gt;RO</td>
<td>SGT</td>
<td>00 001 000</td>
</tr>
<tr>
<td>Skip On ACC &lt;RO</td>
<td>SLT</td>
<td>00 001 001</td>
</tr>
<tr>
<td>Skip on ACC = RO</td>
<td>SEQ</td>
<td>00 001 100</td>
</tr>
<tr>
<td>Skip On ACC &gt;RO</td>
<td>SGE</td>
<td>00 001 101</td>
</tr>
<tr>
<td>Skip On ACC &lt;RO</td>
<td>SLE</td>
<td>00 001 100</td>
</tr>
<tr>
<td>Skip On ACC ≠ RO</td>
<td>SNE</td>
<td>00 001 110</td>
</tr>
<tr>
<td>Skip On ACC = 0</td>
<td>SAZ</td>
<td>00 001 011</td>
</tr>
<tr>
<td>Skip On ACC ≠ 0</td>
<td>SAN</td>
<td>00 001 111</td>
</tr>
<tr>
<td><strong>PROGRAM CONTROL INSTRUCTIONS:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jump To Address</td>
<td>JMP ADDRESS</td>
<td>10 000 [PN*]</td>
</tr>
<tr>
<td>Jump Indirect To Address</td>
<td>JAI U</td>
<td>10 010 U</td>
</tr>
<tr>
<td>Jump Sub T Address *</td>
<td>JSB ADDRESS</td>
<td>10 001 [PN*]</td>
</tr>
<tr>
<td>Jump Indirect Sub To Address</td>
<td>JAS U</td>
<td>10 011 U</td>
</tr>
<tr>
<td>Return From Subroutine</td>
<td>RTS</td>
<td>10 111 000</td>
</tr>
<tr>
<td>Return From Interrupt and Enable Interrupt</td>
<td>RTE</td>
<td>10 110 001</td>
</tr>
<tr>
<td>Return From Interrupt</td>
<td>RTI</td>
<td>10 110 000</td>
</tr>
<tr>
<td>No Operation</td>
<td>NOP</td>
<td>01 011 111</td>
</tr>
<tr>
<td>Disable The Interrupt</td>
<td>DSI</td>
<td>10 111 111</td>
</tr>
<tr>
<td>Enable The Interrupt</td>
<td>ENI</td>
<td>00 101 111</td>
</tr>
</tbody>
</table>

* Double Byte Instructions

+ PN = Page no.

Kamran Firooz
October 21, 1974
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.
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 II).

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 20 (code for NOR). During the assembling 20 is overwritten by other codes; however, the locations not used will remain as 20. 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 interrupt
INB    * Increment the ACC in Binary
SLE    * Skip if ACC < 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 numerical values are taken as OCTAL except in BCD in 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 231

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 OP CODES:

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.

```
0001 + Nano PROCESSOR ASSEMBLER
0002 + EXAMPLE I
0003 +
0004 +
0005 +
0006 +
0007 + THIS PROGRAM ADDS THE CONTENTS OF REGISTER 5 TO THE
0008 + CONTENTS OF REGISTER 6 AND STORE THE RESULT IN REGISTER 6.
0009 +
0010 +
0011 LOOP LDA RS1 LOAD ACC FROM REG. 5
0012 DED DECREMENT IN DECIMAL
0013 SHN SKIP IF ACC 10
0014 JMP OUT END OF ROUTINE
0015 STA RS5 STORE THE ACC IN RS5
0016 LDA RS6 LOAD ACC FROM REG. 6
0017 INC INCREMENT IN DECIMAL
0018 STA RS6 STORE ACC AT REGISTER 6
0019 JMP LOOP
0020 + REPEAT THE DECREMENT AND INCREMENT ROUTINE
0021 OUT LDA RS
0022 INH
0023 STA RS6 REG HAS THE SUM
0024 RS OUT 5 DEFINE RS 5 AS TAL 5
0025 RS OUT 6
0026 END
0027 EOF
```
Load the assembler into the 2100 as follow:

:PR,NPA

After the program is loaded the computer will display:

YES

6

OBJ1

1

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).
<table>
<thead>
<tr>
<th>SYMBOL</th>
<th>ADDRESS OR VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0000$</td>
<td></td>
</tr>
<tr>
<td>$0013$</td>
<td></td>
</tr>
<tr>
<td>$0000$</td>
<td></td>
</tr>
<tr>
<td>$0000$</td>
<td></td>
</tr>
</tbody>
</table>

No of Errors for Pass 1 = 0
This program adds the contents of register 5 to the contents of register 6 and store the result in register 6.

LOAD ACC FROM REG. 5
DECREMENT IN DECIMAL
+SKIP IF ACC #0
(END OF ROUTINE)
STORE THE ACC IN R5
LOAD ACC FROM REG. 6
INCREMENT IN DECIMAL
STORE ACC AT REGISTER 6

REPEAT THE DECREMENT AND INCREMENT ROUTINE

R6 HAS THE SUM
DEFINE R5 AS OCTAL 5

NO 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 on the following pages.
0001  *       HANG PROCESSOR ASSEMBLER
0002  *       EXAMPLE 2
0003  *
0005  *       DISPLAY ROUTINE
0006  *
0007    STR R1,0  *CLEAR REGISTER 1
0008    JMP  STR R0,0  *CLEAR R0
0009    STR R0,0  *SKIP IF DC0 IS SET
0010    JMP DISP *DISPLAY MESSAGE A
0011    STR R0,0  *SKIP IF DC1 IS SET
0012    JMP DISP *DISPLAY MESSAGE B
0013    EIN *ENABLE THE INTERRUPT
0014    JMP READ *NEITHER FLAG IS SET
0015    DISP STR R0,40  *STORE 40 IN R0
0016    LDR R1  *LOAD ACC FROM R1
0017    SLT *SKIP IF ACC < 40
0018    EIN  *CLEAR THE POINTER
0019    JMP READ
0020    DISP STR R0,40
0021    LDR R1
0022    EIN *SET THE POINTER FOR B
0023    JMP READ
0024    JMP 257  *INTERCEPT ROUTINE
0025    STR R2
0026    CLR *CLEAR THE CONTENTS OF ACC DURING INTERRUPT ROUTINE
0027    LDR R1  *LOAD ACC FROM POINTER
0028    OTR DS0
0029    OTR DS0  *OUTPUT ACC TO ADDRESS LATCH (DEVICE SELECT 00)
0030    JMP 3
0031    BL OTR DS2,48  *OUTPUT A BLANK
0032    JMP DISP
0033    A OTR DS2,101  *OUTPUT "A" CODE
0034    JMP DISP
0035    D OTR DS2,104  *OUTPUT "D" CODE
0036    JMP DISP
0037    E OTR DS2,105  *OUTPUT "E" CODE
0038    JMP DISP
0039    I OTR DS2,111  *OUTPUT "I" CODE
0040    JMP DISP
0041    K OTR DS2,114  *OUTPUT "K" CODE
0042    JMP DISP
0043    L OTR DS2,115  *OUTPUT "L" CODE
0044    JMP DISP
0045    M OTR DS2,116  *OUTPUT "M" CODE
0046    JMP DISP
0047    N OTR DS2,117  *OUTPUT "N" CODE
0048    JMP DISP
0049    O OTR DS1,40  *OUTPUT A BLANK
0050    JMP DISP
0051    P OTR DS1,120  *OUTPUT "P" CODE
0052    JMP DISP
0053    R OTR DS1,122  *OUTPUT "R" CODE
0054    JMP DISP
0055    S OTR DS1,123  *OUTPUT "S" CODE
0056    JMP DISP
0057    V OTR DS1,131  *OUTPUT "V" CODE
0058    JMP DISP
0059    DISP OTR DS1,40  *OUTPUT A BLANK
0061 GBA DS3 *START THE DISPLAY
0062 INR
0063 INR #DOUBLE INCREMENT THE POINTER
0064 *
0065 *
0066 STA I #UPDATE THE POINTER
0067 LD A
0068 JMP #WITH ITS VALUE BEFORE THE INTERRUPT OCCURED
0069 MLM #ENABLE THE INTERRUPT
0070 RCL #RETURN FROM INTERRUPT
0071 CLRA #STK R1 -0 #RESET THE POINTER FOR A
0072 JMP B100
0073 SETS #STK R1,40 #RESET THE POINTER FOR B
0074 JMP N100
0075 ORA 1000 #MESSAGE A
0076 JMP BL #DISPLAY A BLANK
0077 JMP R #DISPLAY O
0078 JMP T #DISPLAY I
0079 JMP S #DISPLAY S
0080 JMP F #DISPLAY P
0081 JMP L #DISPLAY L
0082 JMP H #DISPLAY H
0083 JMP Y #DISPLAY Y
0084 JMP BL
0085 JMP T
0086 JMP S
0087 JMP BL
0088 JMP O
0089 JMP K
0090 JMP SETA #RESET THE POINTER
0091 JMP 1040 #MESSAGE B
0092 JMP E
0093 JMP R
0094 JMP K
0095 JMP Q
0096 JMP R
0097 JMP BL
0098 JMP T
0099 JMP N
0100 JMP BL
0101 JMP D
0102 JMP T
0103 JMP S
0104 JMP F
0105 JMP L
0106 JMP A
0107 JMP Y
0108 JMP SETB
0109 R0 OCT 0
0110 R1 OCT 1
0111 R2 OCT 2
0112 DS0 OCT 0
0113 DS1 OCT 1
0114 DS2 OCT 2
0115 DS3 OCT 3
0116 FLAGS OCT 0
0117 FLAG1 OCT 1
0118 END
<table>
<thead>
<tr>
<th>SYMBOL</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>RLD0</td>
<td>0000</td>
</tr>
<tr>
<td>RLDH</td>
<td>0400</td>
</tr>
<tr>
<td>RLDL</td>
<td>0030</td>
</tr>
<tr>
<td>RLD</td>
<td>0402</td>
</tr>
<tr>
<td>R</td>
<td>0413</td>
</tr>
<tr>
<td>R2</td>
<td>0417</td>
</tr>
<tr>
<td>R3</td>
<td>0423</td>
</tr>
<tr>
<td>R4</td>
<td>0427</td>
</tr>
<tr>
<td>R5</td>
<td>0433</td>
</tr>
<tr>
<td>R6</td>
<td>0437</td>
</tr>
<tr>
<td>R7</td>
<td>0443</td>
</tr>
<tr>
<td>R8</td>
<td>0447</td>
</tr>
<tr>
<td>R9</td>
<td>0453</td>
</tr>
<tr>
<td>R10</td>
<td>0457</td>
</tr>
<tr>
<td>R11</td>
<td>0463</td>
</tr>
<tr>
<td>R12</td>
<td>0465</td>
</tr>
<tr>
<td>R13</td>
<td>0474</td>
</tr>
<tr>
<td>R14</td>
<td>0476</td>
</tr>
<tr>
<td>R15</td>
<td>0502</td>
</tr>
<tr>
<td>R16</td>
<td>0508</td>
</tr>
<tr>
<td>R17</td>
<td>050C</td>
</tr>
<tr>
<td>R18</td>
<td>0002</td>
</tr>
<tr>
<td>R19</td>
<td>0003</td>
</tr>
<tr>
<td>R20</td>
<td>0001</td>
</tr>
<tr>
<td>R21</td>
<td>0002</td>
</tr>
<tr>
<td>R22</td>
<td>0003</td>
</tr>
<tr>
<td>R23</td>
<td>0001</td>
</tr>
<tr>
<td>FLAG0</td>
<td>0000</td>
</tr>
<tr>
<td>FLAG1</td>
<td>0001</td>
</tr>
</tbody>
</table>

**NO OF ERRORS FOR PASS 1 = 0**
DISPLAY ROUTINE

3 0036 021 STR R1,0  #CLEAR REGISTER 1
3 0061 500  #CLEAR R0
9 0002 320 READ STR R0,0  #CLEAR QUEUE
9 0003 000  #CLEAR QUEUE
10 0004 030 SFS FLAGA  #SKIP IF DCO IS SET
11 0005 209 JMP DISP A  #DISPLAY MESSAGE A
12 0006 015  #DISPLAY MESSAGE A
12 0007 031 SFS FLAG1  #SKIP IF DC1 IS SET
13 0010 200 JMP DISP B  #DISPLAY MESSAGE B
13 0011 026  #DISPLAY MESSAGE B
14 0012 057 ENI  #DISABLE THE INTERRUPT
14 0013 200 JMP READ  #NEITHER FLAG IS SET
15 0014 002  #NEITHER FLAG IS SET
16 0015 320 DISPA STR R0,40  #LOAD ACC FROM R1
16 0016 040  #LOAD ACC FROM R1
17 0017 141 LDA R1  #LOAD ACC FROM R1
18 0020 011 SLT  #LOAD ACC FROM R1
19 0021 321 STR R1,0  #LOAD ACC FROM R1
19 0022 000  #LOAD ACC FROM R1
20 0023 057 ENI  #LOAD ACC FROM R1
20 0024 200 JMP READ  #LOAD ACC FROM R1
20 0025 002  #LOAD ACC FROM R1
22 0026 320 DISPB STR R0,40  #LOAD ACC FROM R1
22 0027 040  #LOAD ACC FROM R1
23 0030 141 LDA R1  #LOAD ACC FROM R1
24 0031 015 SGE  #LOAD ACC FROM R1
25 0032 321 STR R1,40  #LOAD ACC FROM R1
25 0033 040  #LOAD ACC FROM R1
26 0034 057 ENI  #LOAD ACC FROM R1
26 0035 200 JMP READ  #LOAD ACC FROM R1
26 0036 002  #LOAD ACC FROM R1
28 0037 162 ORG 377  #INTERRUPT ROUTINE
29 0077 162 STA R2  #INTERRUPT ROUTINE
#R2 HOLDS THE CONTENTS OF ACC DURING INTERRUPT ROUTINE
31 0400 141 LDA R1  #LOAD ACC FROM POINTER
32 0401 129 OTA D50  #LOAD ACC FROM POINTER
33 #OUTPUT ACC TO ADDRESS LATCH (DEVICE SELECT 0)
34 0402 222 JAI 2  #OUTPUT ACC TO ADDRESS LATCH
35 0403 302 SL OTR DS2,40  #OUTPUT ACC TO ADDRESS LATCH
36 0404 040 JMP DISP  #OUTPUT ACC TO ADDRESS LATCH
36 0405 201 JMP DISP  #OUTPUT ACC TO ADDRESS LATCH
37 0406 065  #OUTPUT ACC TO ADDRESS LATCH
37 0407 302 A OTR DS2,101  #OUTPUT "A" CODE
37 0410 101  #OUTPUT "A" CODE
38 0411 200 JMP DISP  #OUTPUT "A" CODE
38 0412 065  #OUTPUT "A" CODE
39 0413 302 D OTR DS2,104  #OUTPUT "D" CODE
39 0414 104  #OUTPUT "D" CODE
40 0415 201 JMP DISP  #OUTPUT "D" CODE
40 0416 065  #OUTPUT "D" CODE
<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Instruction</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>41</td>
<td>0417</td>
<td>OTR D52:185</td>
<td>*OUTPUT &quot;E&quot; CODE</td>
</tr>
<tr>
<td>41</td>
<td>0429</td>
<td>JMP DISP</td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>0421</td>
<td>JMP DISP</td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>0422</td>
<td>JMP DISP</td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>0423</td>
<td>OTR D52:111</td>
<td>*OUTPUT &quot;I&quot; CODE</td>
</tr>
<tr>
<td>43</td>
<td>0424</td>
<td>JMP DISP</td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>0425</td>
<td>JMP DISP</td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>0427</td>
<td>OTR D52:114</td>
<td>*OUTPUT &quot;K&quot; CODE</td>
</tr>
<tr>
<td>45</td>
<td>0428</td>
<td>JMP DISP</td>
<td></td>
</tr>
<tr>
<td>46</td>
<td>0430</td>
<td>JMP DISP</td>
<td></td>
</tr>
<tr>
<td>46</td>
<td>0431</td>
<td>JMP DISP</td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>0432</td>
<td>OTR D52:115</td>
<td>*OUTPUT &quot;L&quot; CODE</td>
</tr>
<tr>
<td>47</td>
<td>0433</td>
<td>JMP DISP</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>0434</td>
<td>JMP DISP</td>
<td></td>
</tr>
<tr>
<td>49</td>
<td>0435</td>
<td>JMP DISP</td>
<td></td>
</tr>
<tr>
<td>51</td>
<td>0440</td>
<td>OTR D52:116</td>
<td>*OUTPUT &quot;H&quot; CODE</td>
</tr>
<tr>
<td>51</td>
<td>0443</td>
<td>JMP DISP</td>
<td></td>
</tr>
<tr>
<td>52</td>
<td>0441</td>
<td>JMP DISP</td>
<td></td>
</tr>
<tr>
<td>52</td>
<td>0442</td>
<td>JMP DISP</td>
<td></td>
</tr>
<tr>
<td>53</td>
<td>0447</td>
<td>OTR D52:117</td>
<td>*OUTPUT &quot;G&quot; CODE</td>
</tr>
<tr>
<td>53</td>
<td>0448</td>
<td>JMP DISP</td>
<td></td>
</tr>
<tr>
<td>54</td>
<td>0449</td>
<td>JMP DISP</td>
<td></td>
</tr>
<tr>
<td>55</td>
<td>0450</td>
<td>OTR D52:118</td>
<td>*OUTPUT &quot;F&quot; CODE</td>
</tr>
<tr>
<td>55</td>
<td>0451</td>
<td>JMP DISP</td>
<td></td>
</tr>
<tr>
<td>56</td>
<td>0452</td>
<td>JMP DISP</td>
<td></td>
</tr>
<tr>
<td>56</td>
<td>0453</td>
<td>JMP DISP</td>
<td></td>
</tr>
<tr>
<td>57</td>
<td>0457</td>
<td>OTR D52:119</td>
<td>*OUTPUT &quot;E&quot; CODE</td>
</tr>
<tr>
<td>57</td>
<td>0454</td>
<td>JMP DISP</td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>0455</td>
<td>JMP DISP</td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>0456</td>
<td>JMP DISP</td>
<td></td>
</tr>
<tr>
<td>59</td>
<td>0460</td>
<td>OTR D52:120</td>
<td>*OUTPUT &quot;D&quot; CODE</td>
</tr>
<tr>
<td>59</td>
<td>0461</td>
<td>JMP DISP</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>0462</td>
<td>JMP DISP</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>0463</td>
<td>JMP DISP</td>
<td></td>
</tr>
<tr>
<td>61</td>
<td>0467</td>
<td>OTR D52:121</td>
<td>*OUTPUT &quot;C&quot; CODE</td>
</tr>
<tr>
<td>61</td>
<td>0468</td>
<td>JMP DISP</td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>0469</td>
<td>JMP DISP</td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>0470</td>
<td>JMP DISP</td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>0471</td>
<td>OTR D53</td>
<td>*START THE DISPLAY</td>
</tr>
<tr>
<td>63</td>
<td>0472</td>
<td>JMP DISP</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>0473</td>
<td>JMP DISP</td>
<td></td>
</tr>
<tr>
<td>65</td>
<td>0474</td>
<td>JMP DISP</td>
<td></td>
</tr>
<tr>
<td>66</td>
<td>0475</td>
<td>OTR D53</td>
<td>*UPDATE THE POINTER</td>
</tr>
<tr>
<td>67</td>
<td>0476</td>
<td>OTR D53</td>
<td>*DOUBLE INCREMENT THE POINTE</td>
</tr>
<tr>
<td>68</td>
<td>0477</td>
<td>OTR D53</td>
<td>*START THE DISPLAY</td>
</tr>
<tr>
<td>69</td>
<td>0478</td>
<td>OTR D53</td>
<td>*RETURN FROM INTERRUPT</td>
</tr>
<tr>
<td>70</td>
<td>0479</td>
<td>OTR D53</td>
<td>*RESET POINTER FOR A</td>
</tr>
<tr>
<td>71</td>
<td>047A</td>
<td>OTR D53</td>
<td></td>
</tr>
<tr>
<td>72</td>
<td>047B</td>
<td>OTR D53</td>
<td></td>
</tr>
<tr>
<td>73</td>
<td>047C</td>
<td>OTR D53</td>
<td></td>
</tr>
<tr>
<td>74</td>
<td>047D</td>
<td>OTR D53</td>
<td></td>
</tr>
<tr>
<td>103</td>
<td>104</td>
<td>105</td>
<td>106</td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**SET BST**

**MESSAGE A**

**MESSAGE B**

**DISPLAY**

**DISPLAY**

**DISPLAY**

**DISPLAY**

**DISPLAY**
104 1070 201 IMP P
104 1071 047 JMP L
105 1072 001 JMP A
105 1073 033
106 1074 201 JMP Y
107 1075 066
107 1076 201 JMP SETB
108 1100 201
108 1101 192
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.
DATA FILE  FOR  THE PROGRAM GIVEN ON EXAMPLE 2

JMP 510
JMP 467
JMP 417
JMP 417
JMP 584
END

NO OF ERRORS FOR PASS 2 = 0
ASSEMBLY LANGUAGE INSTRUCTIONS

ACCUMULATOR INSTRUCTIONS:

Skip on Bit N=1  SBS N  00 0100  N
Skip on Bit N=0  SBZ N  00 1100  N
Set Bit N        SBN N  00 1000  N
Clear Bit N      CBZ N  10 1000  N
Increment ACC (Binary) INB  00 0000 000
Increment ACC (Decimal) IND  00 0000 010
Decrement ACC (Binary) DEB  00 0000 001
Decrement ACC (Decimal) DED  11 0000 011
Clear ACC        CLA  00 0000 100
Complement ACC   CMA  00 0000 101
Left Shift ACC   LSA  00 0000 110
Right Shift ACC  RSA  00 0000 111
Load ACC with ROM DATA *  LDR DATA  11 0011 111
Skip on E=1      SES  00 0111 111
Skip on E=0      SEZ  00 1111 111
Set E            STE  10 1100 100
Clear E          CLE  10 1100 101

REGISTER AND I/O INSTRUCTIONS:

Load ACC From Register R  LDA R  01 1000
Load ACC Indexed         LDI Z  11 1000
Store ACC At Register R  STA R  01 1100
Store ACC Indexed        STI Z  11 1100
Input To ACC From DS    INA DS  01 00 00 DS
Output ACC To DS         OTA DS  01 01 00 DS
Store ROM DATA At Register R* STR R,DATA  11 01 00
Output ROM DATA To DS*   CTR DS,DATA  11 00 00 DS
Set Control K  STC K
Clear Control K  CLC K
Skip On Flag J=1  SFS J
Skip On Flag J=Ø  SFZ 'J

COMPARATOR INSTRUCTIONS:

Skip On ACC >RØ  SGT  Ø
Skip On ACC <RØ  SLT  Ø
Skip on ACC = RØ  SEQ  Ø
Skip On ACC >RØ  SGE  Ø
Skip On ACC <RØ  SLE  Ø
Skip On ACC #RØ  SNE  Ø
Skip On ACC = Ø  SAZ  Ø
Skip On ACC # Ø  SAN  Ø

PROGRAM CONTROL INSTRUCTIONS:

Jump To Address  JMP ADDRESS
Jump Indirect To Address  JAI U
Jump Sub To Address*  JSB ADDRESS

Jump Indirect Sub To Address  JAS U
Return From Subroutine  RTS
Return From Interrupt and Enable Interrupt
Return From Interrupt  RTI  Ø
No Operation.  NOP  Ø
Dissable The Interrupt  DSI
Enable The Interrupt  ENI

* Double Byte Instructions

* PN = Page no.
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 NCP (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
<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Description</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0010</td>
<td>OPC 510</td>
<td>OUTPUT CODE FOR &quot;C&quot; TO 182</td>
</tr>
<tr>
<td>9</td>
<td>0510</td>
<td>OMR 210</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0510</td>
<td>JMP 467</td>
<td>JUMP TO DISP</td>
</tr>
<tr>
<td>11</td>
<td>0510</td>
<td>UTR 25126</td>
<td>OUTPUT CODE FOR &quot;V&quot;</td>
</tr>
<tr>
<td>12</td>
<td>0510</td>
<td>JMP 467</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>0510</td>
<td>URG 1060</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>0510</td>
<td>JMP 493</td>
<td>JMP TO BL</td>
</tr>
<tr>
<td>15</td>
<td>0510</td>
<td>JMP 413</td>
<td>JMP TO D</td>
</tr>
<tr>
<td>16</td>
<td>0510</td>
<td>JMP 417</td>
<td>JMP TO E</td>
</tr>
<tr>
<td>17</td>
<td>0510</td>
<td>JMP V</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>0510</td>
<td>JMP 423</td>
<td>JMP TO 1</td>
</tr>
<tr>
<td>19</td>
<td>0510</td>
<td>JMP 0</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>0510</td>
<td>JMP 417</td>
<td>JMP TO E</td>
</tr>
<tr>
<td>21</td>
<td>0510</td>
<td>JMP 004</td>
<td>JMP TO SETB</td>
</tr>
<tr>
<td>22</td>
<td>0510</td>
<td>END</td>
<td></td>
</tr>
</tbody>
</table>

NO OF ERRORS FOR PASS 2 = 0
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:

:DU,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
10 FILE -- 2100 TO 9830 ---
20 REM 2100 HANO AS SEMBLER DECEMBER 1974
30 REM
40 THE PROGRAM READS AN OBJECT FILE PRODUCED BY THE 2100 HANO PROCESSOR
50 THEN EDITS AND STORES THE FILE ON A CASSETTE TAPE.
60 THE EDITED FILES ARE COMPATIBLE TO THE FILES PRODUCED BY THE 9830 ASSEMBLER
70 AND SIMPSON; THEY CAN BE LOADED INTO A ROM-RAM SIMULATOR USING THE LOADER
80 FILE PROGRAM.
90 DIM 0 TO 256
100 K=0
110 DISP "ENTER CODE NUMBER:"
120 INPUT V
130 IF V=99 THEN 300
140 NEXT I
150 GOTO 200
160 IF K=256 OR K<326
170 =C
180 FOR J=1 TO 4
190 FOR I=1 TO 256
200 P=K
210 K=K+1
220 M=M+15
230 IF P=256 THEN 390
240 P=I
250 K=I+1
260 NEXT J
270 NEXT I
280 STORE DATA LIST
290 END
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,8,8, 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 9830 TO 2100

120 REM

130 REM THE PROGRAM PUNCHES A SOURCE FILE GENERATED BY THE NANO PROCESSOR

140 REM PUT ON PAPER TAPE USING A PAPER TAPE PUNCH.

150 REM

160 REM THEN PAPER TAPE CAN THEN BE STORED ON A DISK OF A 2100 COMPUTER.

170 REM THE STORED PROGRAM CAN BE ASSEMBLED USING THE 2100 NANO PROCESSOR

180 REM ASSEMBLER (NPA).

200 REM

210 DIM ALL(100,151)

220 DISK "SELECT CODE NO."

230 INPUT "FILE NO."

240 OPEN DATA 138

250 GOSUB 150

260 FOR I=1 TO 140

270 TRANSFER 9 I+11 TO H$

280 H$=PO$UCASE$"$

290 IF H$=-1 THEN 370

300 H$=ST$

310 GOSUB 116

320 H$=""

330 H$=UP$UCASE$"$+$

340 GOTO 330

350 GOTO 330

360 GOTO 330

370 P=P+$UCASE$"$+

380 IF P=0 THEN 410

390 H$=H$

400 GOSUB 400

410 NEXT I

420 GOSUB 330

430 END

440 H$=P-107/(P/23+2)

450 IF H$<0 THEN 460

460 WRITE "1$"+H$+"$1$+P-11"

470 IF POS$HE:"ED"$:0 THEN 420

480 IF H$<0 THEN 520

490 IF P=1 THEN 510

500 RETURN

510 GOTO 220

520 WRITE 1#$"$HE$YTE$TR"

530 NEXT I

540 CLEAR

550 RETURN

560 END
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, i.e. 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 5061-0769

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.
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 COOLER 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^2L 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 TpCH (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 TpCH at the same time.

G. The equations for interrelating these parameters are:
   1. \( ClkT \geq r_t(data) + T_{IP} + T_{DV \ max} \)
   2. \( ClkT \geq \text{min. spec.} \)
   3. \( ClkT \geq T_{PA2} + T_{AA} + T_{IP} \)
      Note: \( T_{AA} \) include \( r_t(data) \) to 4.0 Volts
   4. \( ClkT \geq T_{pCH \ max.} + T_{EA} + T_{IP} \)

   Reasonable values to use for \( T_{DV \ max} \). (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 0.5 Volts using 10K ohm resistors. Design constants for this method are:
"A" Chips: \( r.t. \text{(data)} = 2.4 \ (C_D + 7)\text{ns} \)

"B" Chips: \( r.t. \text{(data)} = 3.6 \ (C_D + 7)\text{ns} \)

"C" Chips: \( r.t. \text{(data)} = 3.9 \ (C_D + 7)\text{ns} \) \( (C_D \text{ IN pF}) \)

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

"A" Chips: \( r.t. \text{(data)} = 2.2 \ (C_D + 7)\text{ns} \)

"B" Chips: \( r.t. \text{(data)} = 3.1 \ (C_D + 7)\text{ns} \)

"C" Chips: \( r.t. \text{(data)} = 3.4 \ (C_D + 7)\text{ns} \) \( (C_D \text{ in pF}) \).