United States Patent [19]

Koopman, Jr. et al.

[11] **Patent Number: 5,053,952**

[45] **Date of Patent: Oct. 1, 1991**

[54] **STACK-MEMORY-BASED WRITABLE INSTRUCTION SET COMPUTER HAVING A SINGLE DATA BUS**

[75] Inventors: **Philip J. Koopman, Jr.**, N. Kingston, R.I.; **Glen B. Haydon**, La Honda, Calif.

[73] Assignee: **WISC Technologies, Inc.**, La Honda, Calif.

[21] Appl. No.: **58,737**

[22] Filed: **Jun. 5, 1987**

[51] **Int. Cl.**[5] **G06F 9/42**; G06F 9/22; G06F 13/40

[52] **U.S. Cl.** **364/200**; 364/244.3; 364/262.7; 364/240; 364/247.7; 364/240.1

[58] **Field of Search** ... 364/200 MS File, 900 MS File

[56] **References Cited**

U.S. PATENT DOCUMENTS

3,215,987	11/1965	Terzian	364/200
3,629,857	12/1971	Faber .	
3,757,306	9/1978	Boone .	
3,771,141	11/1973	Culler	364/200
3,786,432	1/1974	Woods .	
4,045,781	8/1977	Levy et al.	364/200
4,204,252	5/1980	Hitz et al.	364/200
4,210,960	7/1980	Borgerson et al.	364/200
4,415,969	11/1983	Bayliss et al.	364/200
4,447,875	5/1984	Bolton et al.	364/200
4,491,912	1/1985	Kainaga et al.	364/200
4,546,431	10/1985	Horvath	364/200
4,615,003	9/1986	Logsdon et al.	364/200
4,618,925	10/1986	Bratt et al.	364/200
4,654,780	3/1987	Logsdon et al.	364/200
4,674,032	6/1987	Michaelson .	
4,719,565	1/1988	Moller .	
4,791,551	12/1988	Garde	364/200
4,835,738	5/1989	Niehaus et al. .	

OTHER PUBLICATIONS

"Stack-Oriented WISC Machine", WISC Technologies, La Honda, Ca., 94020, 2 pages.

BYTE 6/86, Microcoded IBM PC Board, Mtn. Vw. Press Advertisement, Haydon, MVP Microcoded CPU/16, Mountain View Press, 4 pages.

Koopman & Haydon, MVP Microcoded CPU/16 Architecture, Mountain View Press, 4 pages.

Koopman, Microcoded Versus Hard-Wired Control, BYTE, Jan. 1987, pp. 235-242.

Haydon, The Multi-Dimensions of Forth, *Forth Dimensions*, vol. 8, No. 3, pp. 32-34, Sep./Oct., 1986.

Rust, ACTION Processor Forth Right, *Rochester Forth Standards Conference*, pp. 309-315, 3/8/79.

Wada, Software and System Evaluation of a Forth Machine System, *Systems, Computers, Controls*, vol. 13, No. 2, pp. 19-28.

Wada, System Design and hardware Structure of a Forth Machine System, *Systems, Computers, Controls*, vol. 13, No. 2, 1982, pp. 11-18.

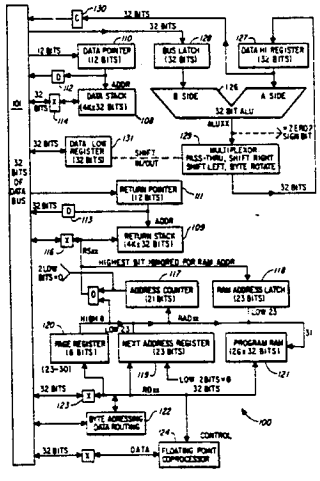
(List continued on next page.)

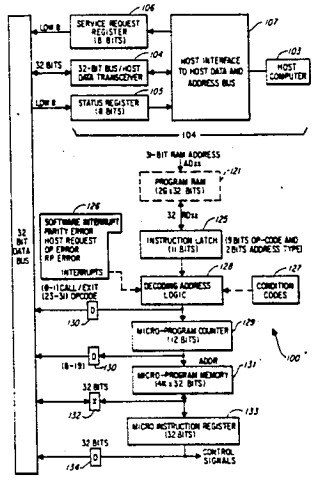
Primary Examiner—Gareth D. Shaw
Assistant Examiner—P. V. Kulik
Attorney, Agent, or Firm—Edward B. Anderson

[57] **ABSTRACT**

A computer is provided as an add-on processor for attachment to a host computer. Included are a single data bus, a 32-bit arithmetic logic unit, a data stack, a return stack, a main program memory, data registers, program memory addressing logic, micro-program memory, and a micro-instruction register. Each machine instruction contains an opcode as well as a next address field and subroutine call/return or unconditional branching information. The return address stack, memory addressing logic, program memory, and microcoded control logic are separated from the data bus to provide simultaneous data operations with program control flow processing and instruction fetching and decoding. Subroutine calls, subroutine returns, and unconditional branches are processed with a zero execution time cost. Program memory may be written as either bytes or full words without read/modify/write operations. The top of data stack ALU register may be exchanged with other registers in two clock cycles instead of the normal three cycles. MVP-FORTH is used for programming a microcode assembler, a cross-compiler, a set of diagnostic programs, and microcode.

9 Claims, 90 Drawing Sheets





OTHER PUBLICATIONS

Norton & Abraham, Adaptive Interpretation as a Means of Exploiting Complex Instruction Sets, *IEEE International Symposium on Computer Architecture,* pp. 277–282, 1983.

Sequin et al., Design and Implementation of RISC I, *ELSI Architecture,* pp. 276–298, 1982.

Patterson et al., RISC Assessment: A High-Level Language Experiment, *Symposium on Computer Architecture,* No. 9, pp. 3–8, 1982.

Folger et al., Computer Architectures–Designing for Speed, *Intellectual Leverage for the Information Society,* Spring 83, pp. 25–31.

Larus, A Comparison of Microcode, Assembly Code & High-Level Langauges on the VAX-11 & RISC I, *Computer Architecture News,* vol. 10, No. 5, pp. 10–15.

Castan et al., μ3L: An HLL-RISC Processor for Parallel Execution of FP Language Programs, *Symposium on Core Computer Architecture,* #9, pp. 239–247, 1982.

Koopman, The WISC Concept, *BYTE,* pp. 187–193, Apr. 1987.

Haydon, A Unification of Software and Hardware; A New Tool for Human Thought, 1987 Rochester, Forth Conference, pp. 25–28.

Koopman, Writable Instruction Set, Stack Oriented Computers: The WISC Concept, 1987 Rochester Forth Conference, pp. 29–51.

Thurber et al., "A Systematic Approach to the Design of Digital Bussing Structures", Fall Joint Computer Conference, 1972, pp. 719–740.

Philip J. Koopman, Jr., *Stack Computers-The New Wave,* 1989.

Ditzel and McLellan, "Branch Folding in the CRISP Microprocessor: Reducing Branch Delay to Zero", *ACM,* 6/2/87, pp. 2–9.

Ditzel, McLellan and Berenbaum, "The Hardware Architecture of the CRISP Microprocessor", *ACM,* 6/2/87, pp. 309–319.

Kaneda, Wada and Maekawa, "High-Speed Execution of Forth and Pascal Programs on a High-Level Language Machine", 1983, pp. 259–266.

Grewe and Dixon, "A Forth Machine for the S–100 System", The Journal of Forth Application and Research, vol. 2, No. 1, 1984, pp. 23–32.

A. C. D. Haley, "The KDF.9 Computer System", *AFIPS Conference Proceedings,* vol. 22, 1962 Fall Joint Computer Conference, pp. 108–120.

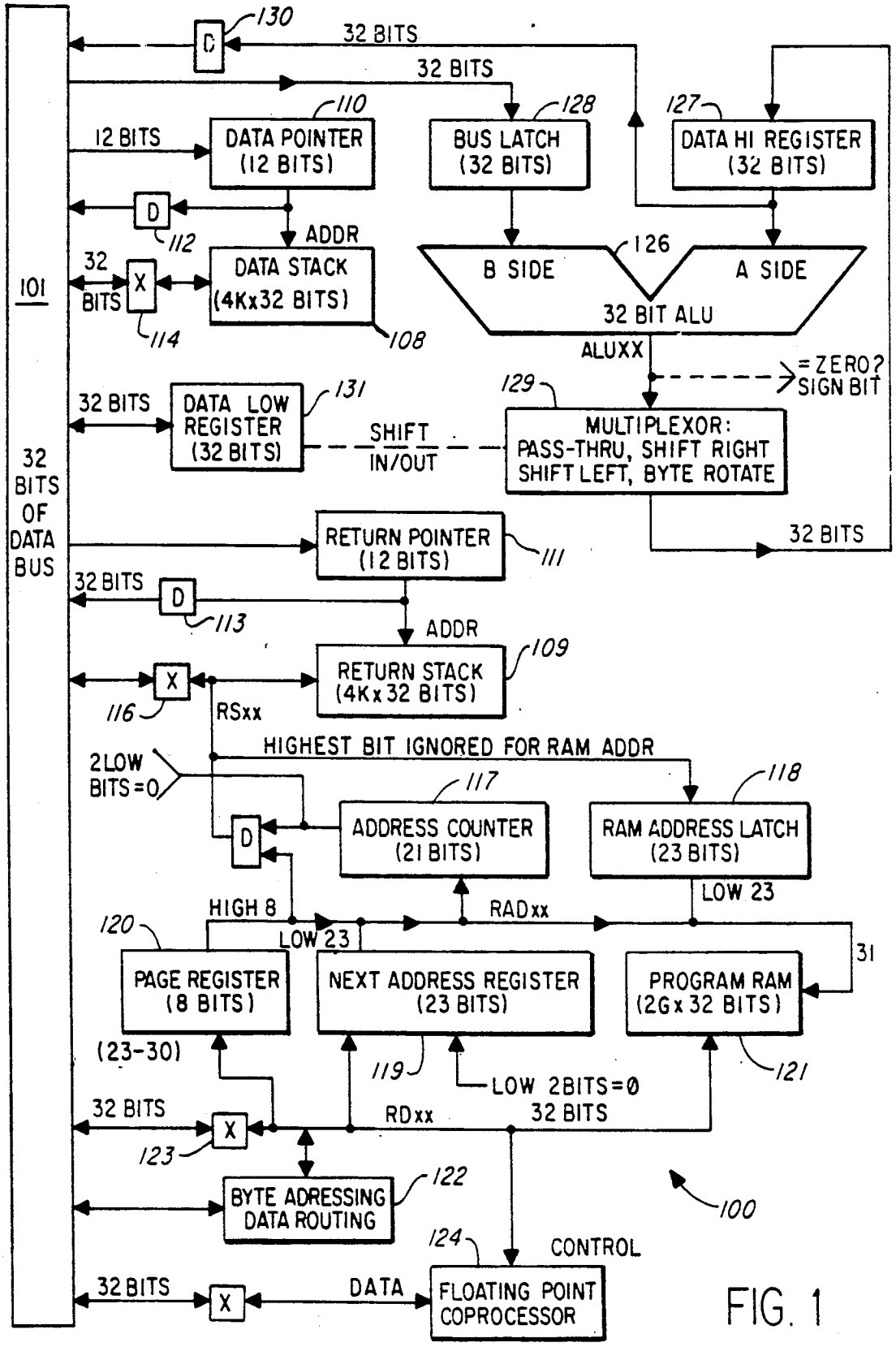

130
32 BITS
32 BITS
110
128
127
12 BITS
DATA POINTER (12 BITS)
BUS LATCH (32 BITS)
DATA HI REGISTER (32 BITS)
D
112
ADDR
126
101
32 BITS
X
DATA STACK (4Kx32 BITS)
B SIDE
A SIDE
32 BIT ALU
114
108
ALUXX
=ZERO?
SIGN BIT
32 BITS
DATA LOW REGISTER (32 BITS)
131
129
SHIFT IN/OUT
MULTIPLEXOR: PASS-THRU, SHIFT RIGHT SHIFT LEFT, BYTE ROTATE
32 BITS OF DATA BUS
RETURN POINTER (12 BITS)
111
32 BITS
113
ADDR
109
RETURN STACK (4Kx32 BITS)
116
RSxx
HIGHEST BIT IGNORED FOR RAM ADDR
2LOW BITS=0
117
118
ADDRESS COUNTER (21 BITS)
RAM ADDRESS LATCH (23 BITS)
LOW 23
HIGH 8
RADxx
120
LOW 23
31
PAGE REGISTER (8 BITS)
NEXT ADDRESS REGISTER (23 BITS)
PROGRAM RAM (2Gx32 BITS)
(23-30)
119
LOW 2BITS=0
121
32 BITS
RDxx
32 BITS
123
BYTE ADRESSING DATA ROUTING
122
100
124
CONTROL
32 BITS
DATA
FLOATING POINT COPROCESSOR

FIG. 1

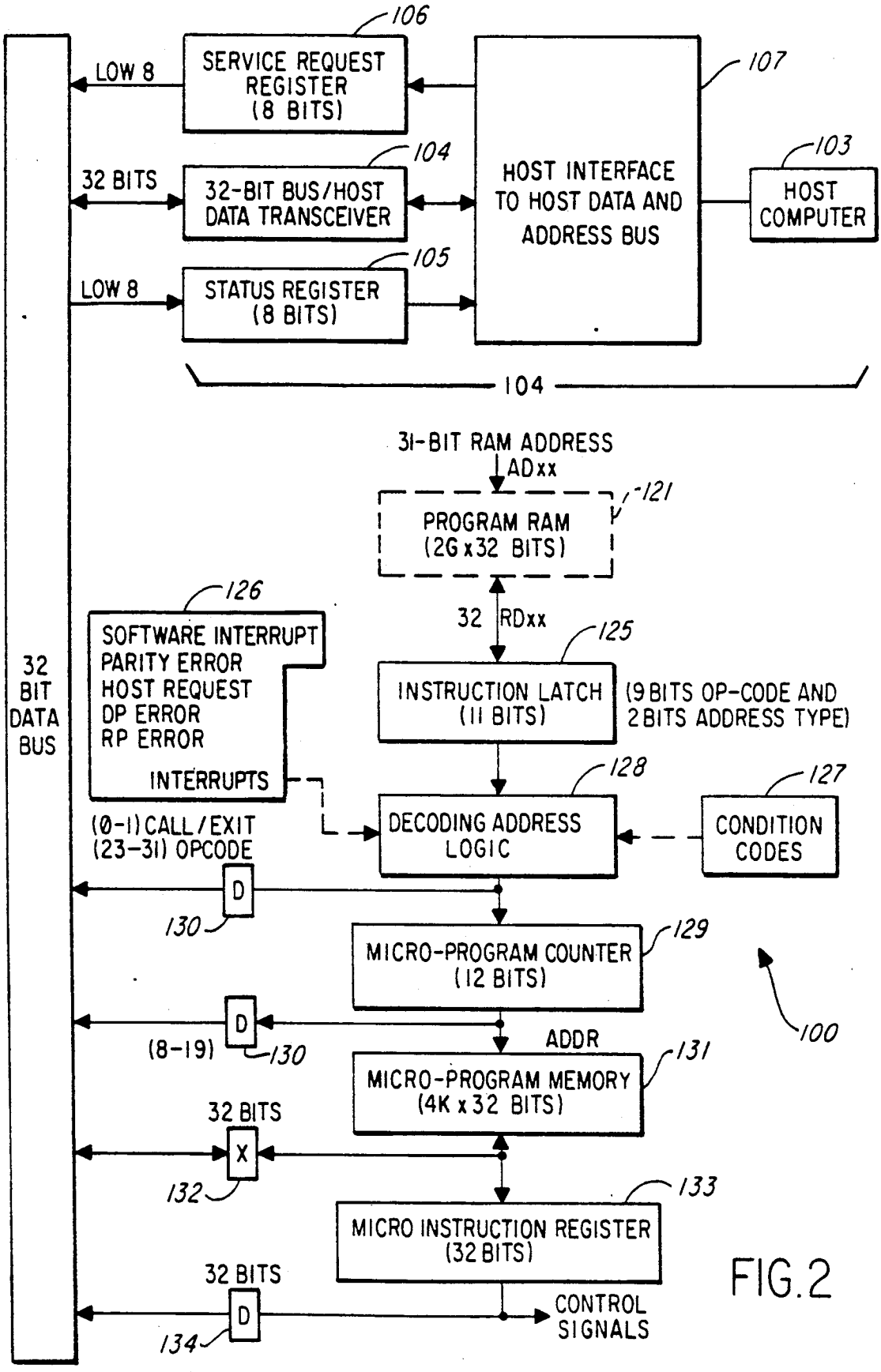
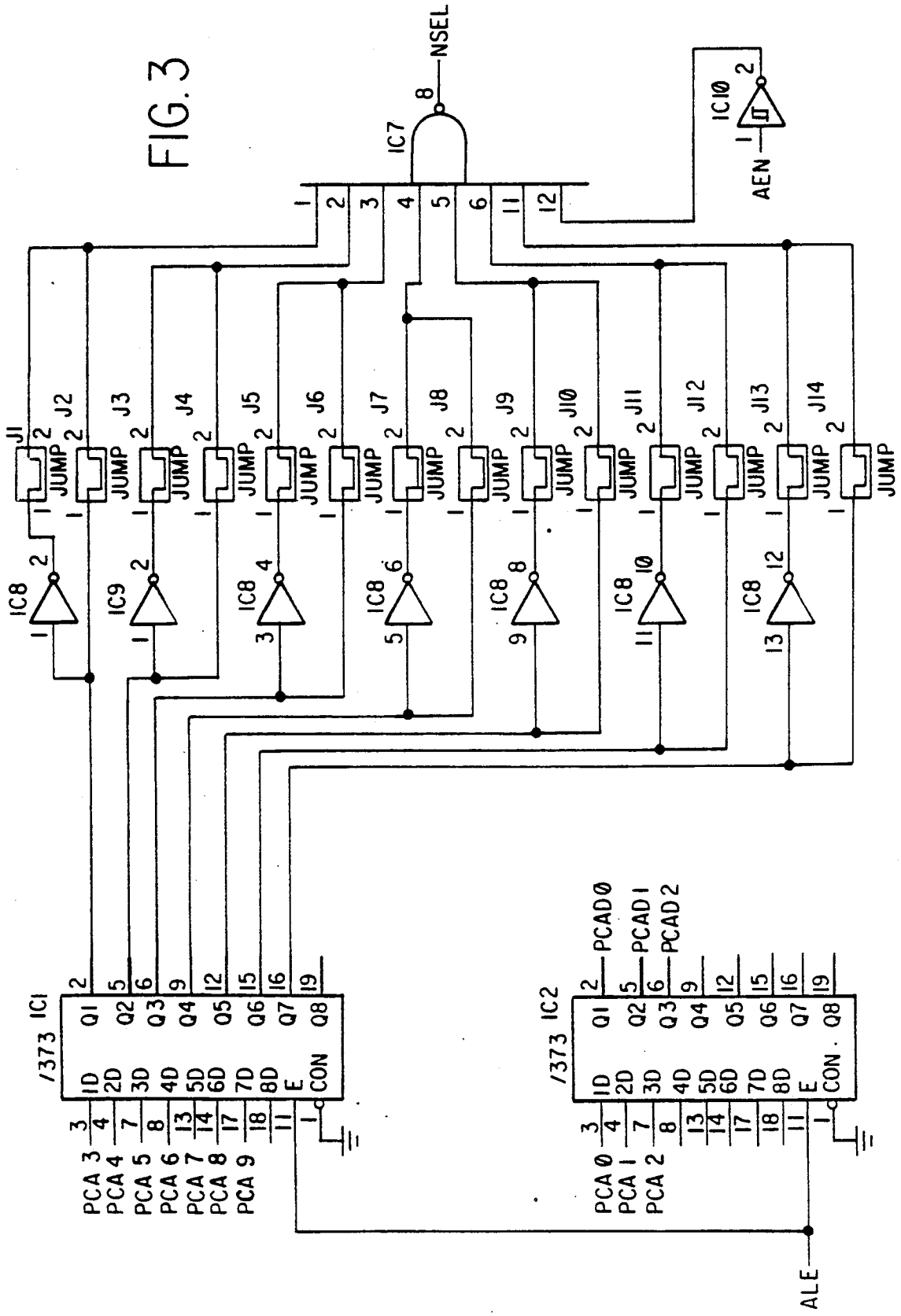

106
SERVICE REQUEST REGISTER (8 BITS)
LOW 8
107
HOST INTERFACE TO HOST DATA AND ADDRESS BUS
103
HOST COMPUTER
104
32 BITS
32-BIT BUS/HOST DATA TRANSCEIVER
105
STATUS REGISTER (8 BITS)
LOW 8
104
31-BIT RAM ADDRESS
ADxx
121
PROGRAM RAM (2G x 32 BITS)
32 RDxx
126
SOFTWARE INTERRUPT
PARITY ERROR
HOST REQUEST
DP ERROR
RP ERROR
INTERRUPTS
125
INSTRUCTION LATCH (11 BITS)
(9 BITS OP-CODE AND 2 BITS ADDRESS TYPE)
32 BIT DATA BUS
128
DECODING ADDRESS LOGIC
127
CONDITION CODES
(0-1) CALL/EXIT
(23-31) OPCODE
D
130
129
MICRO-PROGRAM COUNTER (12 BITS)
100
D
(8-19)
130
ADDR
131
MICRO-PROGRAM MEMORY (4K x 32 BITS)
32 BITS
X
132
133
MICRO INSTRUCTION REGISTER (32 BITS)
32 BITS
D
134
CONTROL SIGNALS

FIG. 2

FIG. 3


NSEL
IC7
IC10
AEN
J1
J2
J3
J4
J5
J6
J7
J8
J9
J10
J11
J12
J13
J14
JUMP
IC8
IC9
IC1
IC2
/373
Q1
Q2
Q3
Q4
Q5
Q6
Q7
Q8
1D
2D
3D
4D
5D
6D
7D
8D
E
CON
PCAD0
PCAD1
PCAD2
PCA 3
PCA 4
PCA 5
PCA 6
PCA 7
PCA 8
PCA 9
PCA 0
PCA 1
PCA 2
ALE

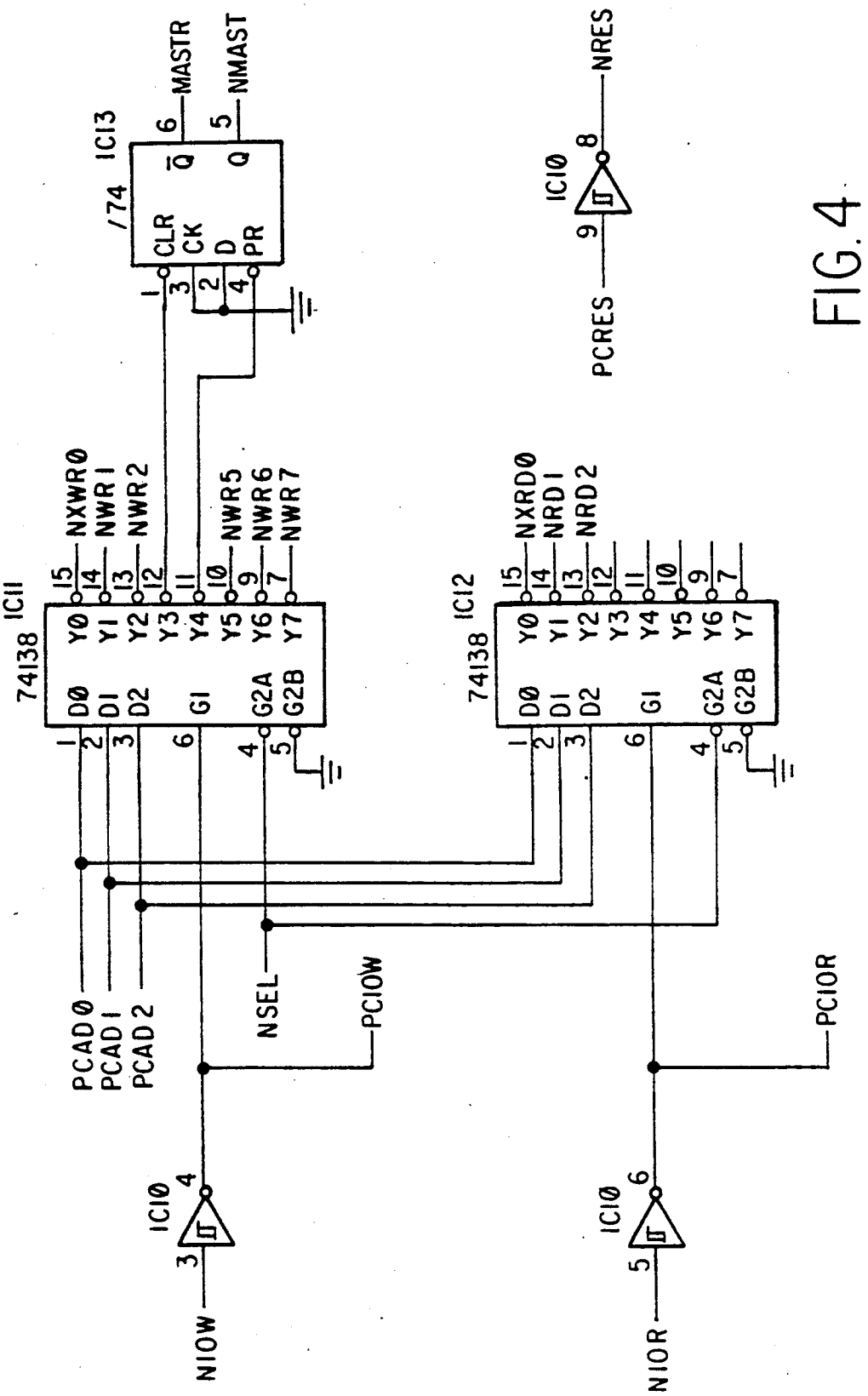

IC13
/74
CLR
CK
D
PR
Q
MASTR
NMAST
IC10
NRES
PCRES
IC11
74138
NXWR0
NWR1
NWR2
NWR5
NWR6
NWR7
IC12
NXRD0
NRD1
NRD2
PCAD0
PCAD1
PCAD2
NSEL
PCIOW
PCIOR
NIOW
NIOR

FIG. 4

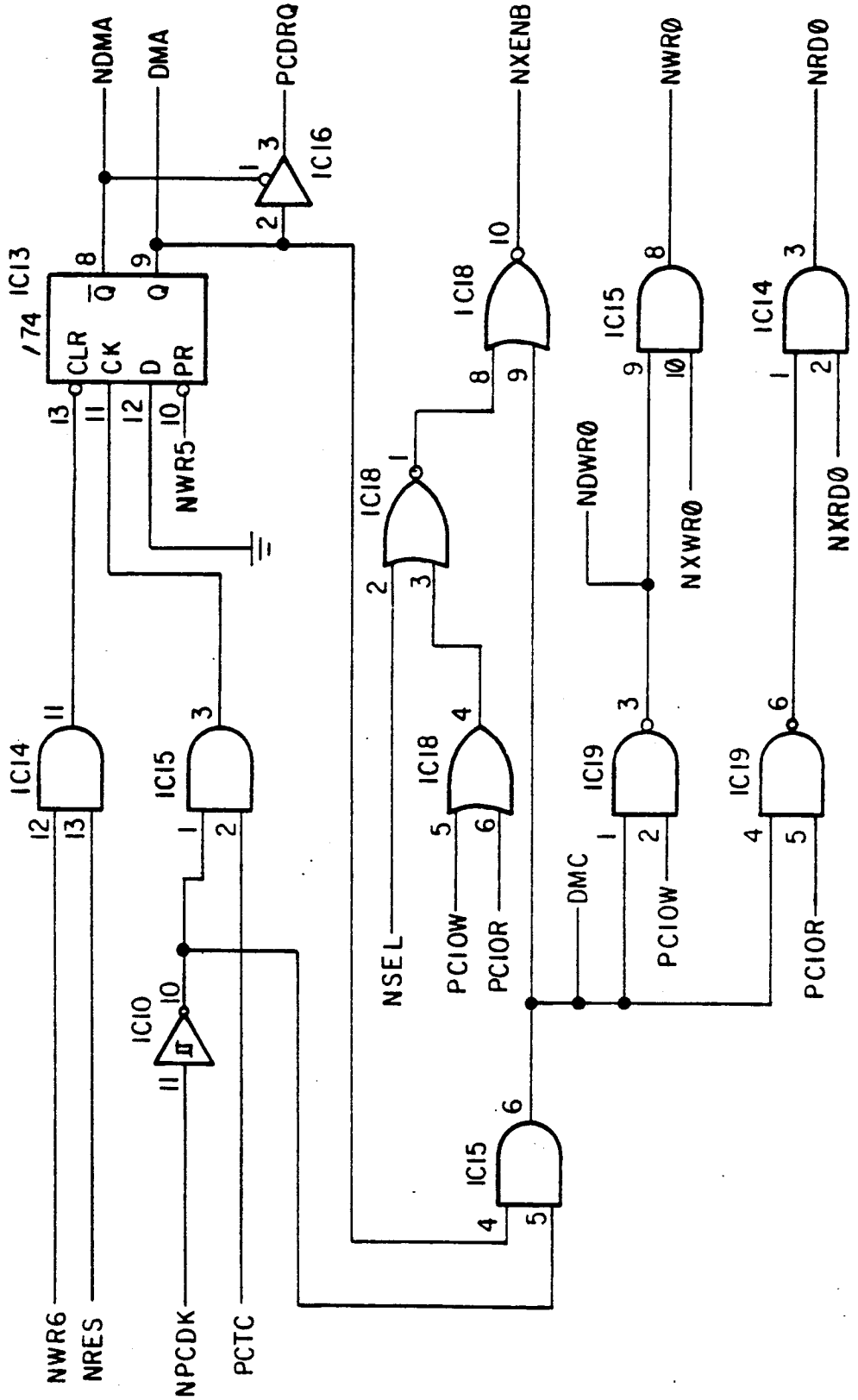

NDMA
DMA
PCDRQ
IC16
IC13
/74
CLR
CK
D
PR
Q
NWR5
NWR6
NRES
NPCDK
PCTC
IC14
IC15
IC10
IC18
NSEL
PCIOW
PCIOR
NXENB
IC15
IC18
IC19
DMC
NDWR0
NWR0
NXWR0
IC15
IC14
IC19
NRD0
NXRD0

FIG. 5

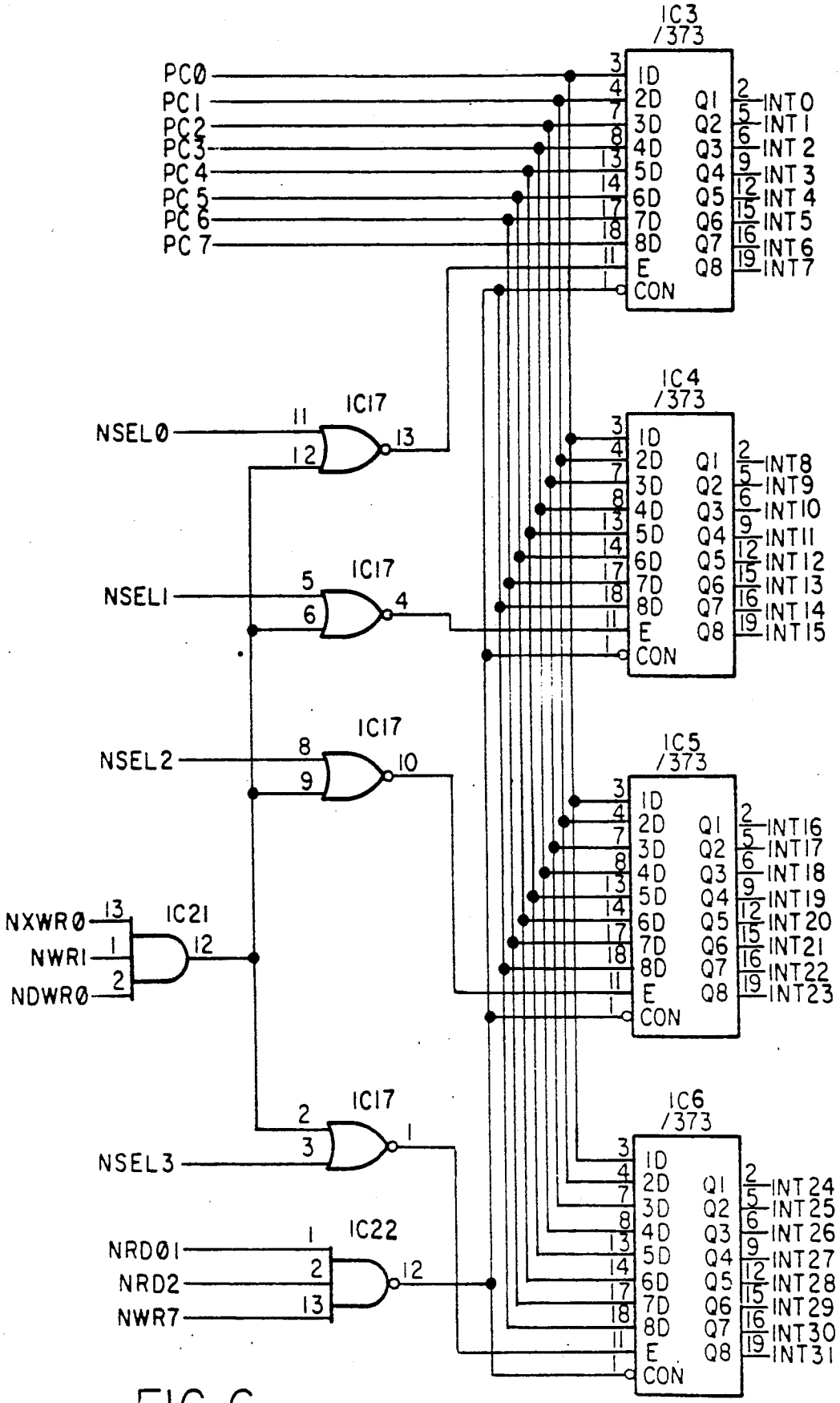

IC3
/373
PC0
PC1
PC2
PC3
PC4
PC5
PC6
PC7
1D
2D
3D
4D
5D
6D
7D
8D
E
CON
Q1
Q2
Q3
Q4
Q5
Q6
Q7
Q8
INT0
INT1
INT2
INT3
INT4
INT5
INT6
INT7
IC4
/373
INT8
INT9
INT10
INT11
INT12
INT13
INT14
INT15
IC5
/373
INT16
INT17
INT18
INT19
INT20
INT21
INT22
INT23
IC6
/373
INT24
INT25
INT26
INT27
INT28
INT29
INT30
INT31
IC17
NSEL0
NSEL1
NSEL2
NSEL3
IC21
NXWR0
NWR1
NDWR0
IC22
NRD01
NRD2
NWR7

FIG. 6

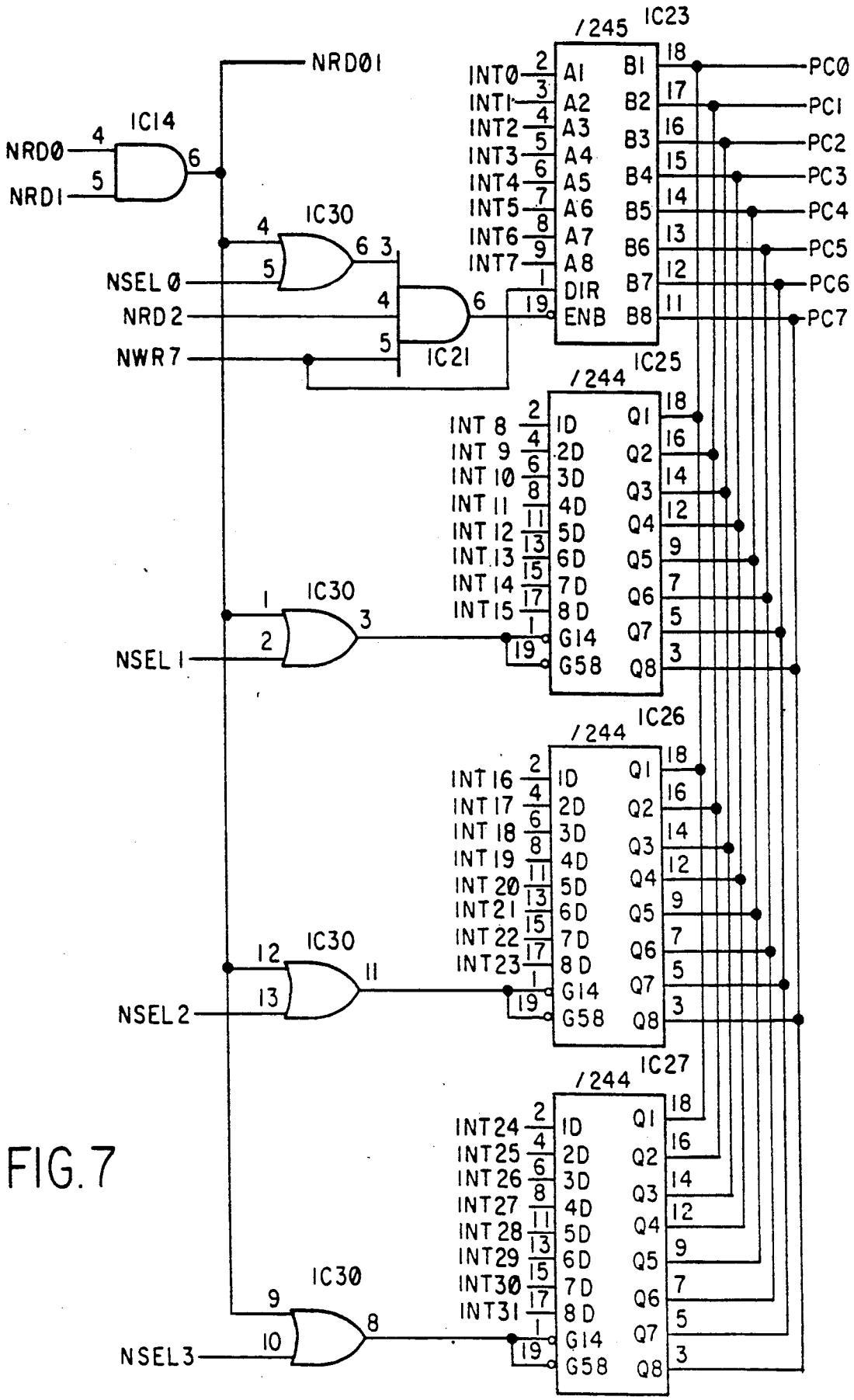

NRD01
IC14
NRD0
NRD1
IC30
NSEL 0
NRD 2
NWR 7
IC21
/245
IC23
INT0
INT1
INT2
INT3
INT4
INT5
INT6
INT7
DIR
ENB
PC0
PC1
PC2
PC3
PC4
PC5
PC6
PC7
/244
IC25
INT 8
INT 9
INT 10
INT 11
INT 12
INT 13
INT 14
INT 15
G14
G58
NSEL 1
IC26
INT 16
INT 17
INT 18
INT 19
INT 20
INT 21
INT 22
INT 23
NSEL 2
IC27
INT24
INT25
INT26
INT27
INT28
INT29
INT30
INT31
NSEL3

FIG. 7

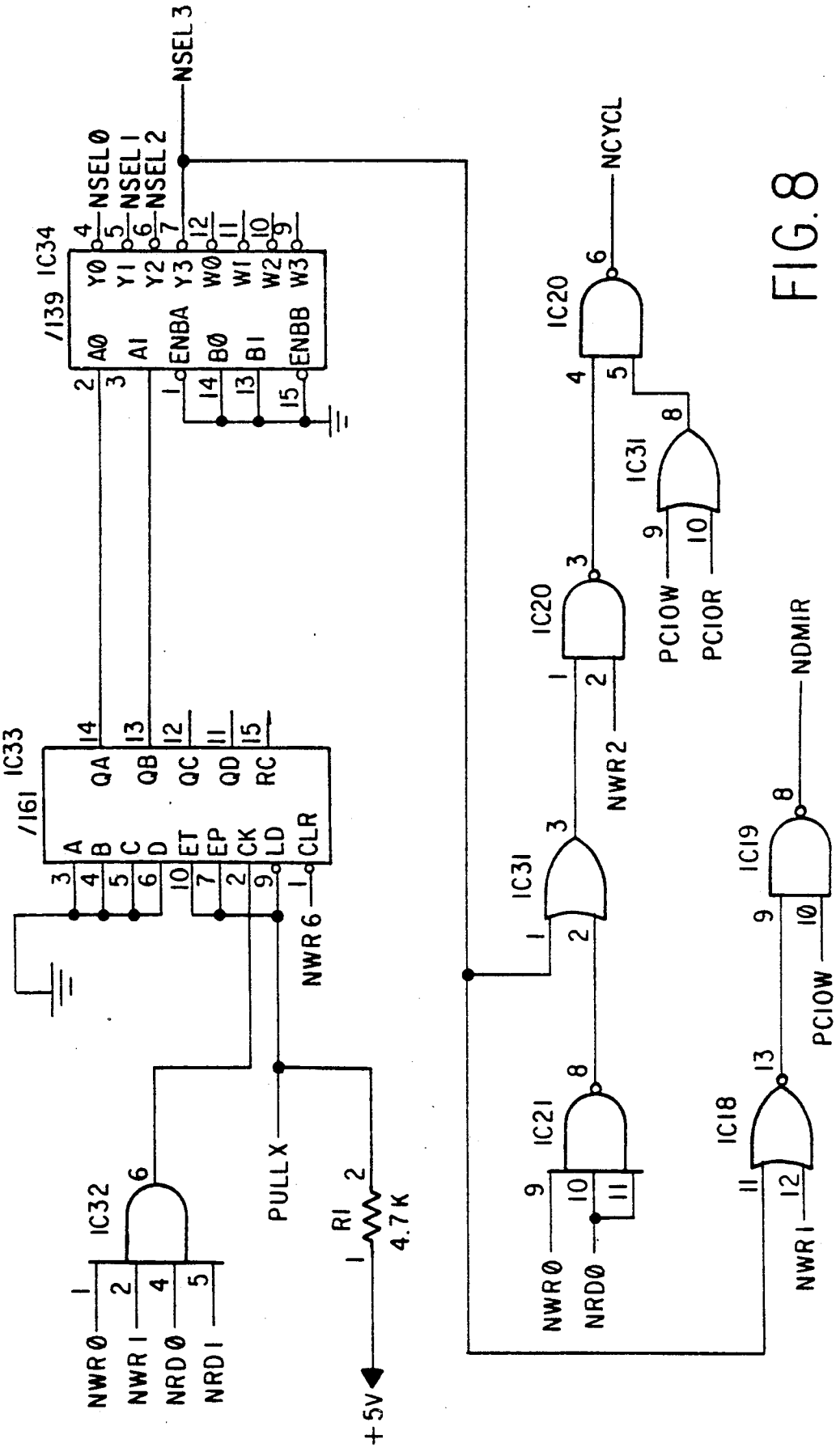

NSEL3
NSEL0
NSEL1
NSEL2
/139 IC34
Y0
Y1
Y2
Y3
W0
W1
W2
W3
A0
A1
ENBA
B0
B1
ENBB
/161 IC33
QA
QB
QC
QD
RC
A
B
C
D
ET
EP
CK
LD
CLR
NWR6
IC32
NWR0
NWR1
NRD0
NRD1
PULLX
RI
4.7K
+5V
NCYCL
IC20
IC31
PC10W
PC10R
NWR2
IC21
NRD0
IC19
NDMIR
IC18
NWR1

FIG. 8

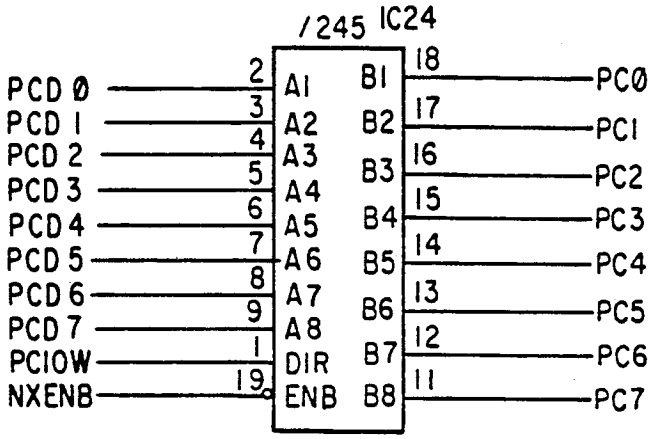

/245 IC24
PCD 0
PCD 1
PCD 2
PCD 3
PCD 4
PCD 5
PCD 6
PCD 7
PCIOW
NXENB
2
3
4
5
6
7
8
9
1
19
A1
A2
A3
A4
A5
A6
A7
A8
DIR
ENB
B1
B2
B3
B4
B5
B6
B7
B8
18
17
16
15
14
13
12
11
PC0
PC1
PC2
PC3
PC4
PC5
PC6
PC7

FIG. 9

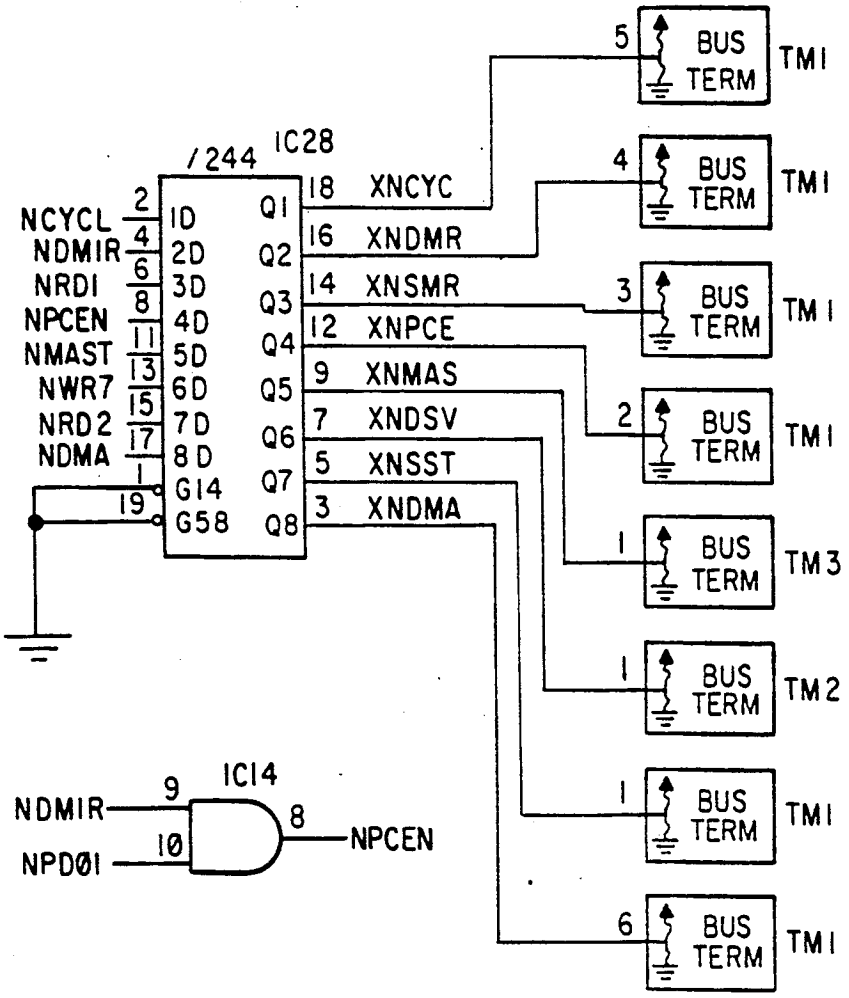

/244 IC28
NCYCL
NDMIR
NRDI
NPCEN
NMAST
NWR7
NRD2
NDMA
2
4
6
8
11
13
15
17
1
19
1D
2D
3D
4D
5D
6D
7D
8D
G14
G58
Q1
Q2
Q3
Q4
Q5
Q6
Q7
Q8
18 XNCYC
16 XNDMR
14 XNSMR
12 XNPCE
9 XNMAS
7 XNDSV
5 XNSST
3 XNDMA
5 BUS TERM TM1
4 BUS TERM TM1
3 BUS TERM TM1
2 BUS TERM TM1
1 BUS TERM TM3
1 BUS TERM TM2
1 BUS TERM TM1
6 BUS TERM TM1
IC14
NDMIR
NPD01
9
10
8
NPCEN

FIG. IC

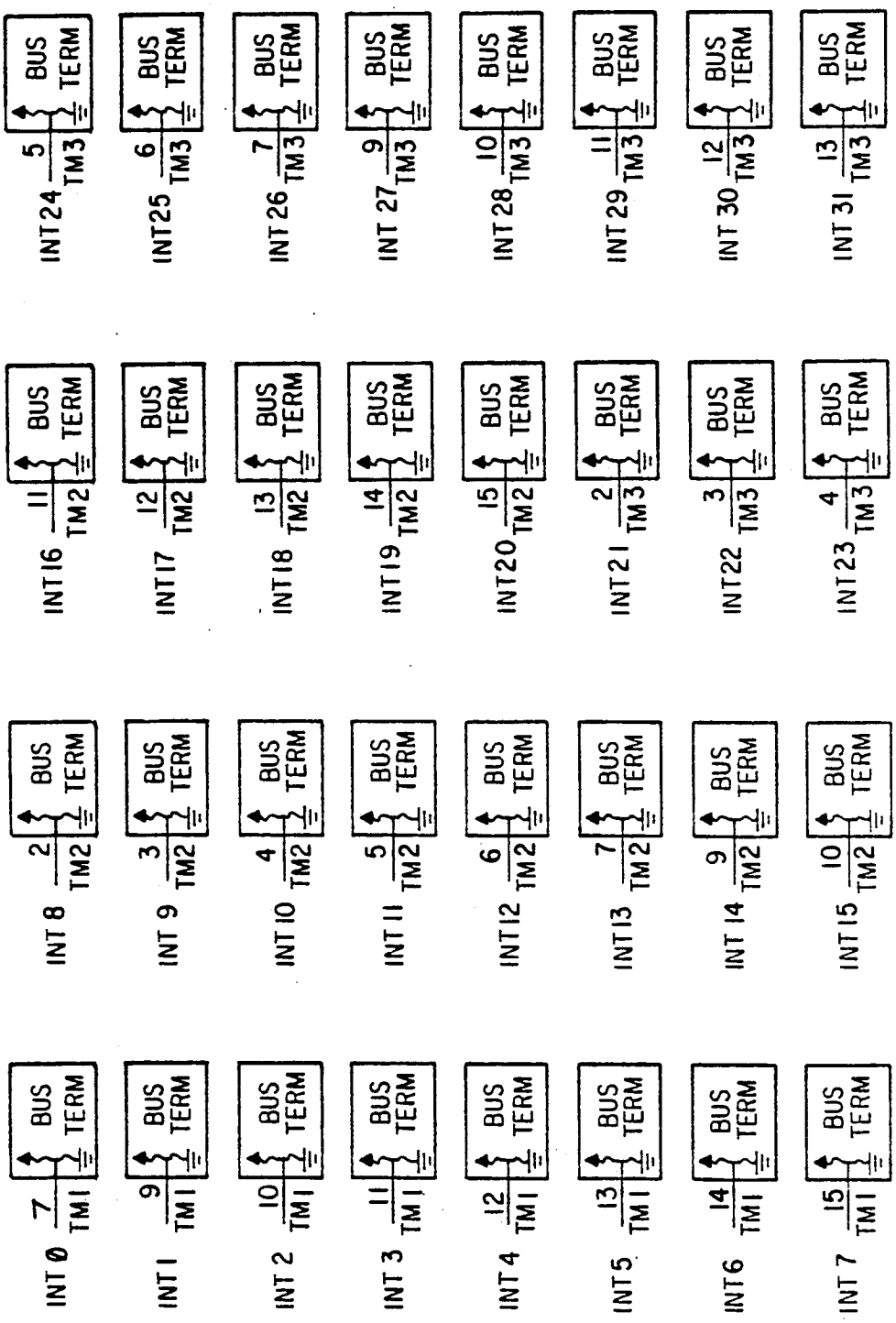

INT 0
7
TM1
BUS
TERM
INT 1
9
TM1
BUS
TERM
INT 2
10
TM1
BUS
TERM
INT 3
11
TM1
BUS
TERM
INT 4
12
TM1
BUS
TERM
INT 5
13
TM1
BUS
TERM
INT 6
14
TM1
BUS
TERM
INT 7
15
TM1
BUS
TERM
INT 8
2
TM2
BUS
TERM
INT 9
3
TM2
BUS
TERM
INT 10
4
TM2
BUS
TERM
INT 11
5
TM2
BUS
TERM
INT 12
6
TM2
BUS
TERM
INT 13
7
TM2
BUS
TERM
INT 14
9
TM2
BUS
TERM
INT 15
10
TM2
BUS
TERM
INT 16
11
TM2
BUS
TERM
INT 17
12
TM2
BUS
TERM
INT 18
13
TM2
BUS
TERM
INT 19
14
TM2
BUS
TERM
INT 20
15
TM2
BUS
TERM
INT 21
2
TM3
BUS
TERM
INT 22
3
TM3
BUS
TERM
INT 23
4
TM3
BUS
TERM
INT 24
5
TM3
BUS
TERM
INT 25
6
TM3
BUS
TERM
INT 26
7
TM3
BUS
TERM
INT 27
9
TM3
BUS
TERM
INT 28
10
TM3
BUS
TERM
INT 29
11
TM3
BUS
TERM
INT 30
12
TM3
BUS
TERM
INT 31
13
TM3
BUS
TERM

FIG. 11

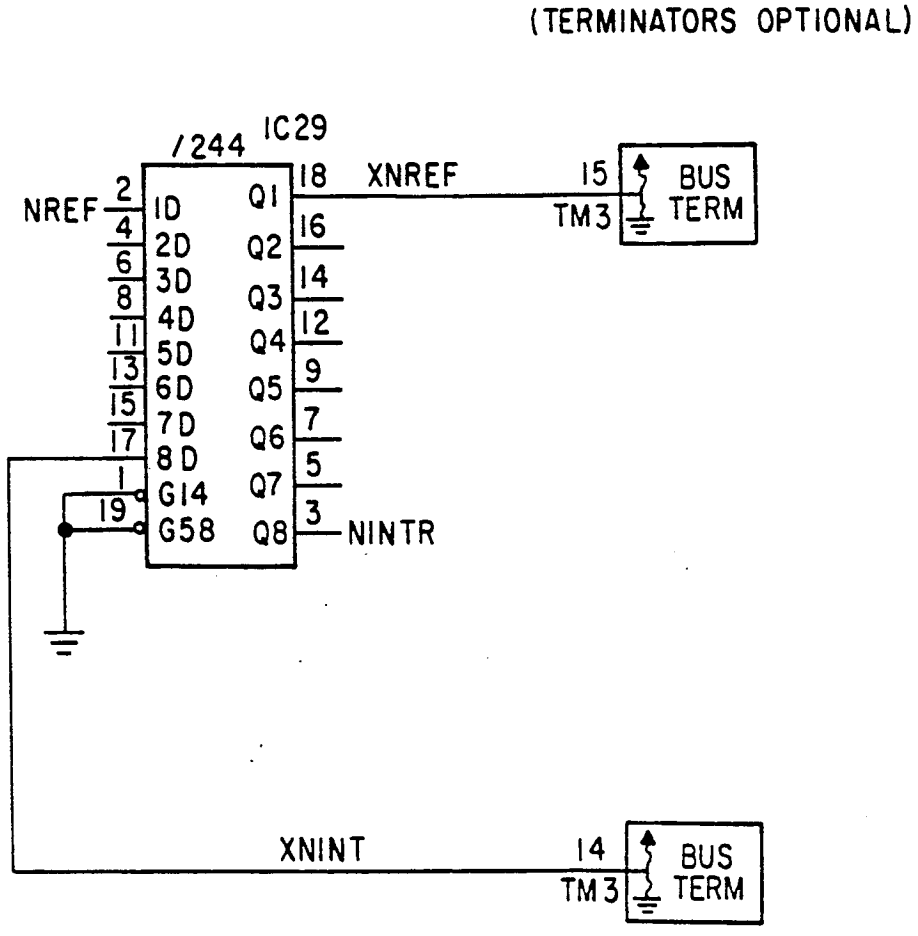

(TERMINATORS OPTIONAL)
IC29
1244
NREF
2
4
6
8
11
13
15
17
1
19
1D
2D
3D
4D
5D
6D
7D
8D
G14
G58
Q1
Q2
Q3
Q4
Q5
Q6
Q7
Q8
18
16
14
12
9
7
5
3
XNREF
NINTR
15
TM3
BUS TERM
XNINT
14
TM3
BUS TERM

FIG. 12

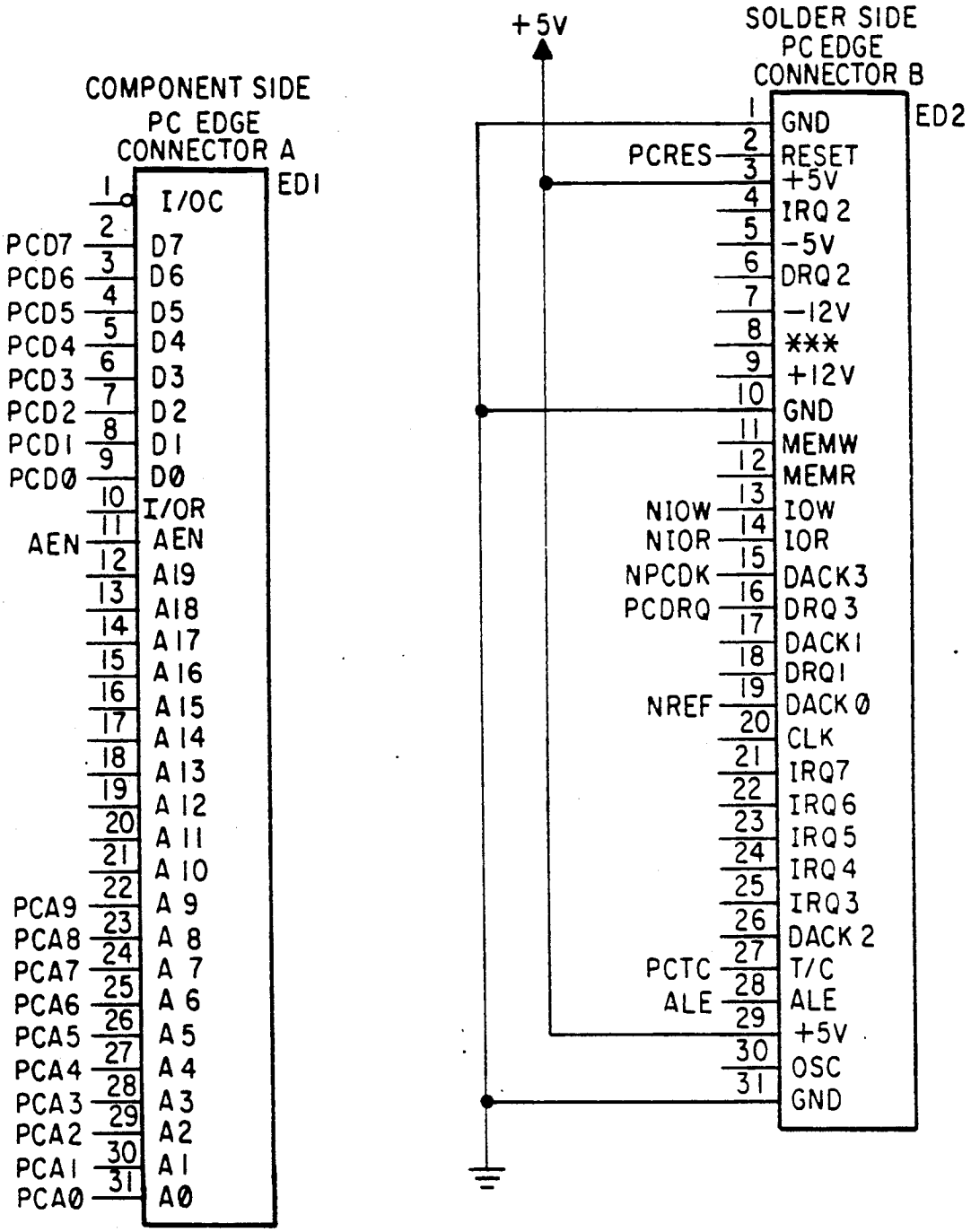

COMPONENT SIDE
PC EDGE
CONNECTOR A
ED1
1 I/OC
PCD7 2 D7
PCD6 3 D6
PCD5 4 D5
PCD4 5 D4
PCD3 6 D3
PCD2 7 D2
PCD1 8 D1
PCD0 9 D0
10 I/OR
AEN 11 AEN
12 A19
13 A18
14 A17
15 A16
16 A15
17 A14
18 A13
19 A12
20 A11
21 A10
PCA9 22 A9
PCA8 23 A8
PCA7 24 A7
PCA6 25 A6
PCA5 26 A5
PCA4 27 A4
PCA3 28 A3
PCA2 29 A2
PCA1 30 A1
PCA0 31 A0
+5V
SOLDER SIDE
PC EDGE
CONNECTOR B
ED2
1 GND
PCRES 2 RESET
3 +5V
4 IRQ2
5 -5V
6 DRQ2
7 -12V
8 ***
9 +12V
10 GND
11 MEMW
12 MEMR
NIOW 13 IOW
NIOR 14 IOR
NPCDK 15 DACK3
PCDRQ 16 DRQ3
17 DACK1
18 DRQ1
NREF 19 DACK0
20 CLK
21 IRQ7
22 IRQ6
23 IRQ5
24 IRQ4
25 IRQ3
26 DACK2
PCTC 27 T/C
ALE 28 ALE
29 +5V
30 OSC
31 GND

FIG.13

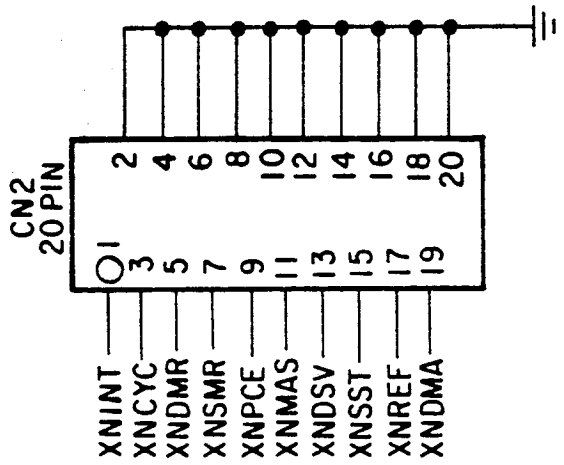
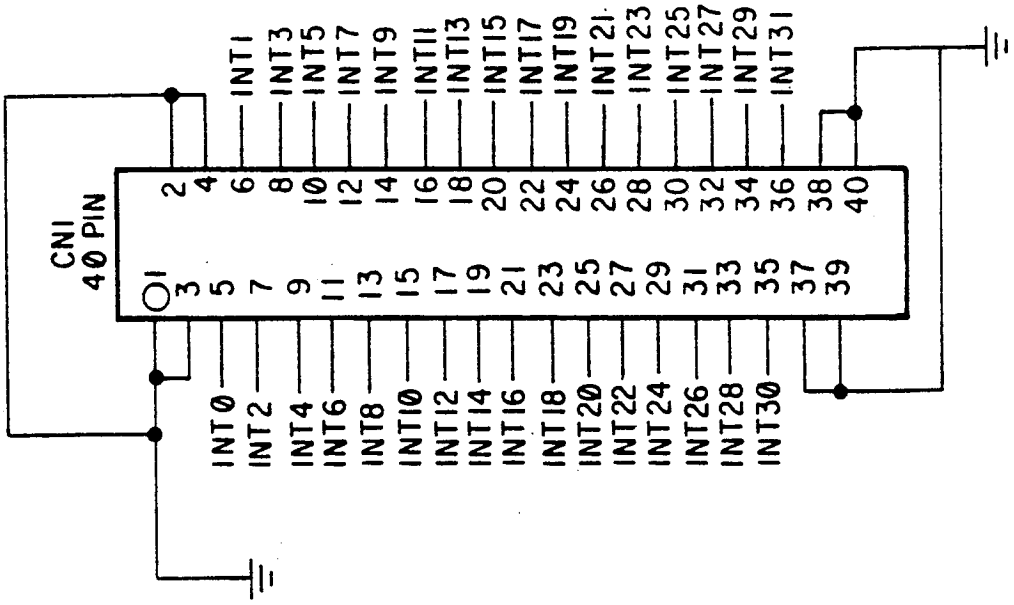

CN2
20 PIN
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
XNINT
XNCYC
XNDMR
XNSMR
XNPCE
XNMAS
XNDSV
XNSST
XNREF
XNDMA

FIG. 14


CN1
40 PIN
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
INT0
INT1
INT2
INT3
INT4
INT5
INT6
INT7
INT8
INT9
INT10
INT11
INT12
INT13
INT14
INT15
INT16
INT17
INT18
INT19
INT20
INT21
INT22
INT23
INT24
INT25
INT26
INT27
INT28
INT29
INT30
INT31

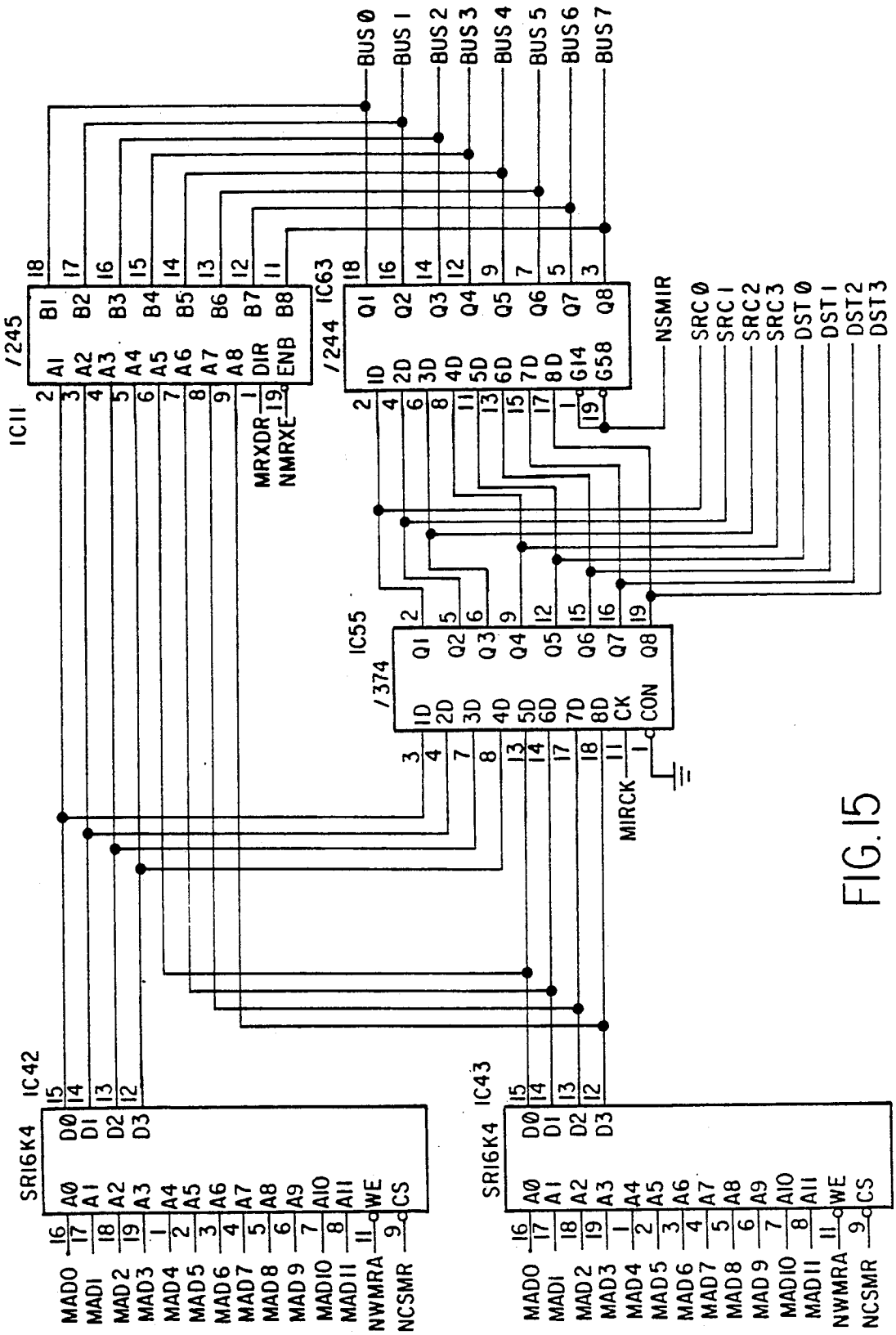

BUS 0
BUS 1
BUS 2
BUS 3
BUS 4
BUS 5
BUS 6
BUS 7
IC11
/245
B1 B2 B3 B4 B5 B6 B7 B8
A1 A2 A3 A4 A5 A6 A7 A8 DIR ENB
MRXDR
NMRXE
IC63
/244
Q1 Q2 Q3 Q4 Q5 Q6 Q7 Q8
1D 2D 3D 4D 5D 6D 7D 8D G14 G58
NSMIR
SRC 0
SRC 1
SRC2
SRC3
DST 0
DST 1
DST2
DST3
IC55
/374
Q1 Q2 Q3 Q4 Q5 Q6 Q7 Q8
1D 2D 3D 4D 5D 6D 7D 8D CK CON
MIRCK
IC42
SR16K4
D0 D1 D2 D3
A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 A10 A11 WE CS
MAD0 MAD1 MAD2 MAD3 MAD4 MAD5 MAD6 MAD7 MAD8 MAD9 MAD10 MAD11 NWMRA NCSMR
IC43
SR16K4
D0 D1 D2 D3
A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 A10 A11 WE CS
MAD0 MAD1 MAD2 MAD3 MAD4 MAD5 MAD6 MAD7 MAD8 MAD9 MAD10 MAD11 NWMRA NCSMR

FIG. 15

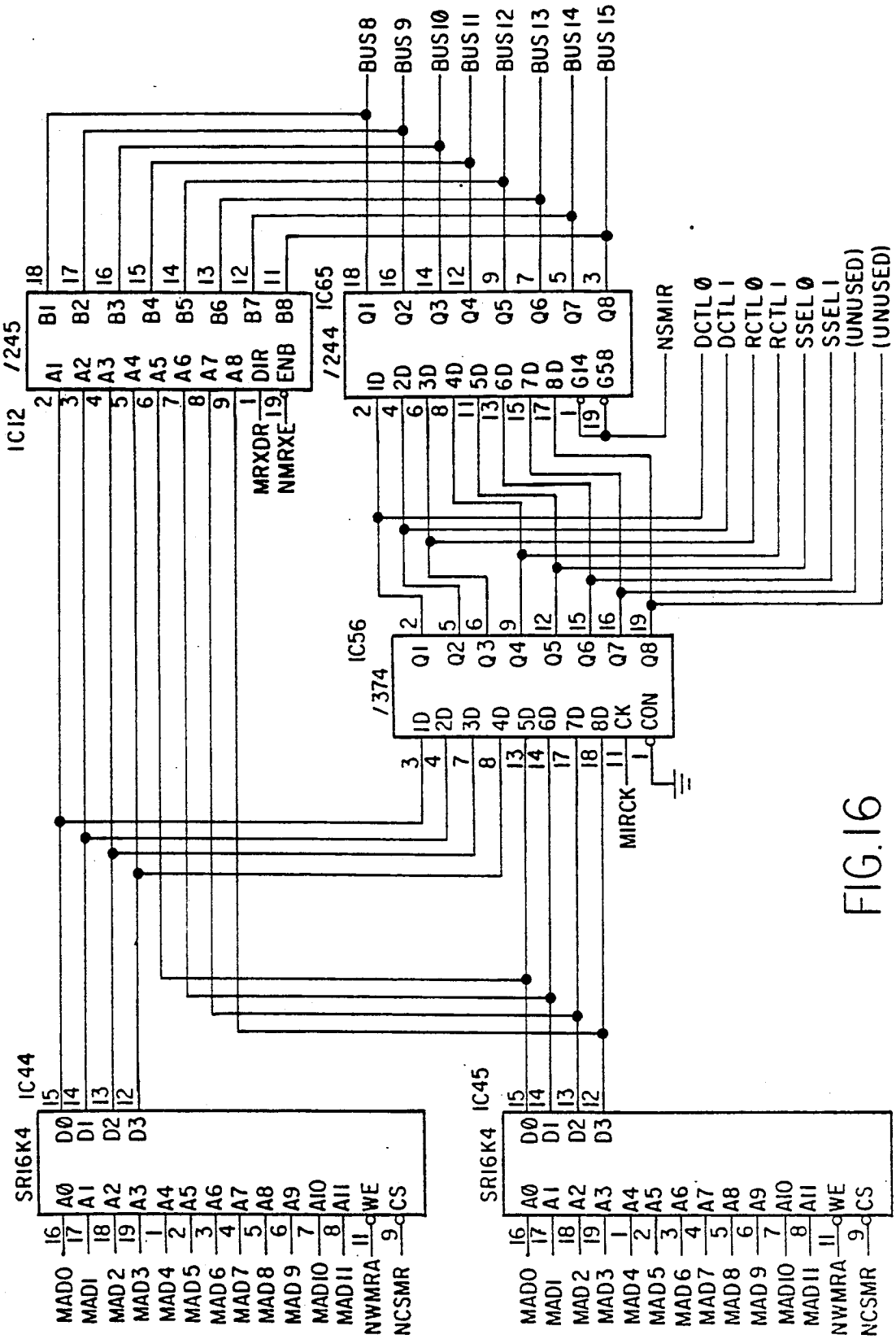

IC12
/245
A1 A2 A3 A4 A5 A6 A7 A8 DIR ENB
B1 B2 B3 B4 B5 B6 B7 B8
MRXDR
NMRXE
IC65
/244
1D 2D 3D 4D 5D 6D 7D 8D G14 G58
Q1 Q2 Q3 Q4 Q5 Q6 Q7 Q8
BUS8 BUS9 BUS10 BUS11 BUS12 BUS13 BUS14 BUS15
NSMIR
DCTL0 DCTL1 RCTL0 RCTL1 SSEL0 SSEL1 (UNUSED) (UNUSED)
IC56
/374
1D 2D 3D 4D 5D 6D 7D 8D CK CON
Q1 Q2 Q3 Q4 Q5 Q6 Q7 Q8
MIRCK
IC44
SR16K4
D0 D1 D2 D3
A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 A10 A11 WE CS
MAD0 MAD1 MAD2 MAD3 MAD4 MAD5 MAD6 MAD7 MAD8 MAD9 MAD10 MAD11 NWMRA NCSMR
IC45
SR16K4
D0 D1 D2 D3
A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 A10 A11 WE CS
MAD0 MAD1 MAD2 MAD3 MAD4 MAD5 MAD6 MAD7 MAD8 MAD9 MAD10 MAD11 NWMRA NCSMR

FIG. 16

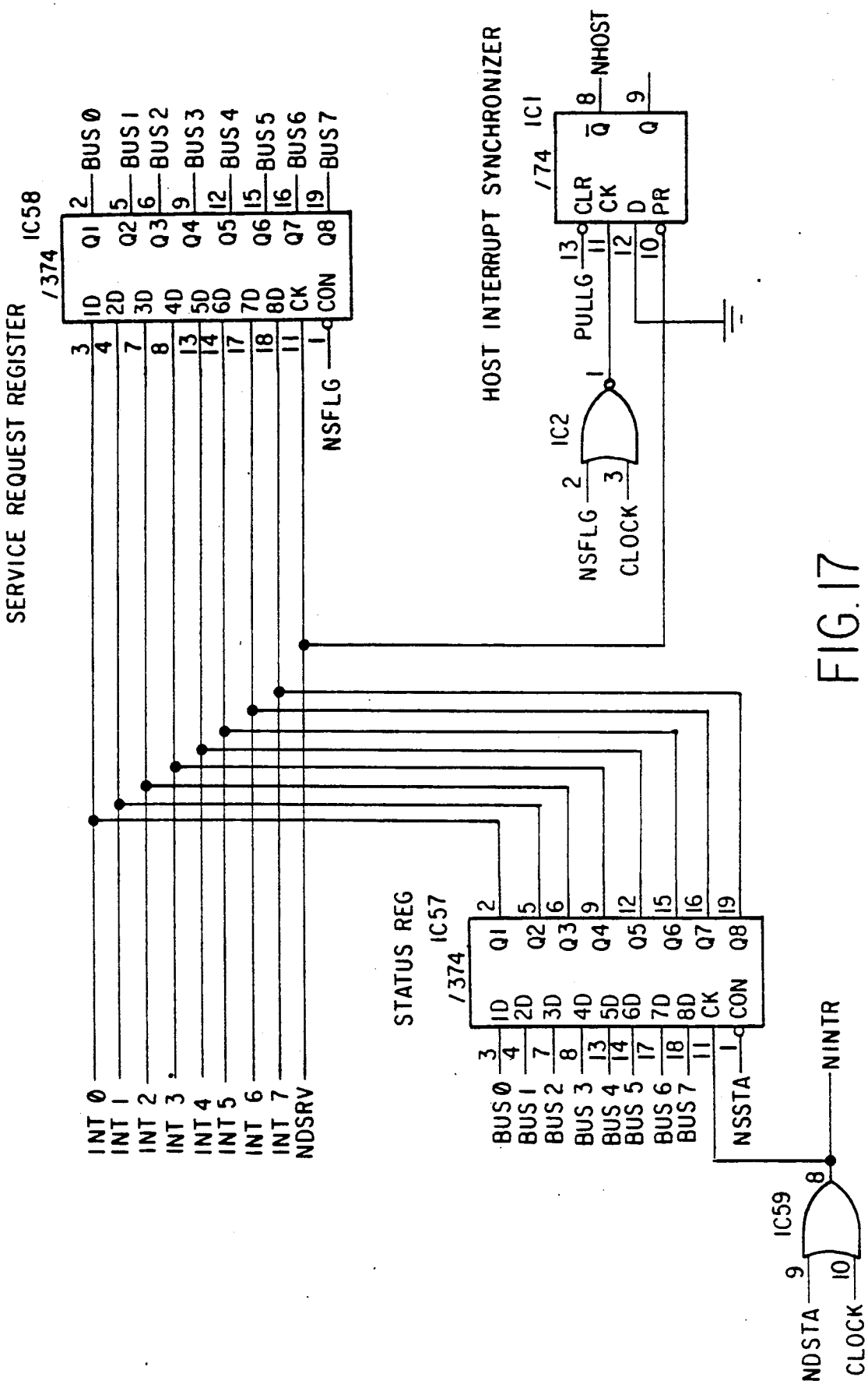

SERVICE REQUEST REGISTER
IC58
/374
1D
2D
3D
4D
5D
6D
7D
8D
CK
CON
Q1
Q2
Q3
Q4
Q5
Q6
Q7
Q8
BUS 0
BUS 1
BUS 2
BUS 3
BUS 4
BUS 5
BUS 6
BUS 7
NSFLG
HOST INTERRUPT SYNCHRONIZER
IC1
/74
CLR
CK
D
PR
Q
NHOST
PULLG
IC2
NSFLG
CLOCK
STATUS REG
IC57
/374
NSSTA
NINTR
IC59
NDSTA
CLOCK
INT 0
INT 1
INT 2
INT 3
INT 4
INT 5
INT 6
INT 7
NDSRV

FIG. 17

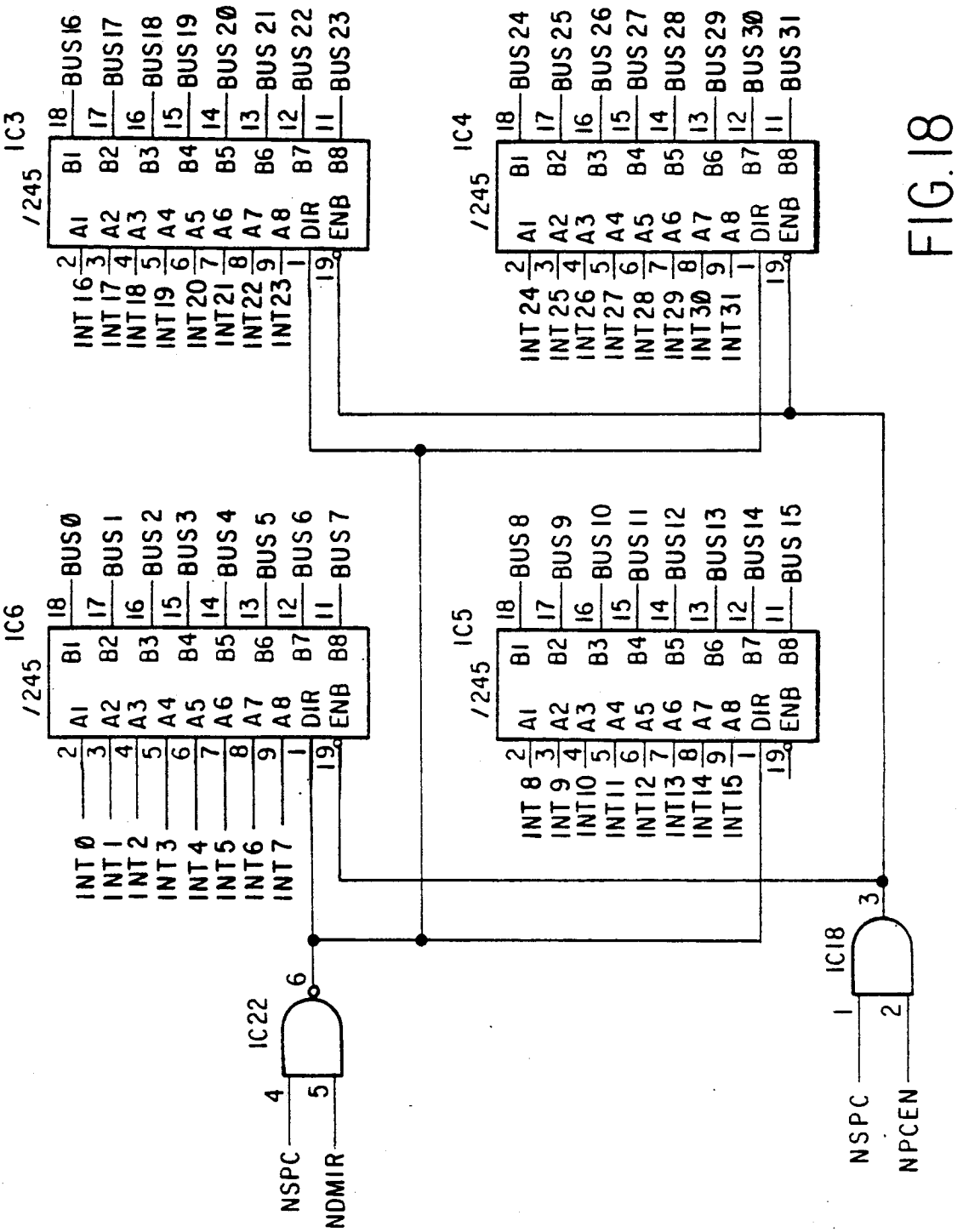

IC3
/245
IC4
/245
IC6
/245
IC5
/245
IC22
IC18
NSPC
NDMIR
NSPC
NPCEN

FIG. 18

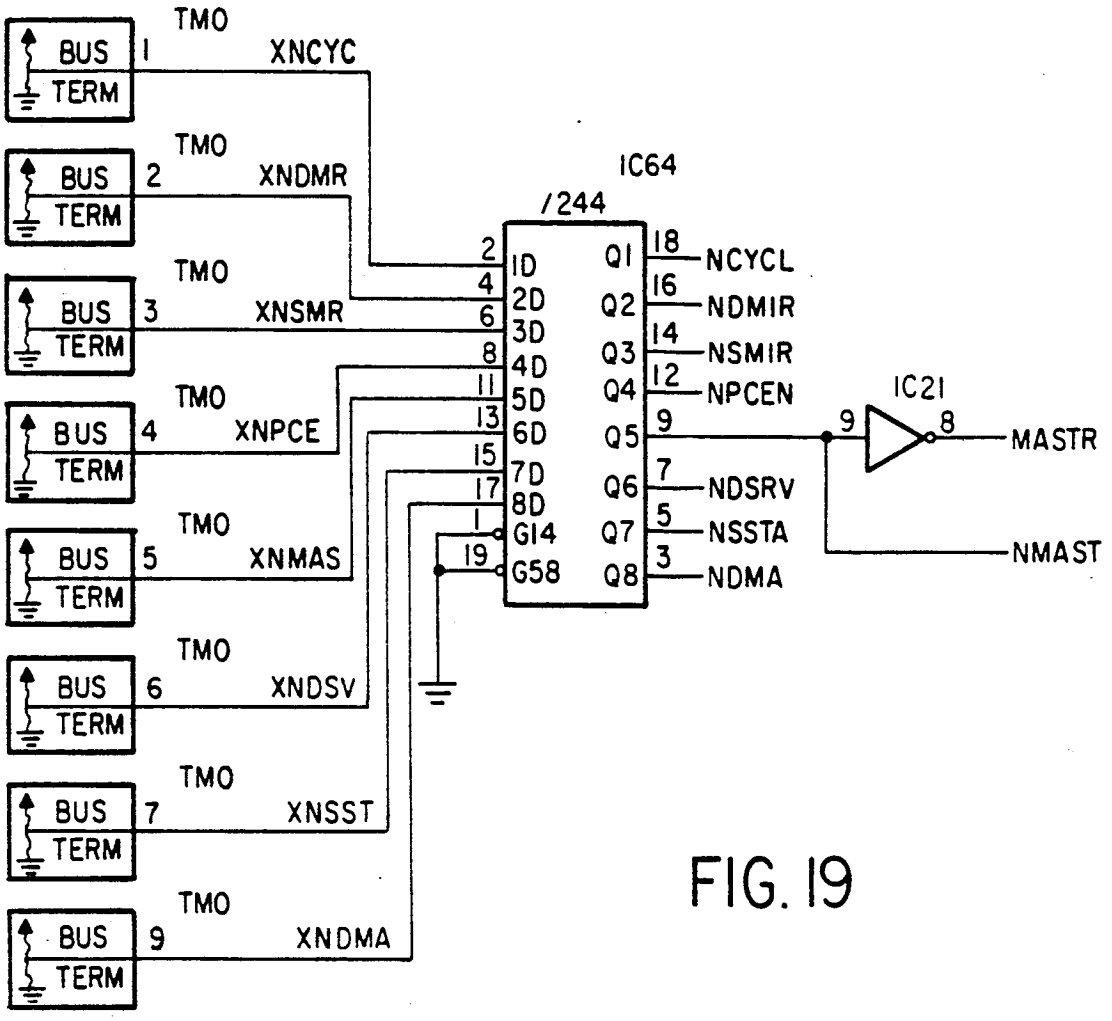

TMO
BUS
TERM
1
XNCYC
2
XNDMR
3
XNSMR
4
XNPCE
5
XNMAS
6
XNDSV
7
XNSST
9
XNDMA
IC64
/244
2 1D
4 2D
6 3D
8 4D
11 5D
13 6D
15 7D
17 8D
1 G14
19 G58
18 Q1
16 Q2
14 Q3
12 Q4
9 Q5
7 Q6
5 Q7
3 Q8
NCYCL
NDMIR
NSMIR
NPCEN
NDSRV
NSSTA
NDMA
IC21
9
8
MASTR
NMAST

FIG. 19

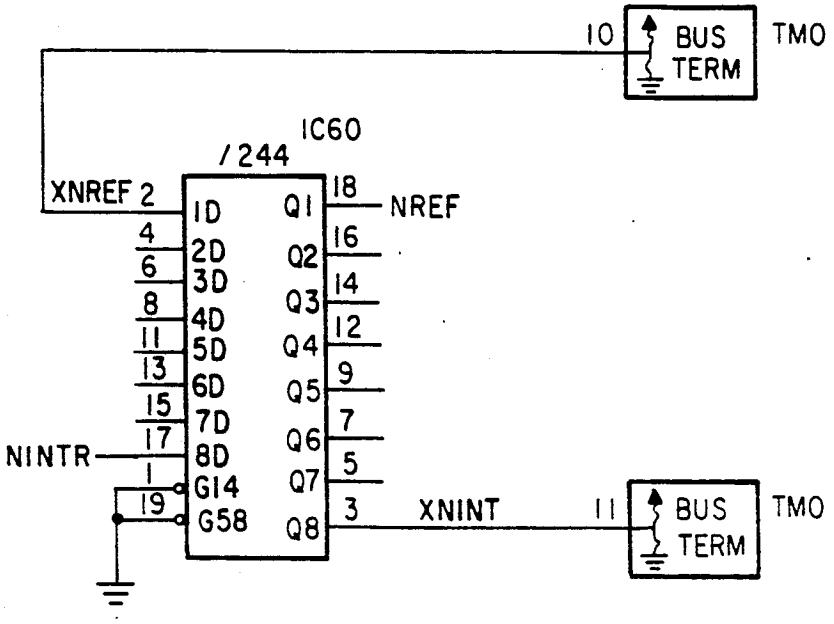

10
BUS
TERM
TMO
IC60
/244
XNREF 2
4 2D
6 3D
8 4D
11 5D
13 6D
15 7D
NINTR 17 8D
1 G14
19 G58
18 Q1
16 Q2
14 Q3
12 Q4
9 Q5
7 Q6
5 Q7
3 Q8
NREF
XNINT
11
BUS
TERM
TMO

FIG. 20

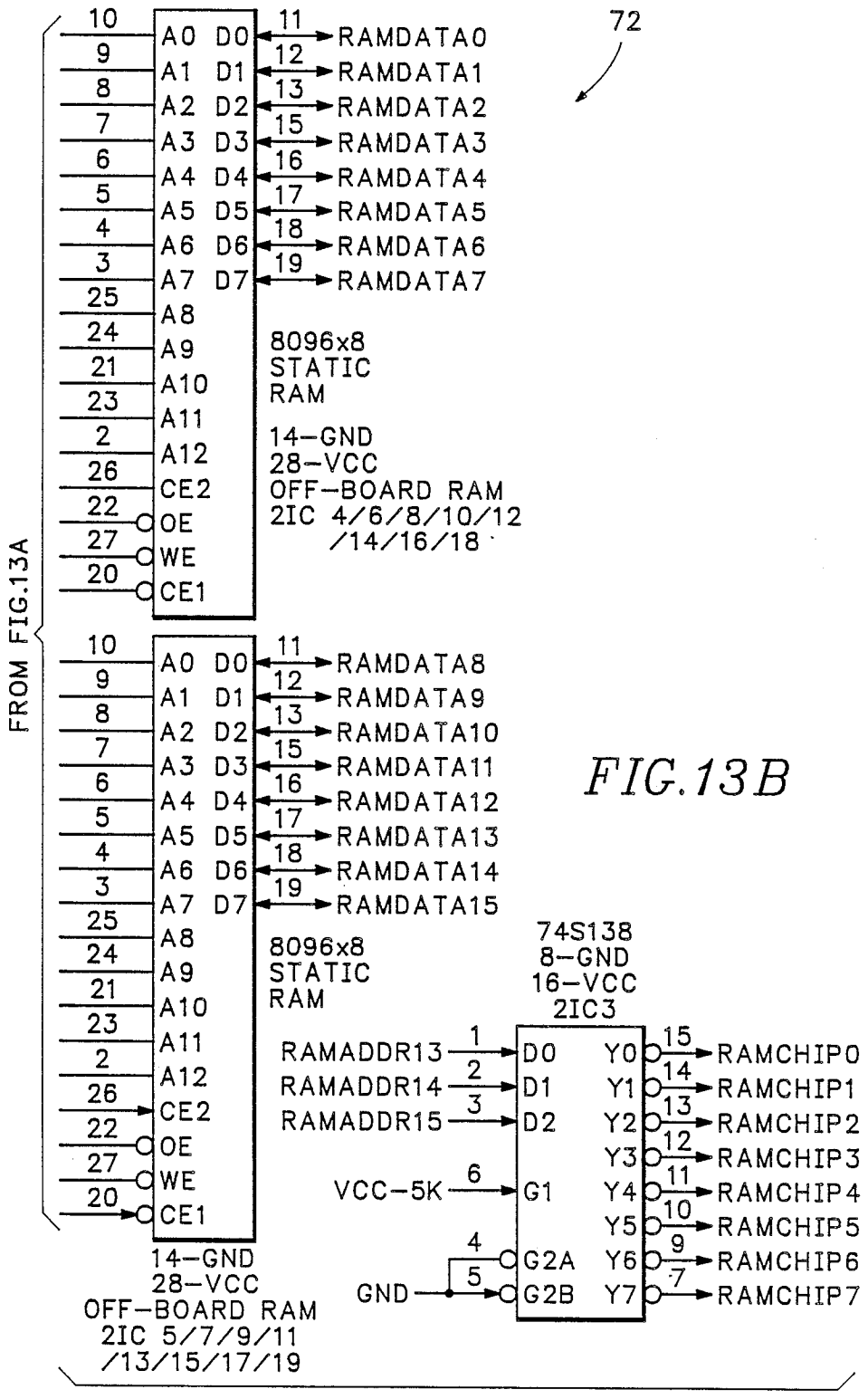

72
FROM FIG.13A
A0 D0 RAMDATA0
A1 D1 RAMDATA1
A2 D2 RAMDATA2
A3 D3 RAMDATA3
A4 D4 RAMDATA4
A5 D5 RAMDATA5
A6 D6 RAMDATA6
A7 D7 RAMDATA7
A8
A9
A10
A11
A12
CE2
OE
WE
CE1
8096x8 STATIC RAM
14-GND
28-VCC
OFF-BOARD RAM
2IC 4/6/8/10/12 /14/16/18
RAMDATA8
RAMDATA9
RAMDATA10
RAMDATA11
RAMDATA12
RAMDATA13
RAMDATA14
RAMDATA15
14-GND
28-VCC
OFF-BOARD RAM
2IC 5/7/9/11 /13/15/17/19
74S138
8-GND
16-VCC
2IC3
RAMADDR13
RAMADDR14
RAMADDR15
VCC-5K
GND
D0
D1
D2
G1
G2A
G2B
Y0
Y1
Y2
Y3
Y4
Y5
Y6
Y7
RAMCHIP0
RAMCHIP1
RAMCHIP2
RAMCHIP3
RAMCHIP4
RAMCHIP5
RAMCHIP6
RAMCHIP7

FIG.13B

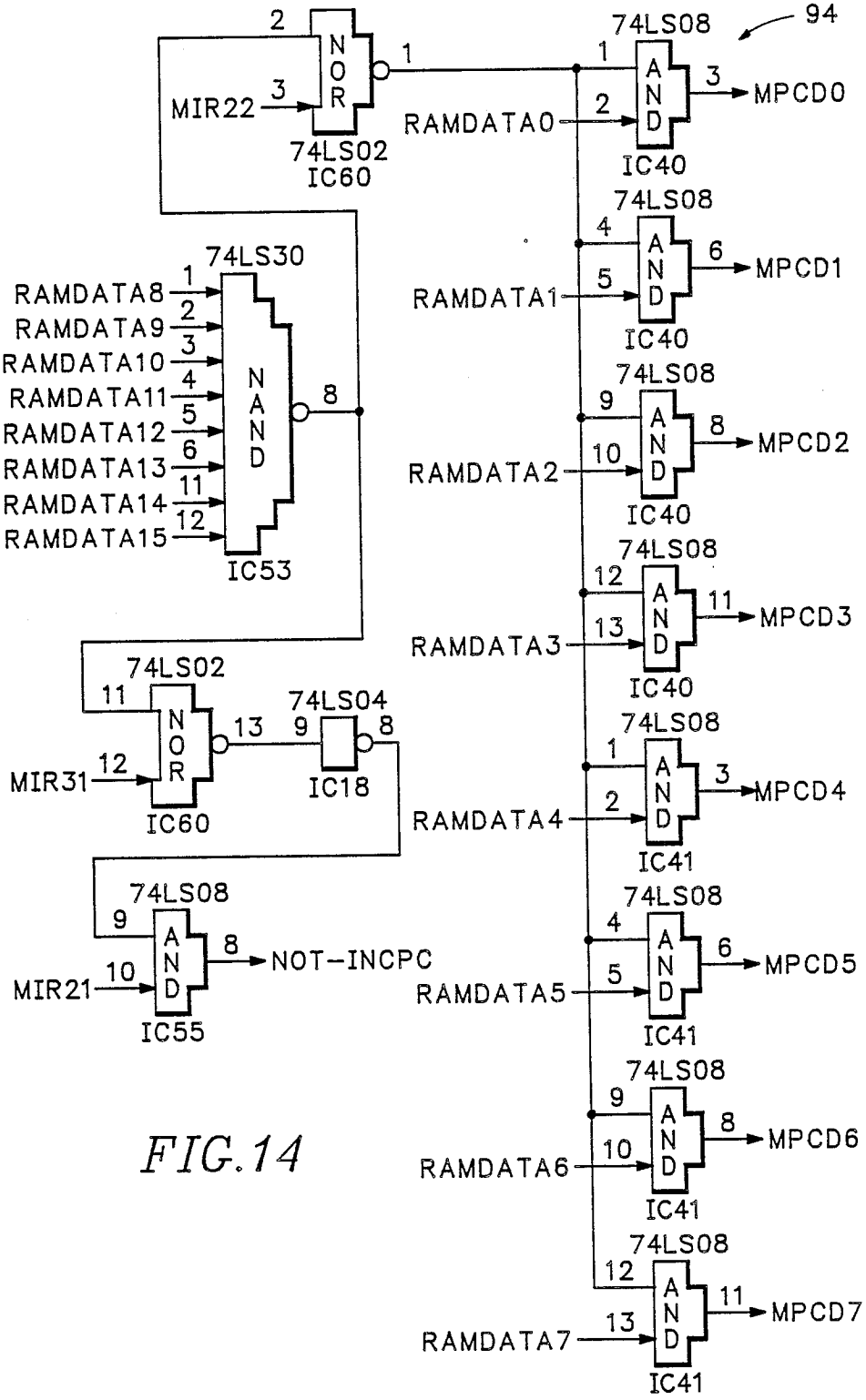

74LS02
IC60
MIR22
74LS30
IC53
RAMDATA8
RAMDATA9
RAMDATA10
RAMDATA11
RAMDATA12
RAMDATA13
RAMDATA14
RAMDATA15
74LS02
IC60
MIR31
74LS04
IC18
74LS08
IC55
MIR21
NOT-INCPC
94
74LS08
IC40
RAMDATA0
MPCD0
74LS08
IC40
RAMDATA1
MPCD1
74LS08
IC40
RAMDATA2
MPCD2
74LS08
IC40
RAMDATA3
MPCD3
74LS08
IC41
RAMDATA4
MPCD4
74LS08
IC41
RAMDATA5
MPCD5
74LS08
IC41
RAMDATA6
MPCD6
74LS08
IC41
RAMDATA7
MPCD7

FIG. 14

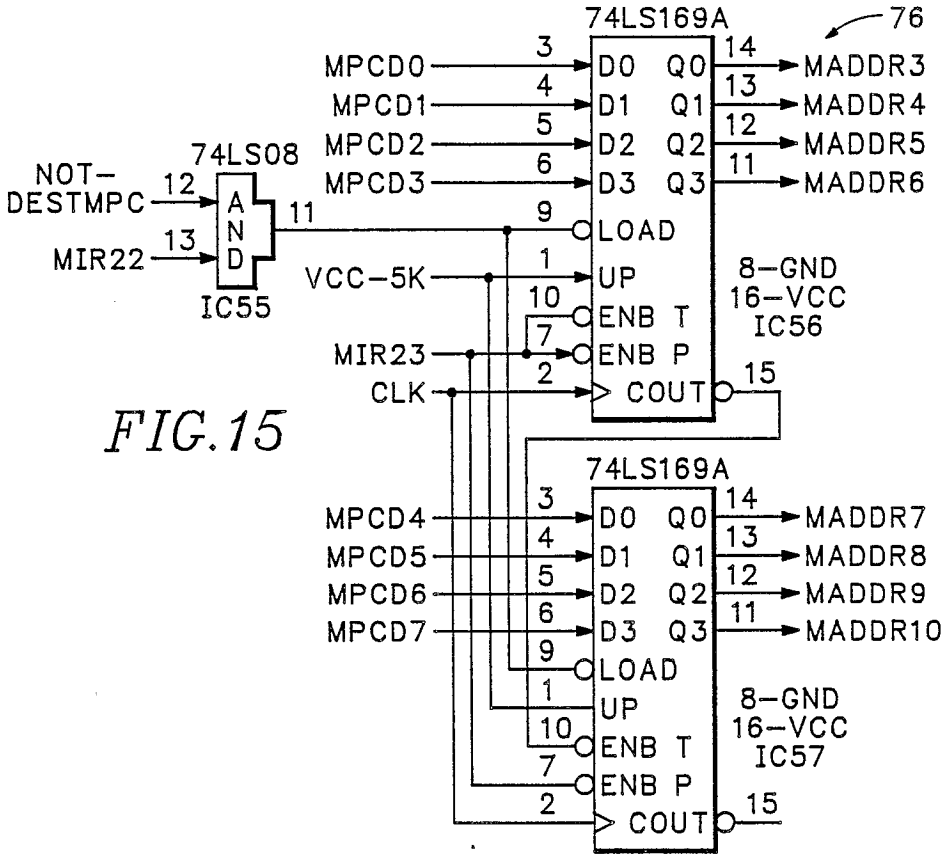

74LS169A
76
MPCD0
MPCD1
MPCD2
MPCD3
D0 Q0
D1 Q1
D2 Q2
D3 Q3
MADDR3
MADDR4
MADDR5
MADDR6
74LS08
NOT-DESTMPC
MIR22
A N D
IC55
LOAD
UP
ENB T
ENB P
COUT
VCC-5K
MIR23
CLK
8-GND
16-VCC
IC56
74LS169A
MPCD4
MPCD5
MPCD6
MPCD7
MADDR7
MADDR8
MADDR9
MADDR10
8-GND
16-VCC
IC57

FIG. 15

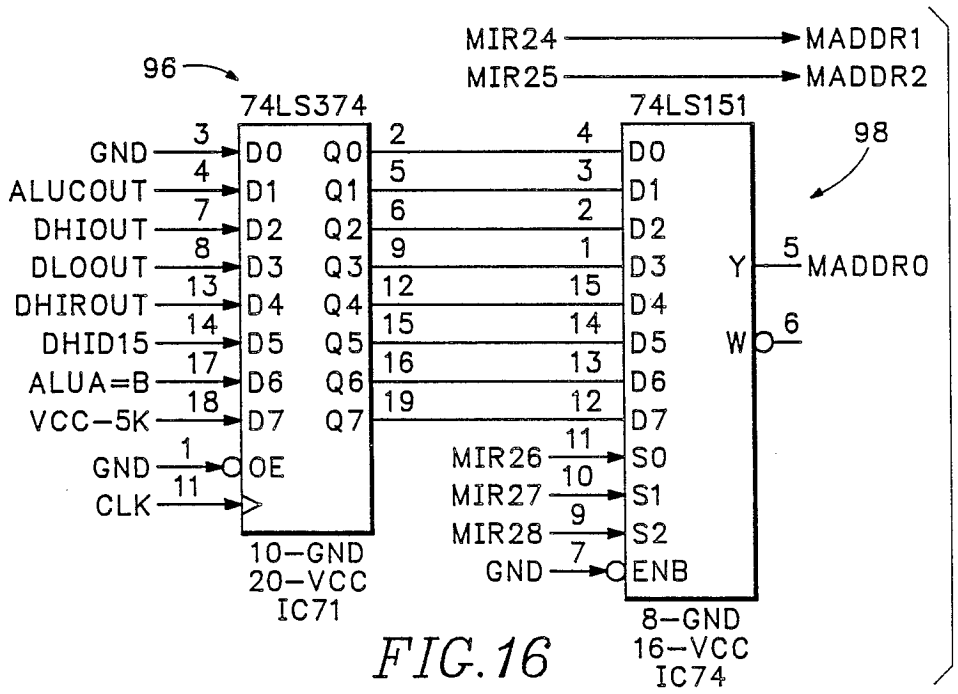

MIR24
MIR25
MADDR1
MADDR2
96
74LS374
74LS151
98
GND
ALUCOUT
DHIOUT
DLOOUT
DHIROUT
DHID15
ALUA=B
VCC-5K
GND
CLK
D0 Q0
D1 Q1
D2 Q2
D3 Q3
D4 Q4
D5 Q5
D6 Q6
D7 Q7
OE
10-GND
20-VCC
IC71
MIR26
MIR27
MIR28
GND
S0
S1
S2
ENB
Y
W
MADDR0
8-GND
16-VCC
IC74

FIG. 16

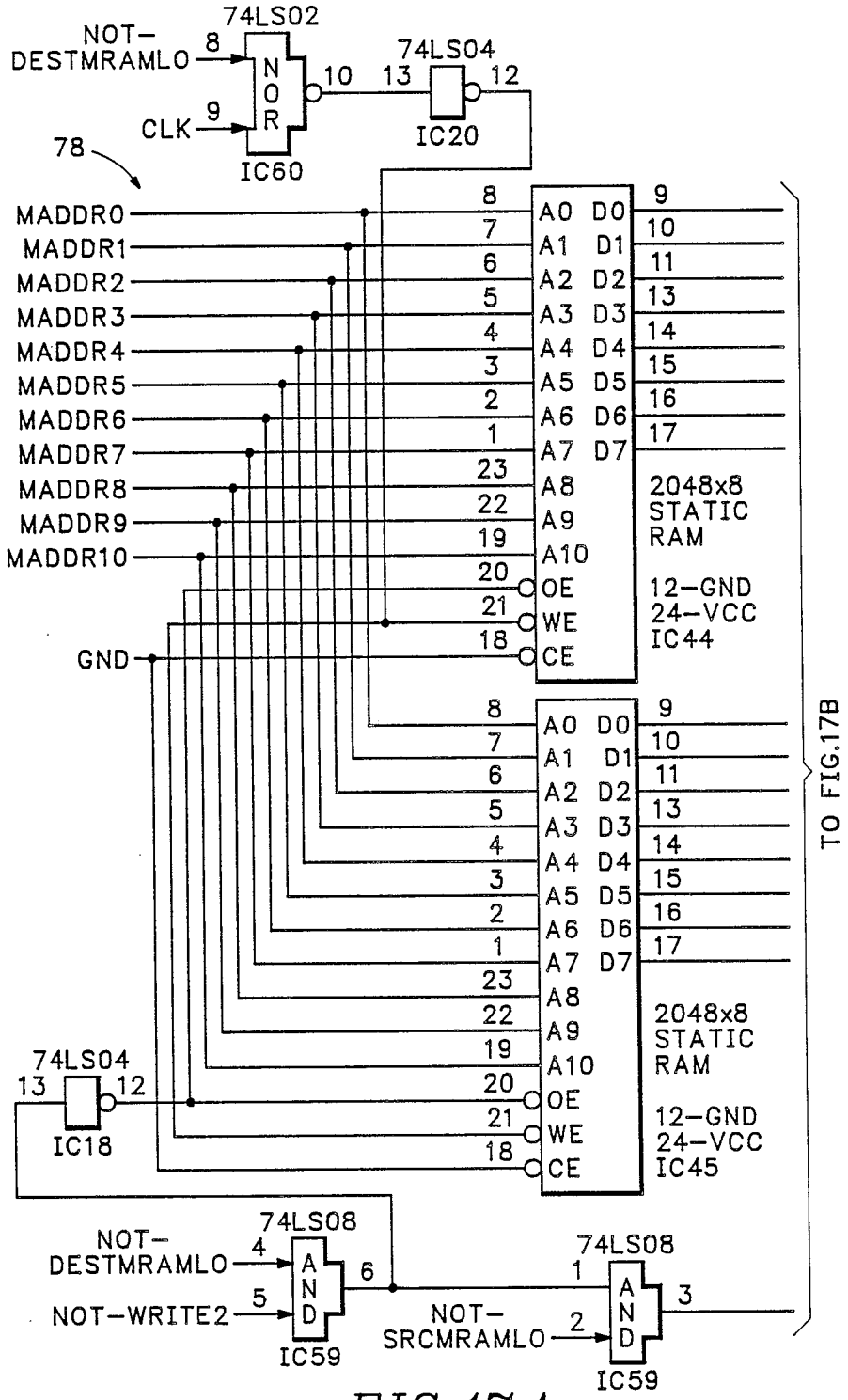

74LS02
NOT-
DESTMRAMLO
8
CLK
9
NOR
10
IC60
74LS04
13
12
IC20
78
MADDR0
MADDR1
MADDR2
MADDR3
MADDR4
MADDR5
MADDR6
MADDR7
MADDR8
MADDR9
MADDR10
GND
A0 D0
A1 D1
A2 D2
A3 D3
A4 D4
A5 D5
A6 D6
A7 D7
A8
A9
A10
OE
WE
CE
2048x8
STATIC
RAM
12-GND
24-VCC
IC44
IC45
74LS04
IC18
74LS08
NOT-
DESTMRAMLO
NOT-WRITE2
AND
IC59
NOT-
SRCMRAMLO
TO FIG.17B

FIG.17A

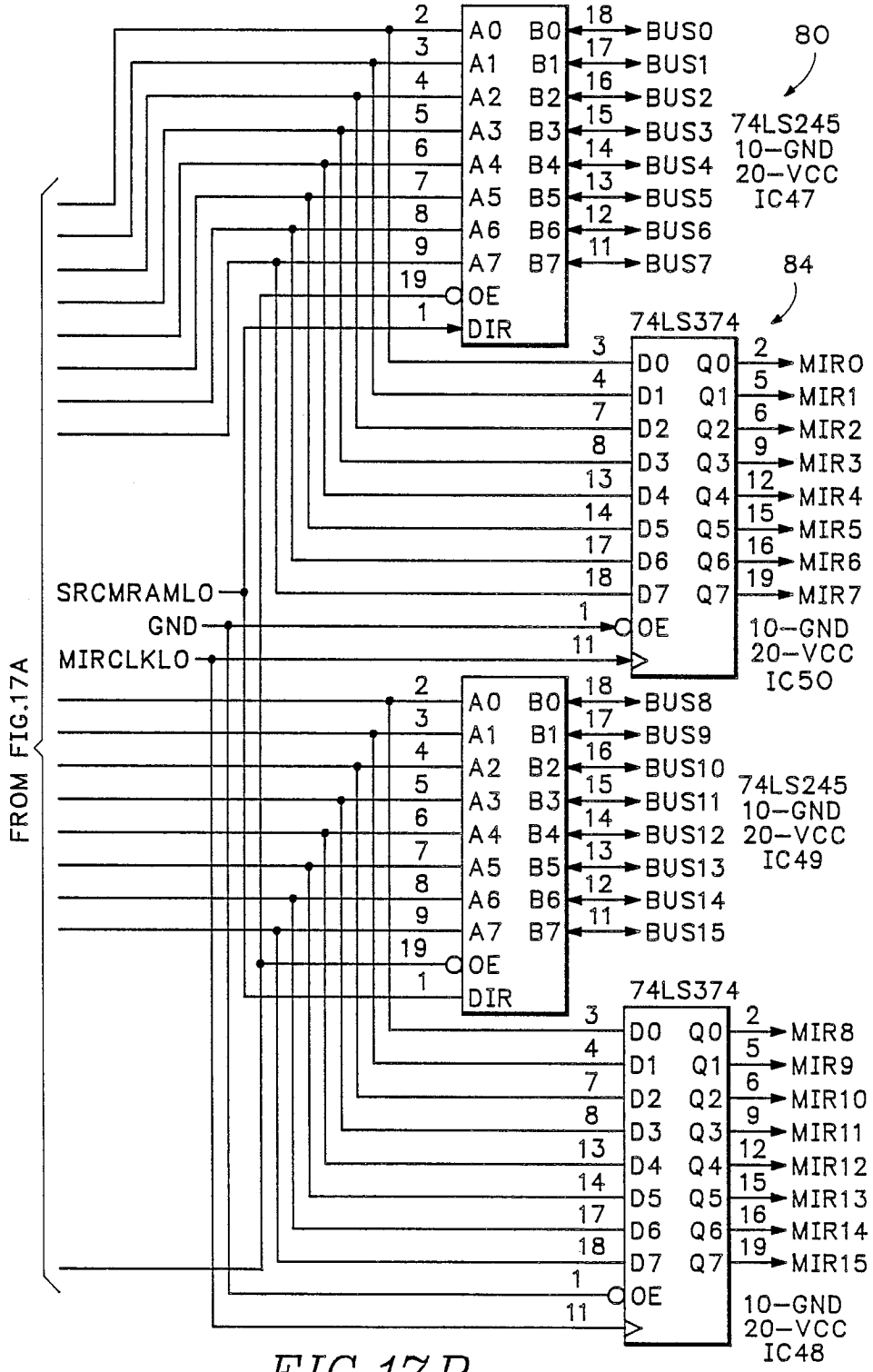

FROM FIG.17A
SRCMRAMLO
GND
MIRCLKLO
80
74LS245
10-GND
20-VCC
IC47
84
74LS374
74LS245
10-GND
20-VCC
IC49
74LS374
10-GND
20-VCC
IC50
10-GND
20-VCC
IC48
A0 B0
A1 B1
A2 B2
A3 B3
A4 B4
A5 B5
A6 B6
A7 B7
OE
DIR
BUS0
BUS1
BUS2
BUS3
BUS4
BUS5
BUS6
BUS7
BUS8
BUS9
BUS10
BUS11
BUS12
BUS13
BUS14
BUS15
D0 Q0
D1 Q1
D2 Q2
D3 Q3
D4 Q4
D5 Q5
D6 Q6
D7 Q7
MIR0
MIR1
MIR2
MIR3
MIR4
MIR5
MIR6
MIR7
MIR8
MIR9
MIR10
MIR11
MIR12
MIR13
MIR14
MIR15

FIG.17B

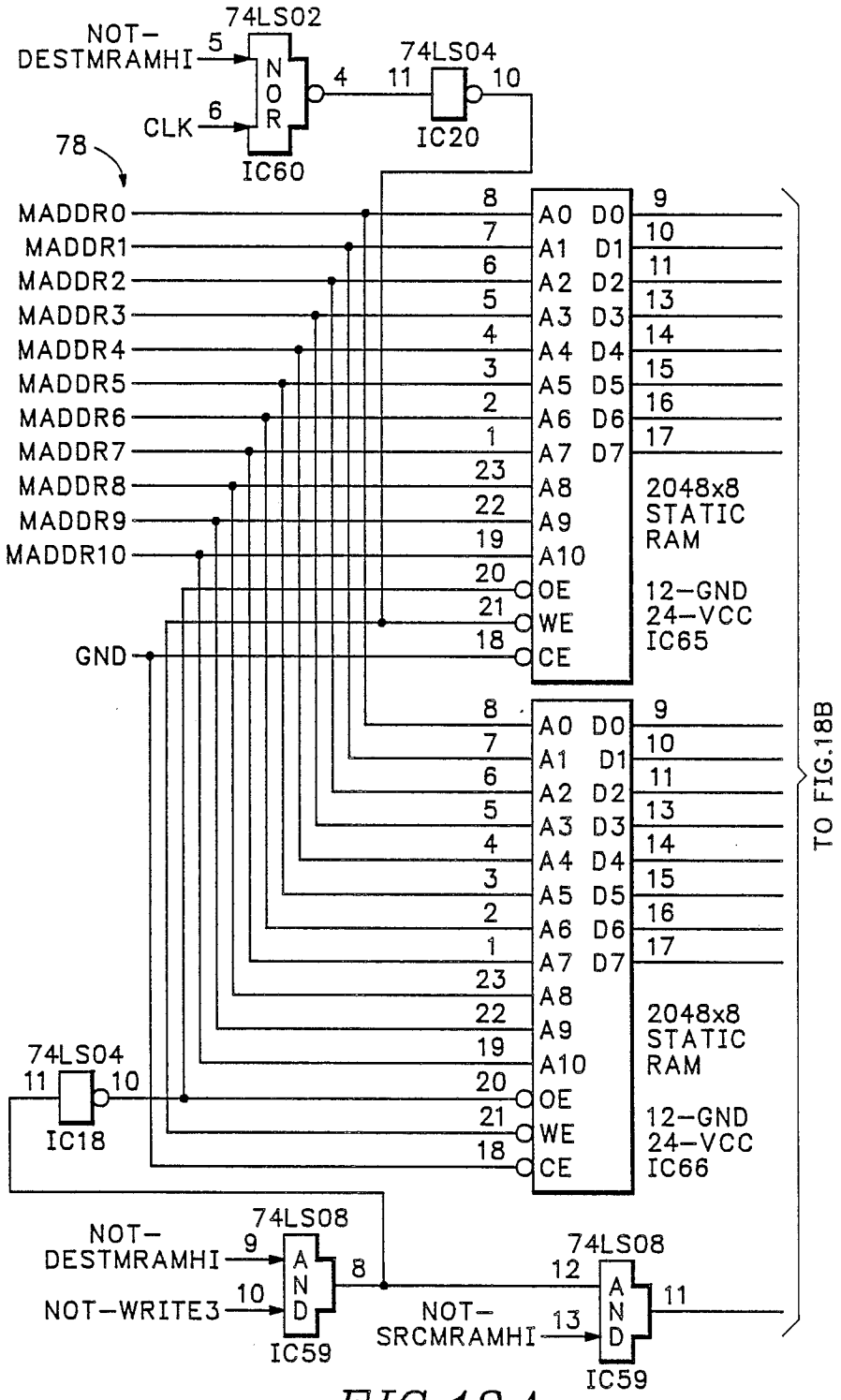

74LS02
NOT-
DESTMRAMHI
5
CLK
6
NOR
4
IC60
74LS04
11
10
IC20
78
MADDR0
MADDR1
MADDR2
MADDR3
MADDR4
MADDR5
MADDR6
MADDR7
MADDR8
MADDR9
MADDR10
GND
A0 D0
A1 D1
A2 D2
A3 D3
A4 D4
A5 D5
A6 D6
A7 D7
A8
A9
A10
OE
WE
CE
2048x8
STATIC
RAM
12-GND
24-VCC
IC65
IC66
TO FIG.18B
74LS04
IC18
74LS08
NOT-
DESTMRAMHI
NOT-WRITE3
AND
IC59
NOT-
SRCMRAMHI

FIG.18A

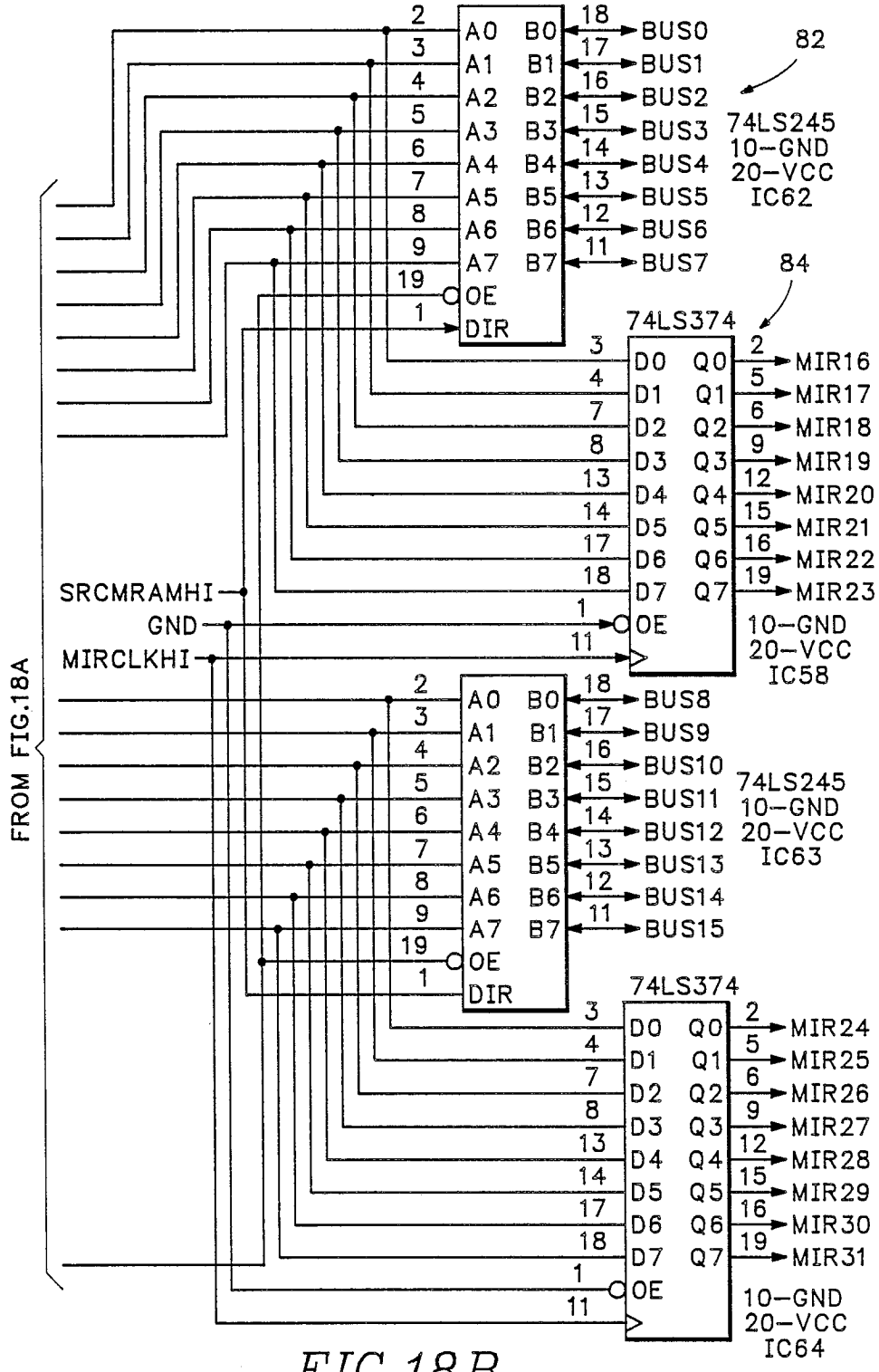

FROM FIG.18A
SRCMRAMHI
GND
MIRCLKHI
82
74LS245
10-GND
20-VCC
IC62
84
74LS374
IC58
74LS245
IC63
74LS374
IC64
A0 B0 BUS0
A1 B1 BUS1
A2 B2 BUS2
A3 B3 BUS3
A4 B4 BUS4
A5 B5 BUS5
A6 B6 BUS6
A7 B7 BUS7
OE
DIR
D0 Q0 MIR16
D1 Q1 MIR17
D2 Q2 MIR18
D3 Q3 MIR19
D4 Q4 MIR20
D5 Q5 MIR21
D6 Q6 MIR22
D7 Q7 MIR23
BUS8
BUS9
BUS10
BUS11
BUS12
BUS13
BUS14
BUS15
MIR24
MIR25
MIR26
MIR27
MIR28
MIR29
MIR30
MIR31

FIG.18B

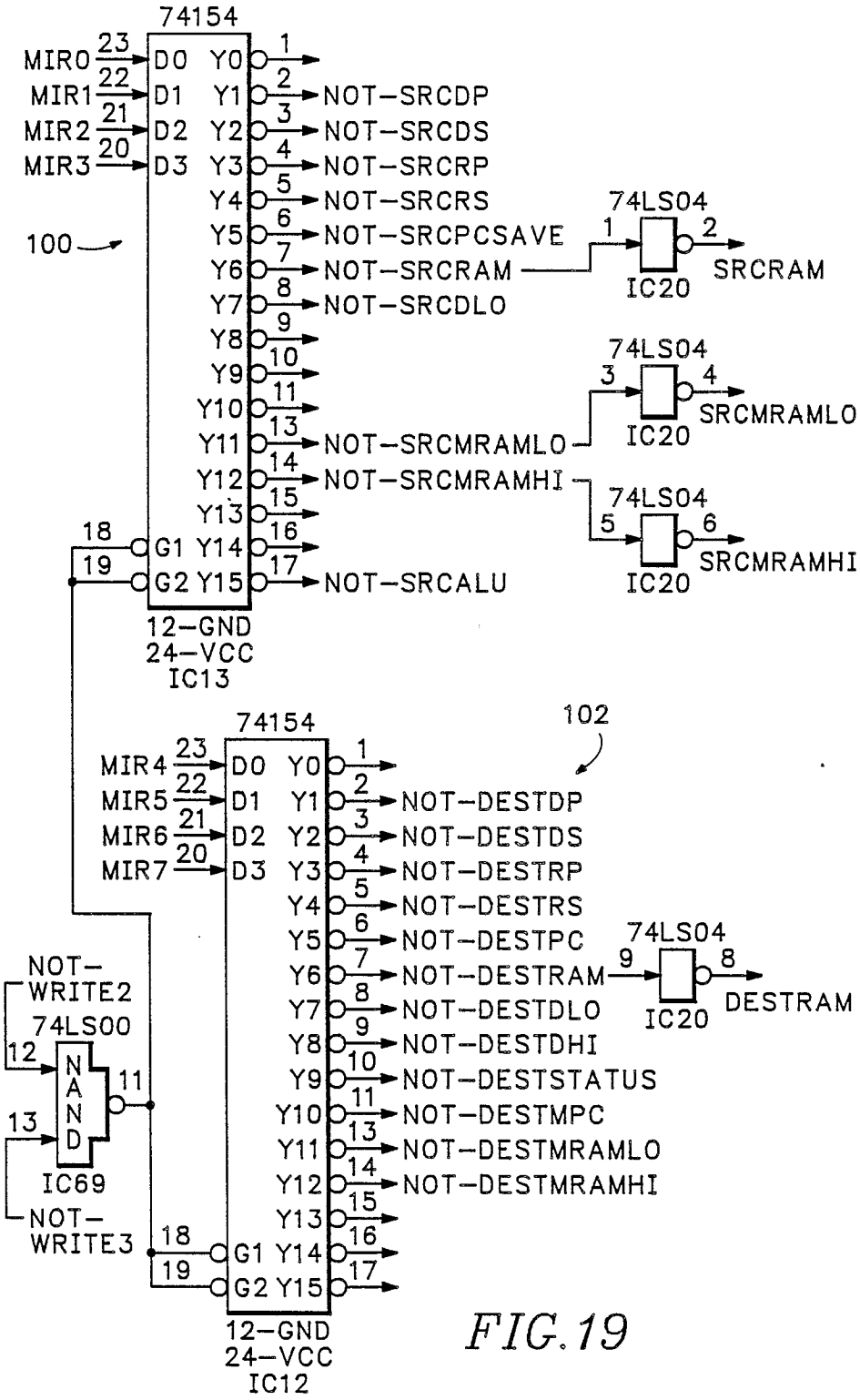

74154
MIR0
MIR1
MIR2
MIR3
D0
D1
D2
D3
Y0
Y1
Y2
Y3
Y4
Y5
Y6
Y7
Y8
Y9
Y10
Y11
Y12
Y13
Y14
Y15
G1
G2
NOT-SRCDP
NOT-SRCDS
NOT-SRCRP
NOT-SRCRS
NOT-SRCPCSAVE
NOT-SRCRAM
NOT-SRCDLO
NOT-SRCMRAMLO
NOT-SRCMRAMHI
NOT-SRCALU
100
12-GND
24-VCC
IC13
74LS04
IC20
SRCRAM
SRCMRAMLO
SRCMRAMHI
74154
102
MIR4
MIR5
MIR6
MIR7
NOT-DESTDP
NOT-DESTDS
NOT-DESTRP
NOT-DESTRS
NOT-DESTPC
NOT-DESTRAM
NOT-DESTDLO
NOT-DESTDHI
NOT-DESTSTATUS
NOT-DESTMPC
NOT-DESTMRAMLO
NOT-DESTMRAMHI
DESTRAM
NOT-WRITE2
74LS00
NAND
IC69
NOT-WRITE3
12-GND
24-VCC
IC12

FIG. 19

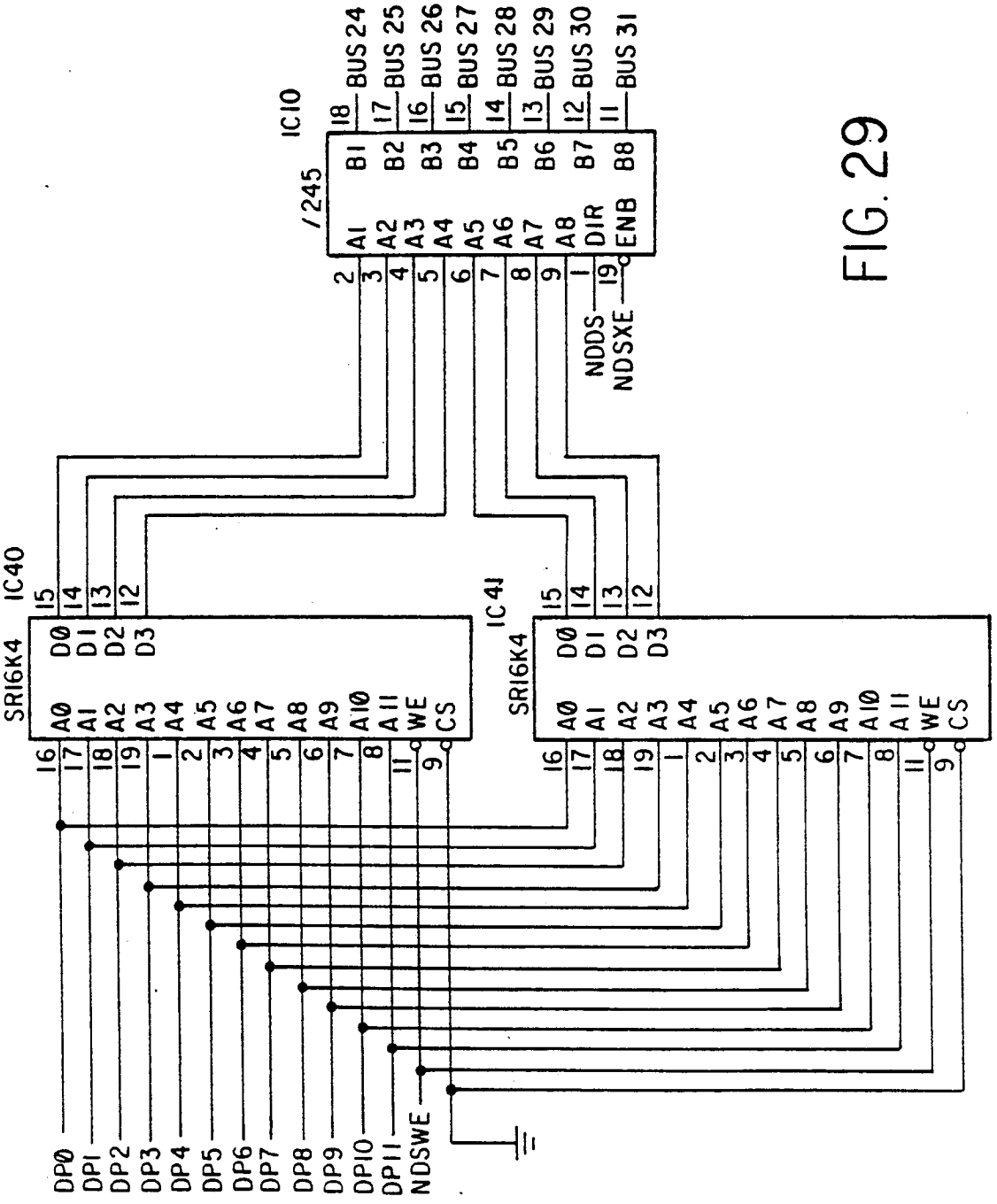

IC10
/245
BUS 24
BUS 25
BUS 26
BUS 27
BUS 28
BUS 29
BUS 30
BUS 31
NDDS
NDSXE
IC40
IC41
SR16K4
DP0
DP1
DP2
DP3
DP4
DP5
DP6
DP7
DP8
DP9
DP10
DP11
NDSWE

FIG. 29

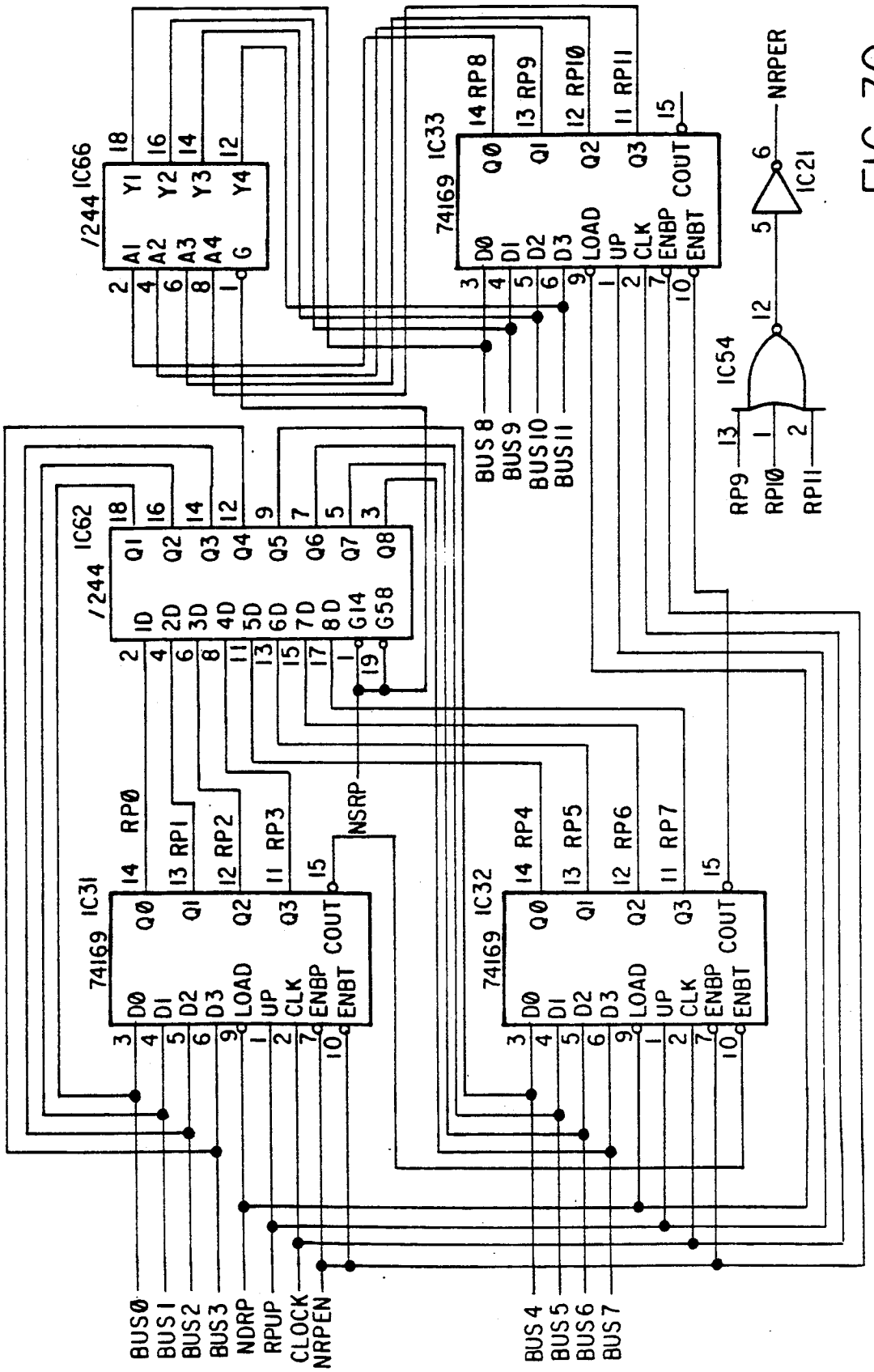

IC66
/244
A1 A2 A3 A4 G
Y1 Y2 Y3 Y4
IC33
74169
D0 D1 D2 D3 LOAD UP CLK ENBP ENBT
Q0 Q1 Q2 Q3 COUT
14 RP8
13 RP9
12 RP10
11 RP11
NRPER
IC21
IC54
RP9
RP10
RP11
BUS 8
BUS 9
BUS 10
BUS 11
IC62
/244
1D 2D 3D 4D 5D 6D 7D 8D G14 G58
Q1 Q2 Q3 Q4 Q5 Q6 Q7 Q8
IC31
74169
14 RP0
13 RP1
12 RP2
11 RP3
NSRP
IC32
14 RP4
13 RP5
12 RP6
11 RP7
BUS0
BUS1
BUS2
BUS3
NDRP
RPUP
CLOCK
NRPEN
BUS 4
BUS 5
BUS 6
BUS 7

FIG. 30

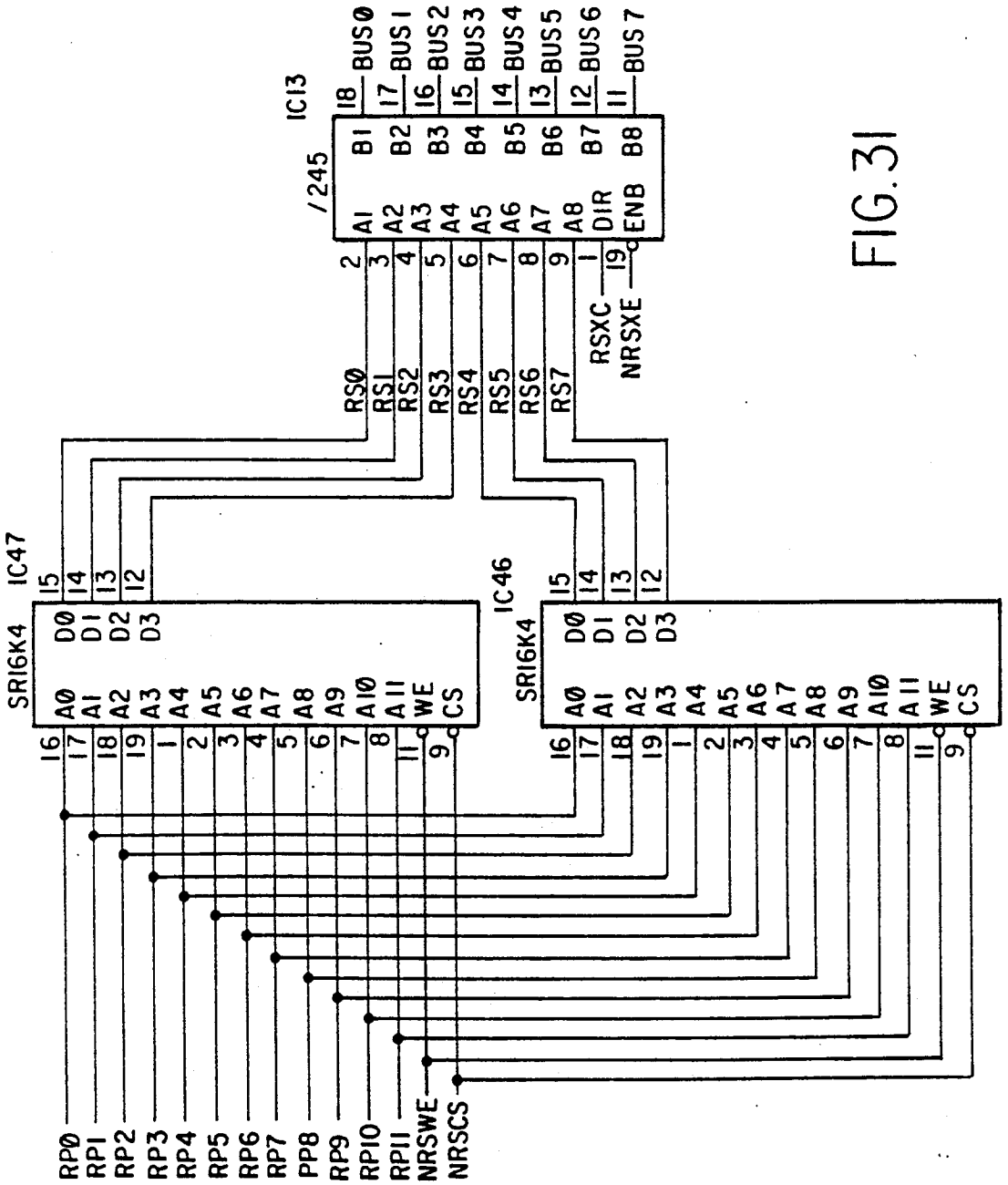

IC13
/245
BUS0
BUS1
BUS2
BUS3
BUS4
BUS5
BUS6
BUS7
RSXC
NRSXE
RS0
RS1
RS2
RS3
RS4
RS5
RS6
RS7
IC47
IC46
SR16K4
RP0
RP1
RP2
RP3
RP4
RP5
RP6
RP7
PP8
RP9
RP10
RP11
NRSWE
NRSCS

FIG. 31

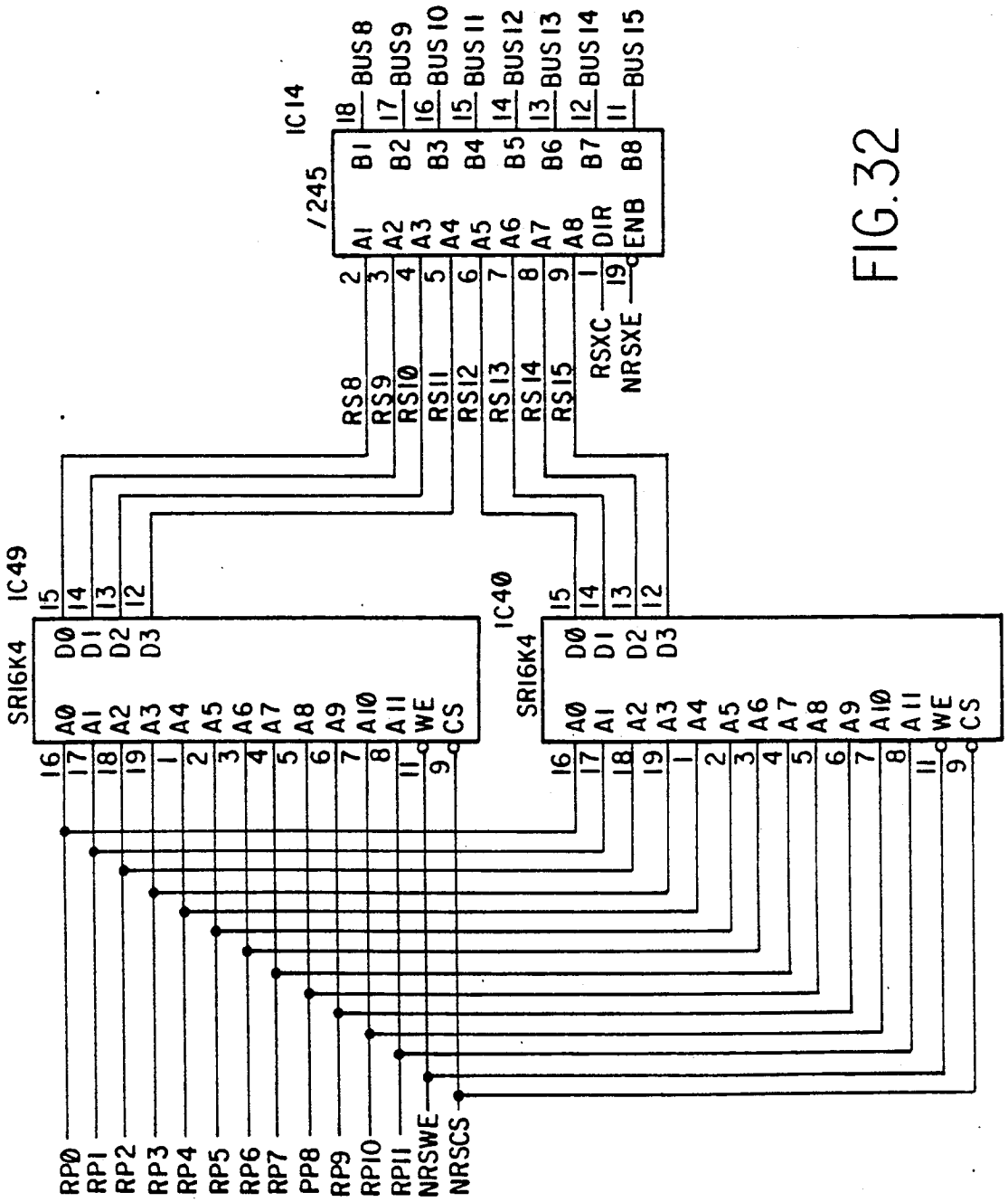

IC14
/245
BUS8
BUS9
BUS10
BUS11
BUS12
BUS13
BUS14
BUS15
RS8
RS9
RS10
RS11
RS12
RS13
RS14
RS15
RSXC
NRSXE
IC49
IC40
SR16K4
RP0
RP1
RP2
RP3
RP4
RP5
RP6
RP7
PP8
RP9
RP10
RP11
NRSWE
NRSCS

FIG. 32

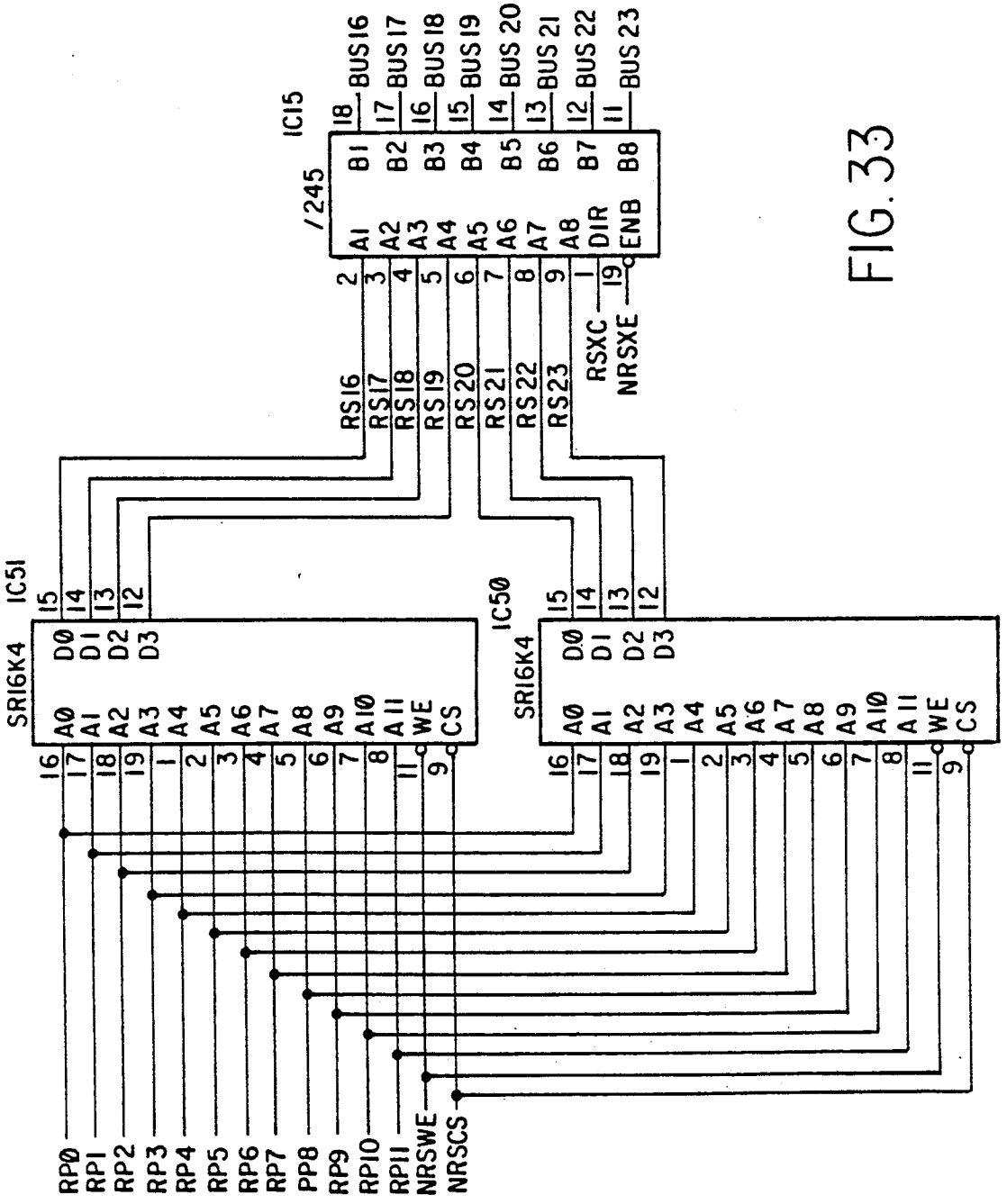

IC15
/245
BUS16
BUS17
BUS18
BUS19
BUS 20
BUS 21
BUS 22
BUS 23
RS16
RS17
RS18
RS19
RS20
RS21
RS22
RS23
RSXC
NRSXE
IC51
IC50
SR16K4
RP0
RP1
RP2
RP3
RP4
RP5
RP6
RP7
PP8
RP9
RP10
RP11
NRSWE
NRSCS

FIG. 33

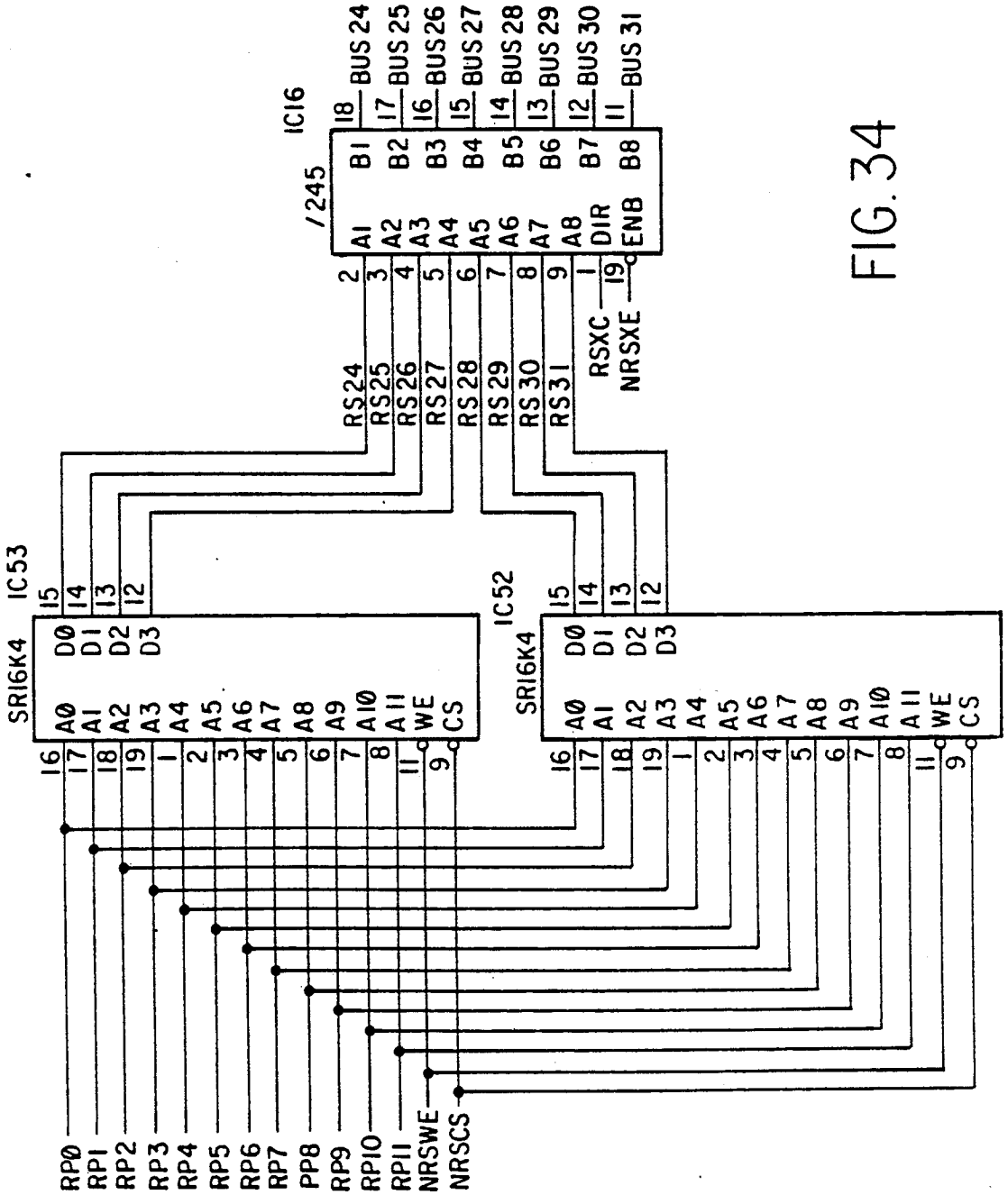

IC16
/245
A1 A2 A3 A4 A5 A6 A7 A8 DIR ENB
B1 B2 B3 B4 B5 B6 B7 B8
BUS 24 BUS 25 BUS 26 BUS 27 BUS 28 BUS 29 BUS 30 BUS 31
RS24 RS25 RS26 RS27 RS28 RS29 RS30 RS31
RSXC
NRSXE
IC53
IC52
SRI6K4
D0 D1 D2 D3
A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 A10 A11 WE CS
RP0 RP1 RP2 RP3 RP4 RP5 RP6 RP7 PP8 RP9 RP10 RP11 NRSWE NRSCS

FIG. 34

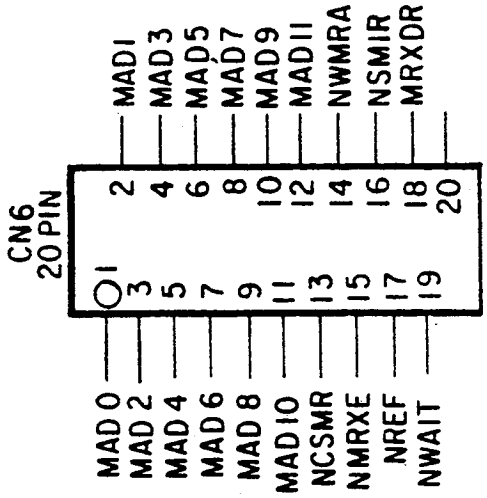

CN6
20 PIN
MAD 0
MAD 2
MAD 4
MAD 6
MAD 8
MAD 10
NCSMR
NMRXE
NREF
NWAIT
MAD1
MAD3
MAD5
MAD7
MAD9
MAD11
NWMRA
NSMIR
MRXDR

FIG. 35


CN1
40 PIN
BUS 0
BUS 2
BUS 4
BUS 6
BUS 8
BUS 10
BUS 12
BUS 14
BUS 16
BUS 18
BUS 20
BUS 22
BUS 24
BUS 26
BUS 28
BUS 30
BUS 1
BUS 3
BUS 5
BUS 7
BUS 9
BUS 11
BUS 13
BUS 15
BUS 17
BUS 19
BUS 21
BUS 23
BUS 25
BUS 27
BUS 29
BUS 31
MIRCK
FASTC
XCLK
OSC

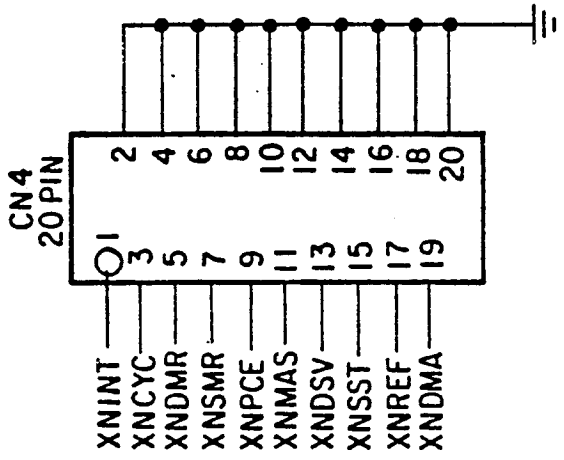
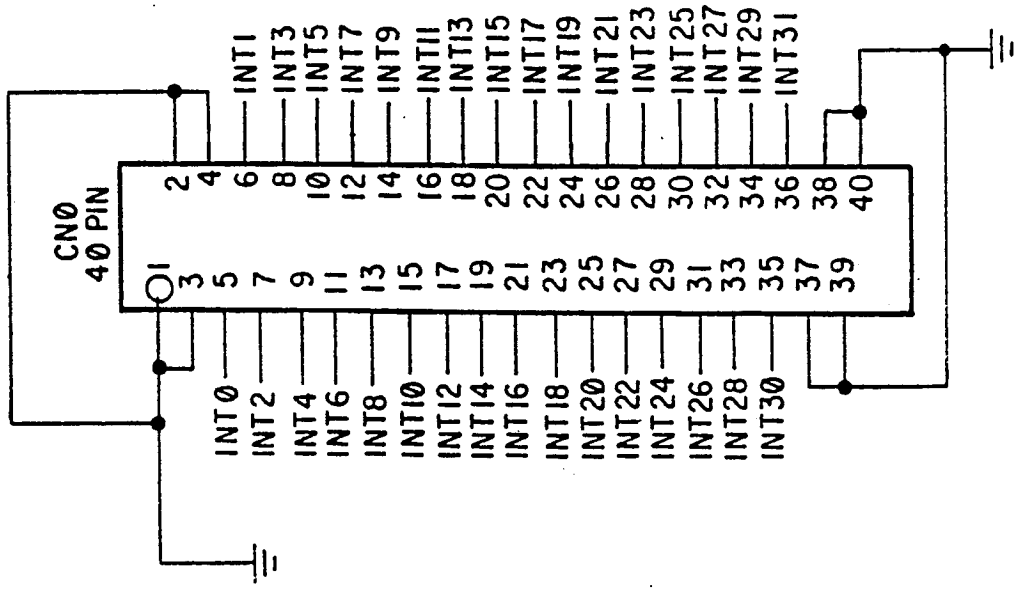

CN4
20 PIN
1
3
5
7
9
11
13
15
17
19
2
4
6
8
10
12
14
16
18
20
XNINT
XNCYC
XNDMR
XNSMR
XNPCE
XNMAS
XNDSV
XNSST
XNREF
XNDMA

FIG. 36


CN0
40 PIN
1
3
5
7
9
11
13
15
17
19
21
23
25
27
29
31
33
35
37
39
2
4
6
8
10
12
14
16
18
20
22
24
26
28
30
32
34
36
38
40
INT0
INT2
INT4
INT6
INT8
INT10
INT12
INT14
INT16
INT18
INT20
INT22
INT24
INT26
INT28
INT30
INT1
INT3
INT5
INT7
INT9
INT11
INT13
INT15
INT17
INT19
INT21
INT23
INT25
INT27
INT29
INT31

since it takes a full clock cycle for the effects to flow through the microcode pre-fetch pipe. (3) The DECODE micro-operation, which conditionally sets MPC **76**, must also be used in the next-to-last microinstruction in a microcoded operation. This limits the minimum microcoded operation length to 2 clock cycles.

Also, the microassembler forces a JMP=000 micro-operation whenever the END micro-operation is used. This ensures that the 0 location of the page for a microcoded operation is the first microinstruction executed.

Board Interconnection

FIGS. **20** and **21** identify two ribbon cables which are used to make connections between the two circuit boards. In particular FIG. **20** identifies a 24-connector ribbon cable **104**, which connects from the baseboard side, shown on the left, to the expansion board side, shown on the right of the figure. FIG. **21** shows the use of a 16-connector ribbon cable **106** for making similar connections between the two boards. FIG. **22** shows the preferred integrated circuit layout on a base board **108**, while FIG. **23** shows the corresponding integrated circuit layout on an expansion board **110**. The numbers on each of the illustrated integrated circuit outlines identify the integrated circuit numbers shown in FIGS. **2–21**.

SYSTEM SOFTWARE

Computer **30** in this preferred embodiment uses various software packages, including a FORTH kernel, a cross-compiler, a microassembler, as well as microcode. The software for these packages. Written using MVP-FORTH, are listed in Appendix A. Further, the microcode format is defined in Appendix B as Table **1***a*–**1***d*. Some general comments about the software are in order.

The Cross-Compiler

The cross-compiler maintains a sealed vocabulary with all the words currently defined for computer **30**. At the base of this dictionary are special cross-compiler words such as IF ELSE THEN : and ; . After cross-compilation has started, words are added to this sealed vocabulary and are also cross-compiled into computer **30**. Whenever the keyword CROSS-COMPILER is used, any word definitions constants, variables, etc. will be compiled to computer **30**. However, any immediate operations will be taken from the cross-compiler s vocabulary, which is chained to the normal MVP-FORTH vocabulary.

By entering the FORTH word {, the cross-compiler enters the immediate execution mode for computer **30**. All words are searched for in the sealed vocabulary for computer **30** and executed by computer **30** itself. The "START . . . " "END" that is displayed indicates the start and the end of execution of computer **30**. If the execution freezes in between the start and end, that means that computer **30** is hung up. The cross-compiler builds a special FORTH word in computer **30** to execute the desired definition, then perform a HALT instruction. Entering the FORTH word } will leave the computer **30** mode of execution and return to the cross-compiler. No colon definitions or other creation of dictionary entries should be performed while between { and }.

The FORTH word BOARD will automatically transfer control of the system to computer **30** via its COLD command. The host MVP-FORTH will then execute an idle loop waiting for computer **30** to request services. The word BYE will return control back to the host's MVP-FORTH.

The current cross-compiler can not keep track of DP, etc., in computer **30** if it is out of sync with the cross-compiler's copy. This means that no cross-compiling or microassembly may be done after the FORTH of computer **30** has altered the dictionary in any way. This could be fixed at a later date by updating the cross-compiler's variables from computer **30** after every BYE command to computer **30**.

Cross-compiled code should be kept to a minimum, since it is tricky to write. After a bare minimum kernel is up and running, computer **30** should do all further FORTH compilation.

The Microassembler

The microassembler is a tool to save the programmer from having to set all the bits for microcode by hand. It allows the use of mnemonics for setting the micro-operation fields in a microinstruction, and, for the most part, automatically handles the microinstruction addressing scheme.

The microassembler is written to be co-resident with the cross-compiler. It uses the same routines for computer **30** and sealed host vocabulary dictionary handling, etc. Currently all microcode must be defined before the board starts altering its dictionary, but this could be changed as discussed above.

In the terminology used here, a microinstruction is a 32-bit instruction in microcode. while a micro-operation is formed by one or more microcode fields within a single microinstruction.

Tables **1***a*–**1***d* in Appendix B give a quick reference to all the hardware-defined microinstruction fields supported by the microassembler. Since the microcode layout is very horizontal, you can find a direct relationship between bit settings and control line inputs to various chips on the board. The fields in the 32-bit microinstruction format will be explained by discussing examples from the kernel's microcode. As with most horizontally microcoded machines, as many micro-operations as desired may take place at the same time, although some operations don't do anything useful when used together.

Microcode Definition Format

The microassembler has a few keywords to make life easier for the microprogrammer. The Word OP-CODE: starts a microcode definition. The input parameter is the page number from O - OFF hex that the op-code resides in. For example, the word ± is op-code **7**. This means that whenever computer **30** interprets a hex FF07 as an op-code, the word ± will be executed in microcode. The character string after OP-CODE: is the name of the op-code that will be added to the cross-compiler and computer **30** dictionaries. It is the programmer's responsibility to insure that he does not assign two op-codes to the same micromemory page.

The variable CURRENT-PAGE contains the page currently assigned by OP-CODE: It may be changed to facilitate multi-page definitions. See MPC control below.

The word :: signifies the start of the definition of a microinstruction. The number before :: must be from 0 to 7, and signifies the offset from 0 to 7 within the current micromemory page for that microinstruction. Microinstructions may be defined in any order desired.

The word ;; signifies the end of a microinstruction and stores the microinstruction into the appropriate location in micromemory.

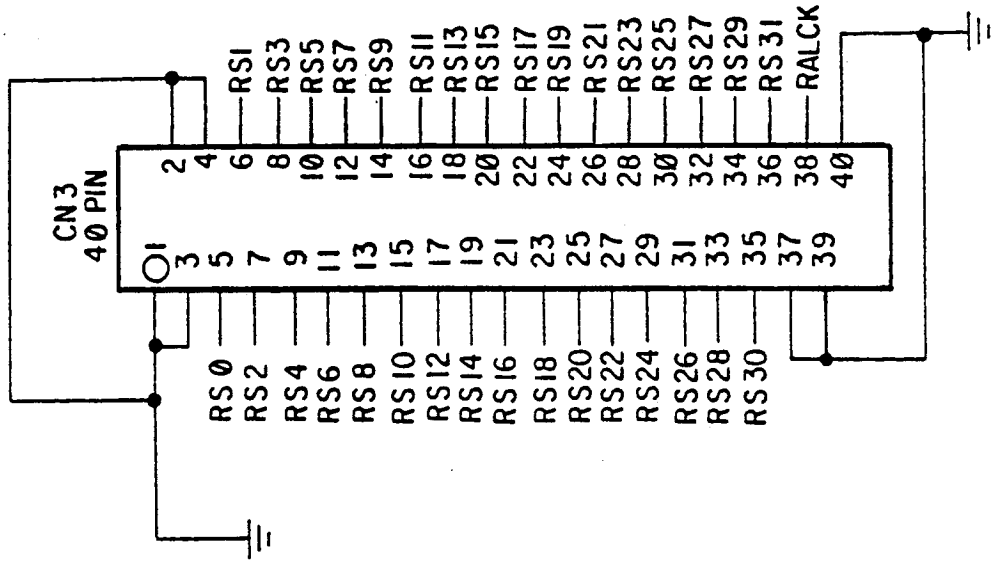

CN3
40 PIN
RS0
RS1
RS2
RS3
RS4
RS5
RS6
RS7
RS8
RS9
RS10
RS11
RS12
RS13
RS14
RS15
RS16
RS17
RS18
RS19
RS20
RS21
RS22
RS23
RS24
RS25
RS26
RS27
RS28
RS29
RS30
RS31
RALCK

FIG. 39

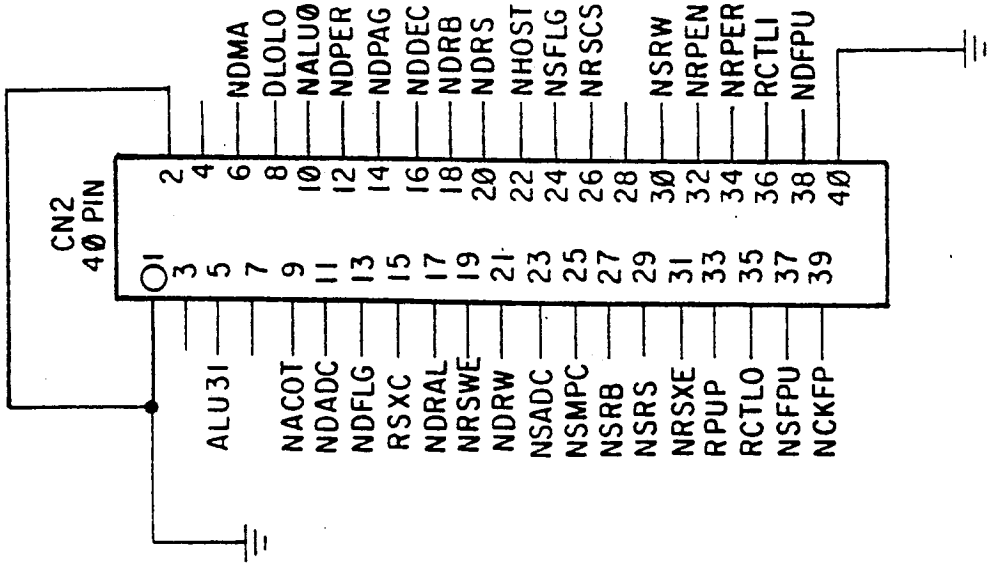

CN2
40 PIN
ALU3I
NACOT
NDADC
NDFLG
RSXC
NDRAL
NRSWE
NDRW
NSADC
NSMPC
NSRB
NSRS
NRSXE
RPUP
RCTLO
NSFPU
NCKFP
NDMA
DLOLO
NALU0
NDPER
NDPAG
NDDEC
NDRB
NDRS
NHOST
NSFLG
NRSCS
NSRW
NRPEN
NRPER
RCTLI
NDFPU

FIG. 38

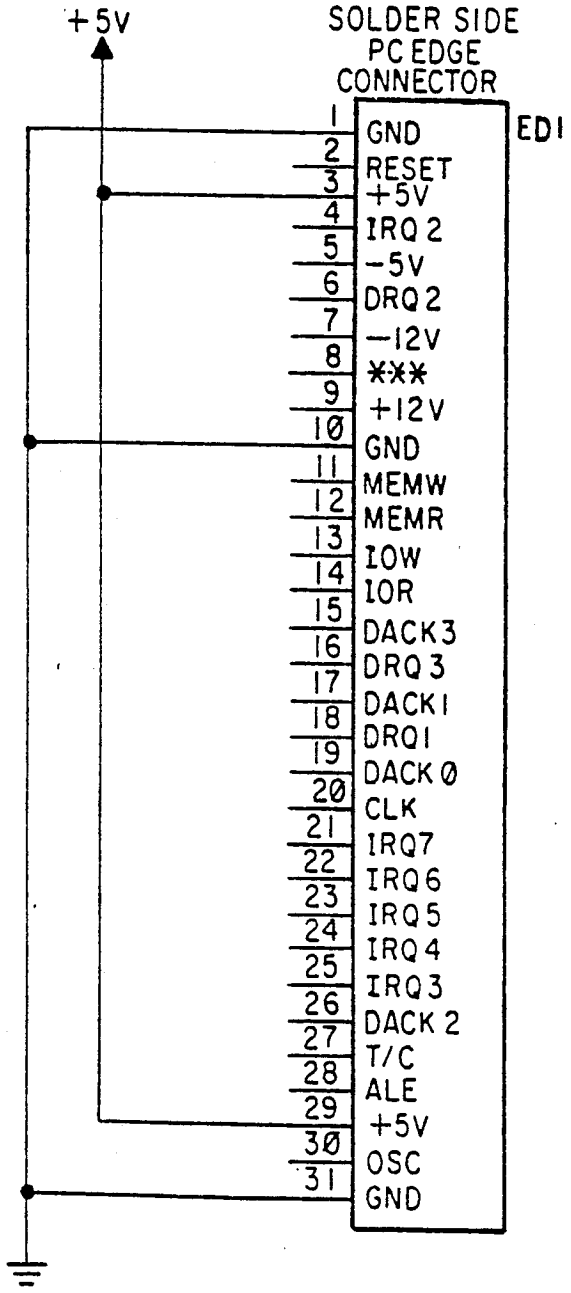

+5V
SOLDER SIDE
PC EDGE
CONNECTOR
ED1
1 GND
2 RESET
3 +5V
4 IRQ 2
5 -5V
6 DRQ 2
7 -12V
8 ***
9 +12V
10 GND
11 MEMW
12 MEMR
13 IOW
14 IOR
15 DACK3
16 DRQ 3
17 DACK1
18 DRQ1
19 DACK 0
20 CLK
21 IRQ7
22 IRQ6
23 IRQ5
24 IRQ4
25 IRQ3
26 DACK 2
27 T/C
28 ALE
29 +5V
30 OSC
31 GND

FIG. 40

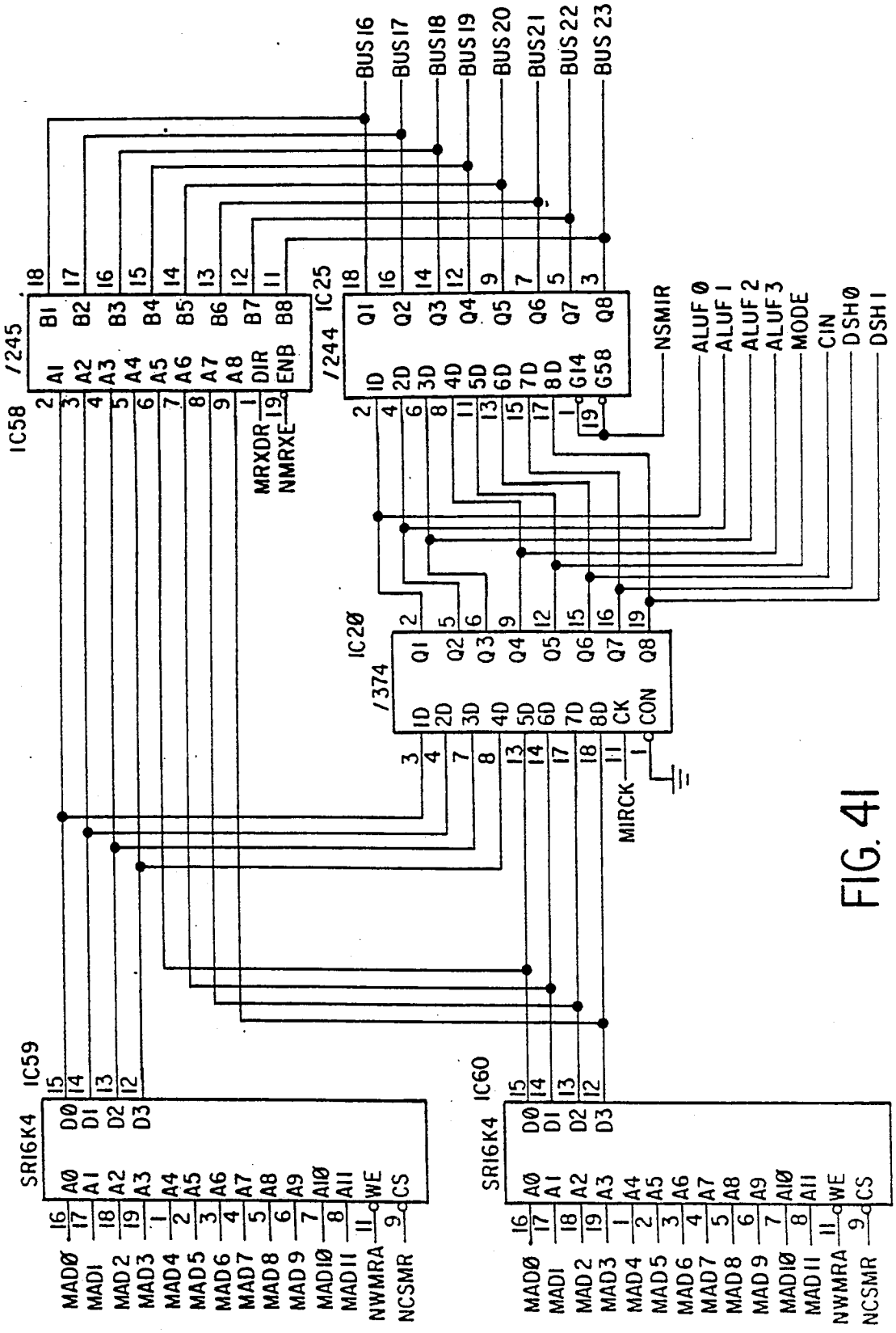

IC58
/245
IC25
/244
IC20
/374
IC59
SR16K4
IC60
SR16K4
BUS16
BUS17
BUS18
BUS19
BUS20
BUS21
BUS22
BUS23
MRXDR
NMRXE
NSMIR
ALUF0
ALUF1
ALUF2
ALUF3
MODE
CIN
DSH0
DSH1
MIRCK
MAD0
MAD1
MAD2
MAD3
MAD4
MAD5
MAD6
MAD7
MAD8
MAD9
MAD10
MAD11
NWMRA
NCSMR

FIG. 41

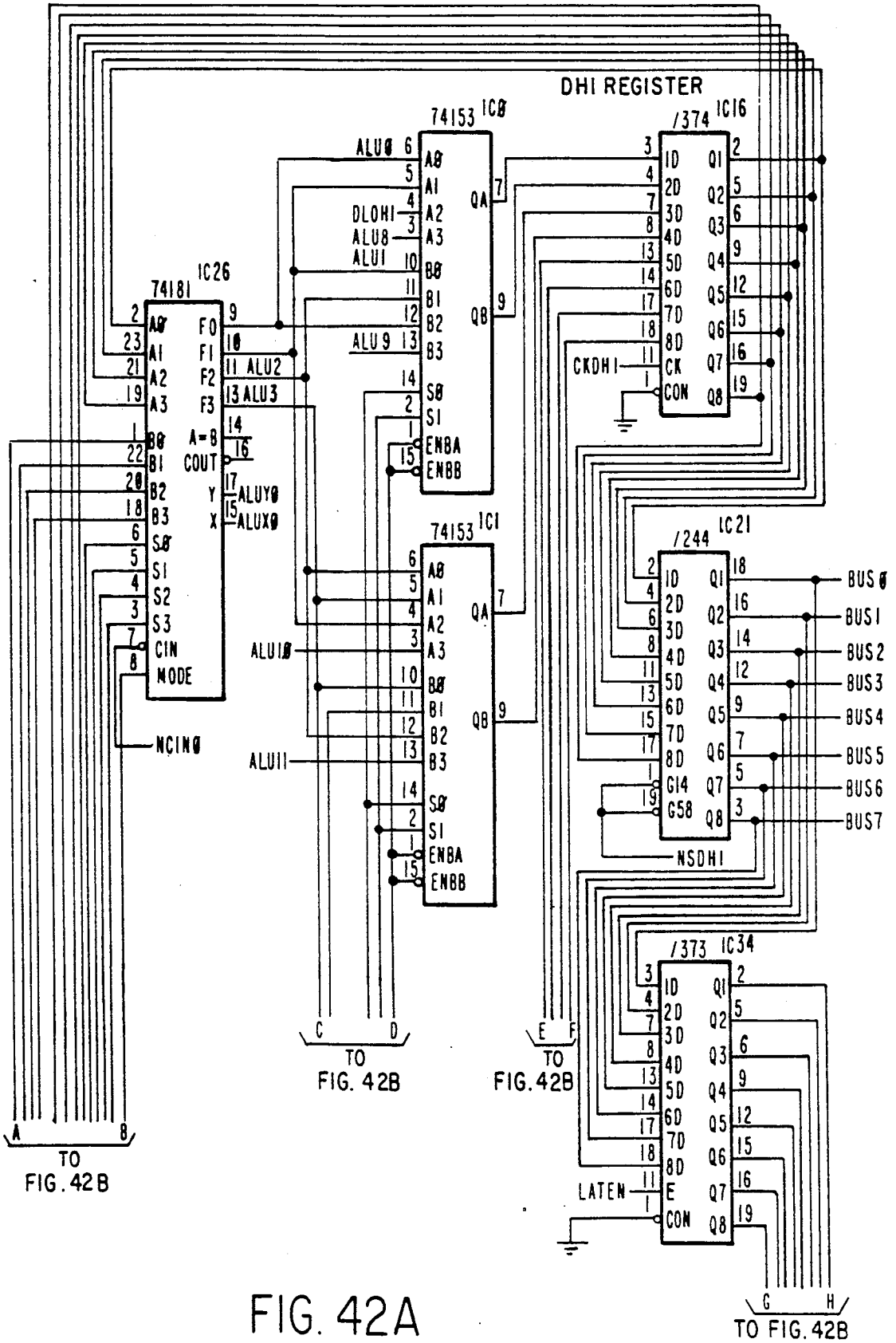

DHI REGISTER
74153 IC8
/374 IC16
74181 IC26
74153 IC1
/244 IC21
/373 IC34
ALU8
DLOHI
ALU8
ALU1
ALU9
ALU2
ALU3
ALU0
ALU1
NCIN0
CKDH1
NSDH1
LATEN
BUS0
BUS1
BUS2
BUS3
BUS4
BUS5
BUS6
BUS7
TO FIG. 42B
TO FIG. 42B
TO FIG. 42B
TO FIG. 42B

FIG. 42A

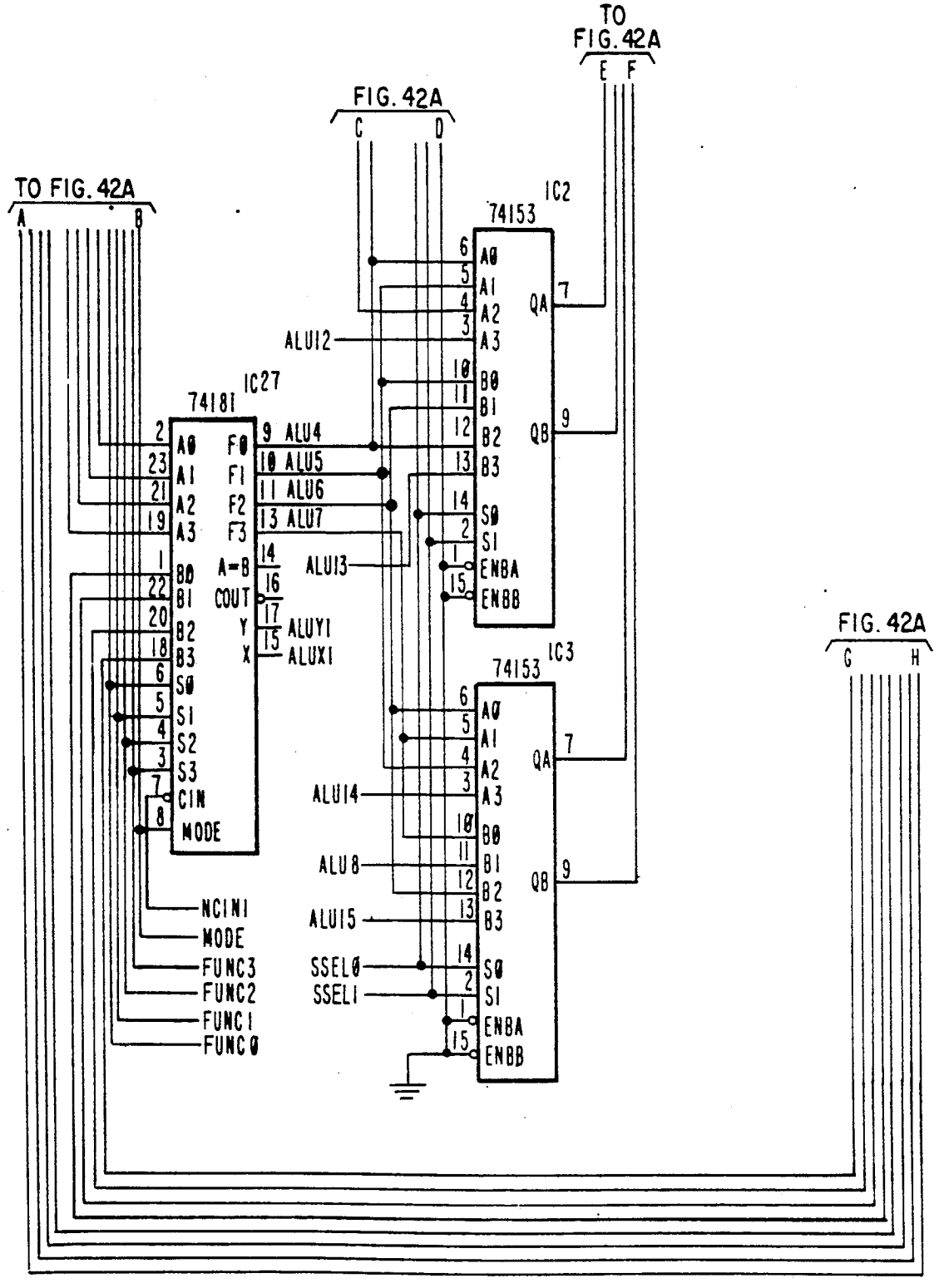

TO FIG. 42A
E F
FIG. 42A
C D
TO FIG. 42A
A B
IC2
74153
IC27
74181
ALU12
ALU4
ALU5
ALU6
ALU7
ALU13
ALUY1
ALUX1
FIG. 42A
G H
IC3
74153
ALU14
ALU8
ALU15
SSEL0
SSEL1
NCIN1
MODE
FUNC3
FUNC2
FUNC1
FUNC0

FIG. 42B

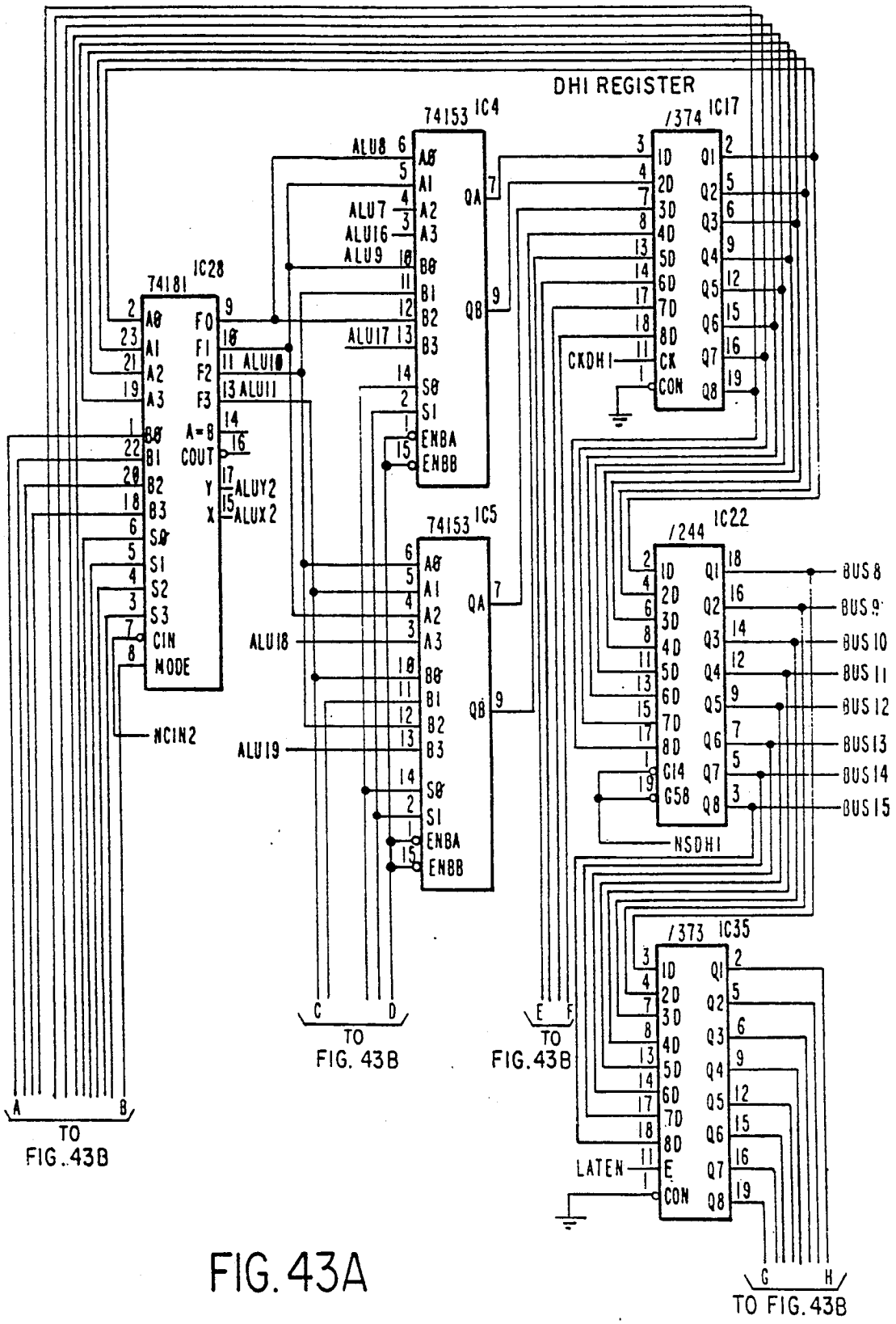

DHI REGISTER
74153 IC4
/374 IC17
74181 IC28
74153 IC5
/244 IC22
/373 IC35
ALU8
ALU7
ALU16
ALU9
ALU17
ALU18
ALU19
ALU10
ALU11
ALUY2
ALUX2
NCIN2
CKDHI
LATEN
NSDHI
BUS 8
BUS 9
BUS 10
BUS 11
BUS 12
BUS 13
BUS 14
BUS 15
TO FIG. 43B
TO FIG. 43B
TO FIG. 43B
TO FIG. 43B

FIG. 43A

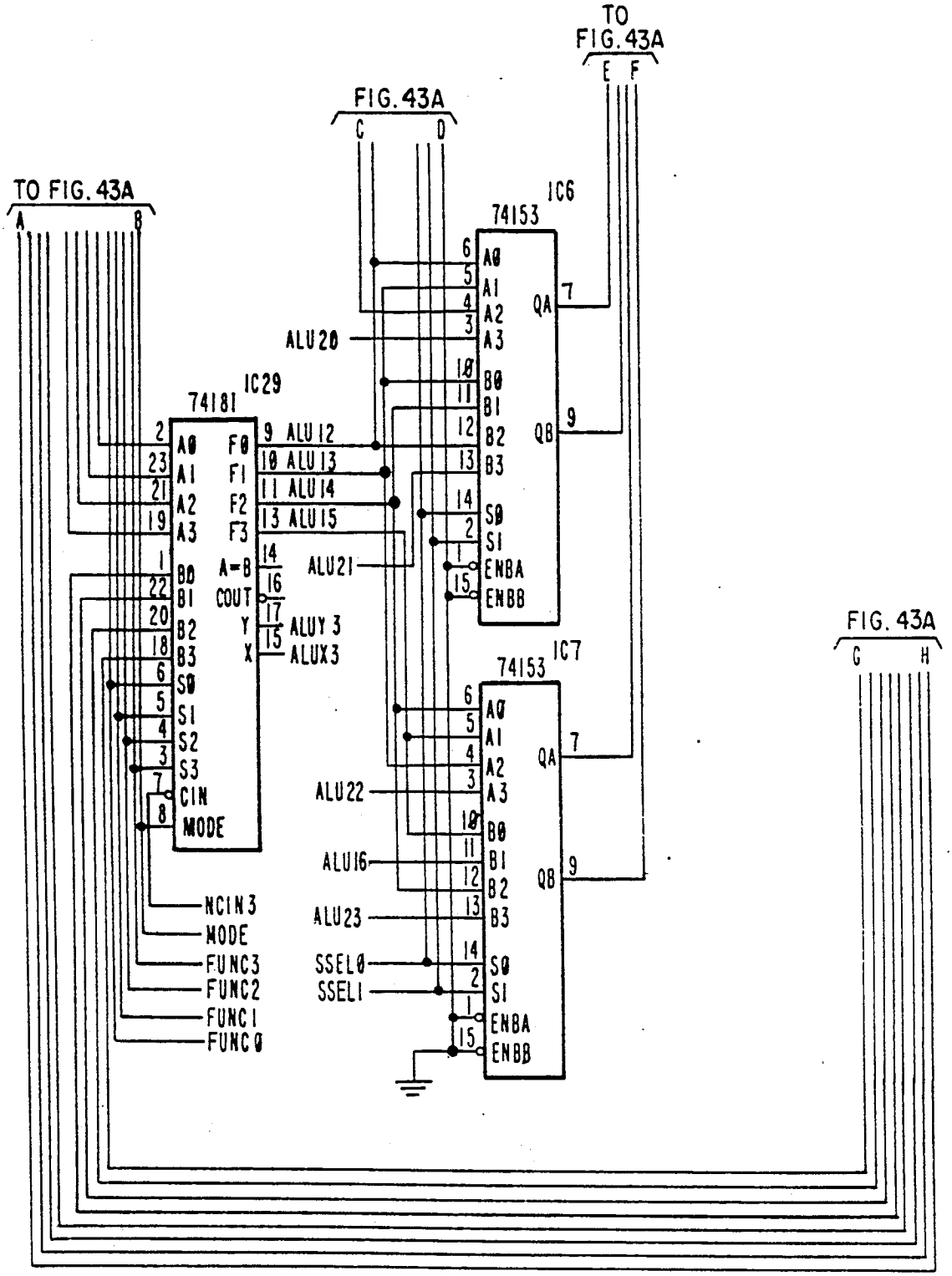

TO FIG. 43A
E F
FIG. 43A
C D
TO FIG. 43A
A B
IC6
74153
A0
A1
A2
A3
QA
ALU20
B0
B1
B2
B3
QB
S0
S1
ENBA
ENBB
IC29
74181
A0
A1
A2
A3
B0
B1
B2
B3
S0
S1
S2
S3
CIN
MODE
F0
F1
F2
F3
A=B
COUT
Y
X
ALU12
ALU13
ALU14
ALU15
ALU21
ALUY3
ALUX3
IC7
74153
ALU22
ALU16
ALU23
SSEL0
SSEL1
FIG. 43A
G H
NCIN3
MODE
FUNC3
FUNC2
FUNC1
FUNC0

FIG. 43B

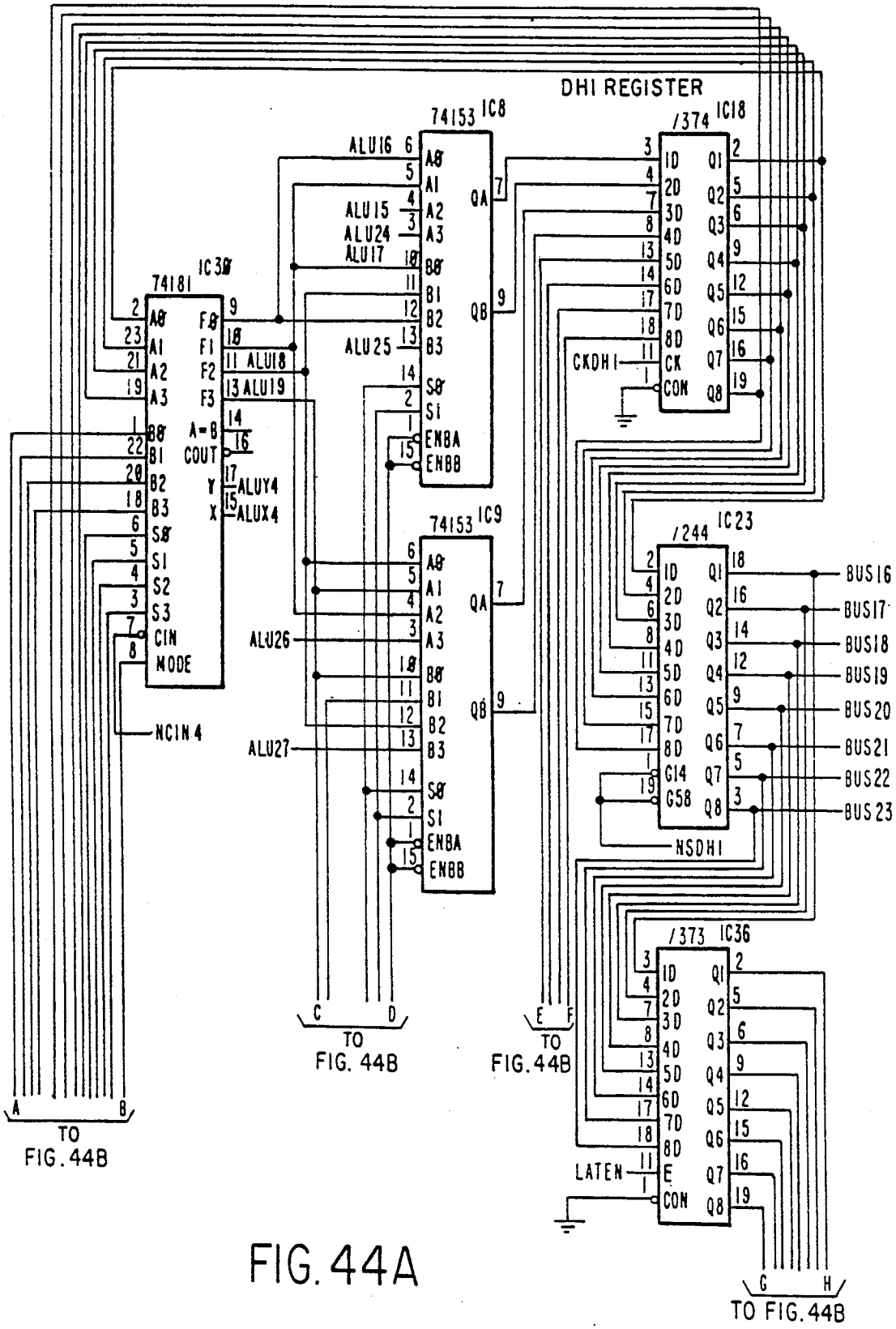

DHI REGISTER
74153 IC8
74153 IC9
74181 IC38
/374 IC18
/244 IC23
/373 IC36
ALU16
ALU15
ALU24
ALU17
ALU25
ALU26
ALU27
ALU18
ALU19
ALUY4
ALUX4
NCIN4
CKDHI
NSDHI
LATEN
BUS16
BUS17
BUS18
BUS19
BUS20
BUS21
BUS22
BUS23
TO FIG. 44B
TO FIG. 44B
TO FIG. 44B
TO FIG. 44B

FIG. 44A

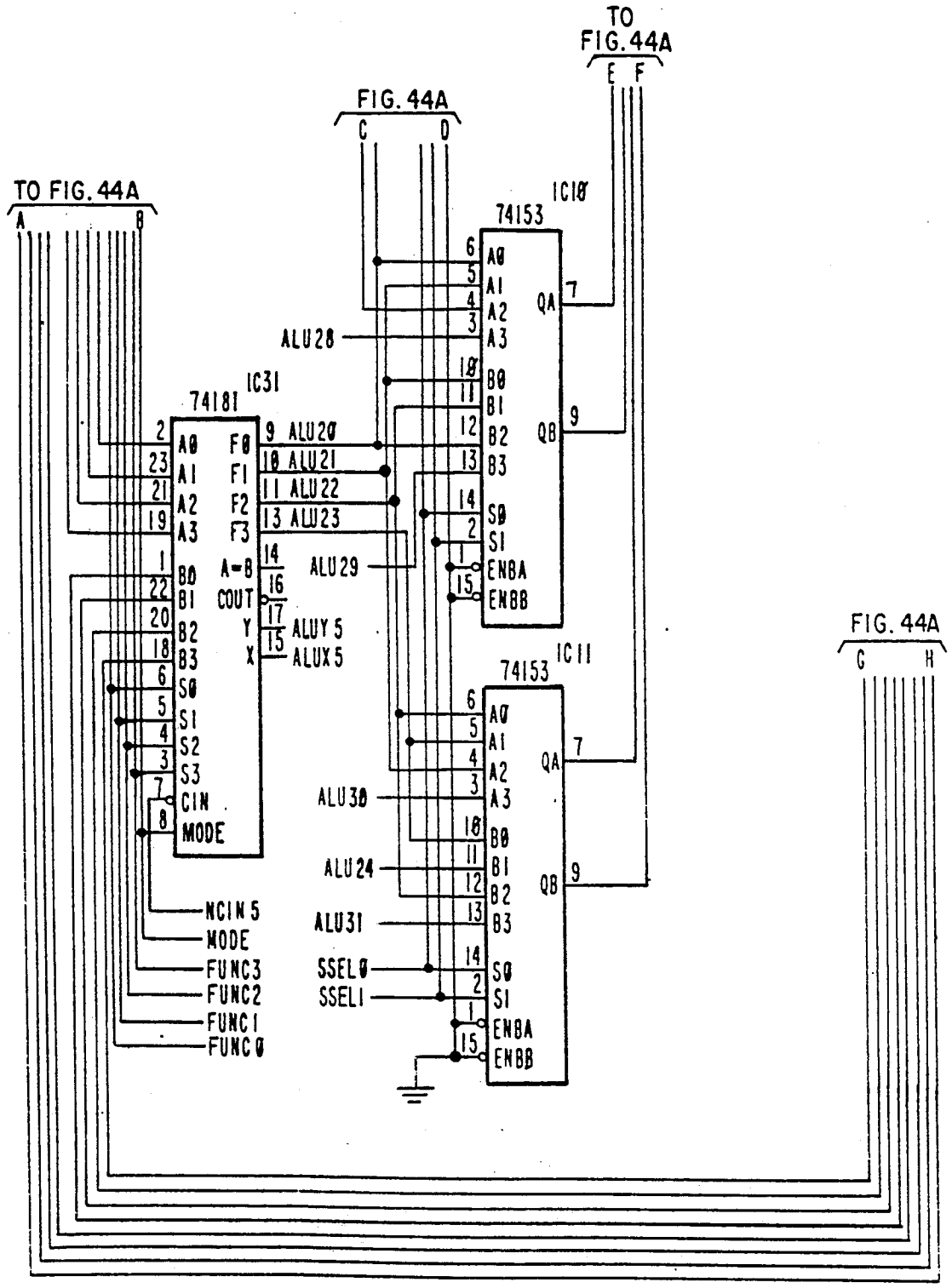

TO FIG. 44A
E F
FIG. 44A
C D
TO FIG. 44A
A B
IC18
74153
IC31
74181
ALU28
ALU29
ALU20
ALU21
ALU22
ALU23
ALUY 5
ALUX 5
IC11
74153
ALU30
ALU24
ALU31
SSEL0
SSEL1
NCIN 5
MODE
FUNC3
FUNC2
FUNC1
FUNC0
FIG. 44A
G H

FIG. 44B

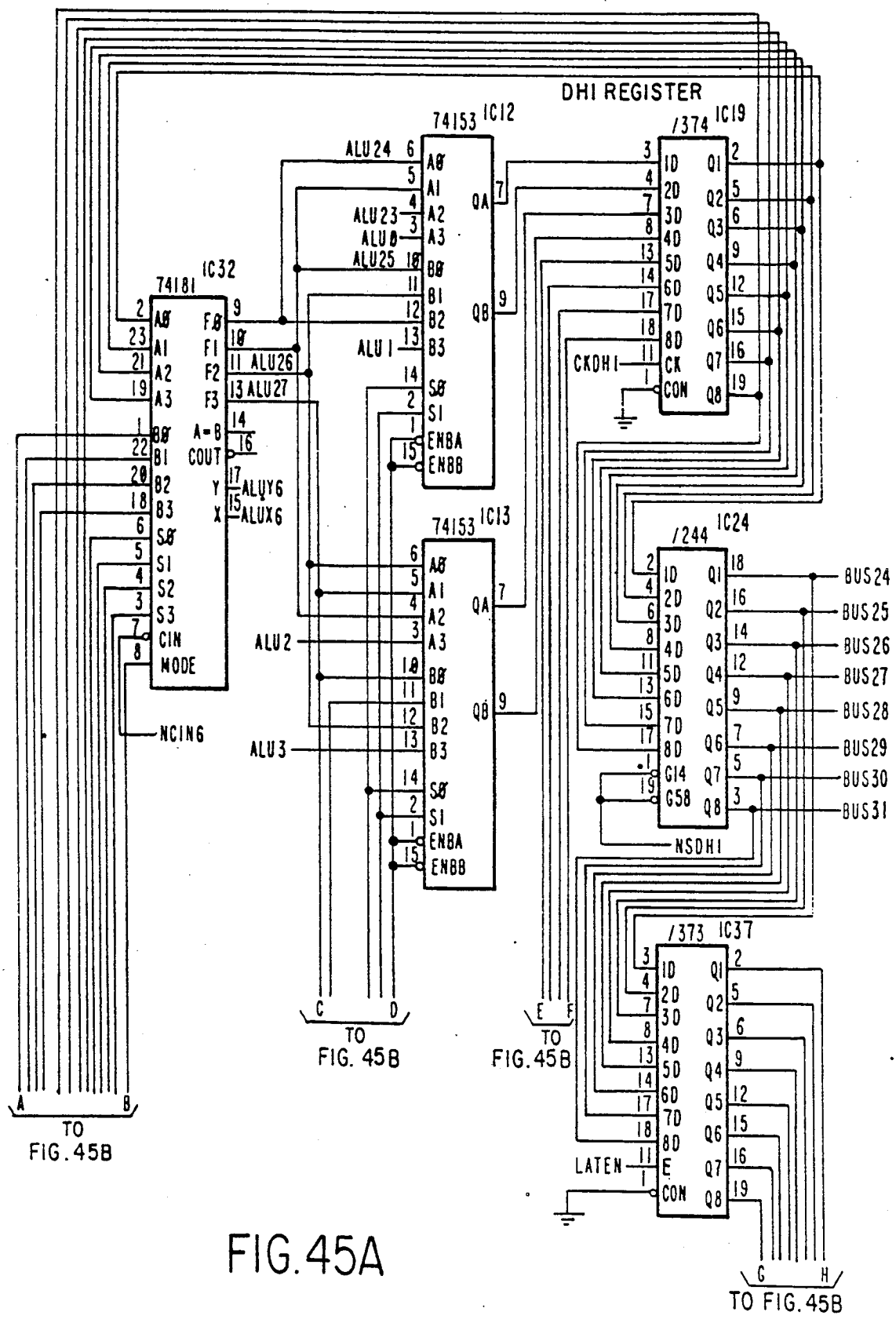

DHI REGISTER
74153 IC12
/374 IC19
74181 IC32
74153 IC13
/244 IC24
/373 IC37
ALU24
ALU23
ALU0
ALU25
ALU1
ALU2
ALU3
ALU26
ALU27
ALUY6
ALUX6
NCIN6
CKDHI
NSDHI
LATEN
BUS24
BUS25
BUS26
BUS27
BUS28
BUS29
BUS30
BUS31
TO FIG. 45B
TO FIG. 45B
TO FIG. 45B
TO FIG. 45B

FIG. 45A

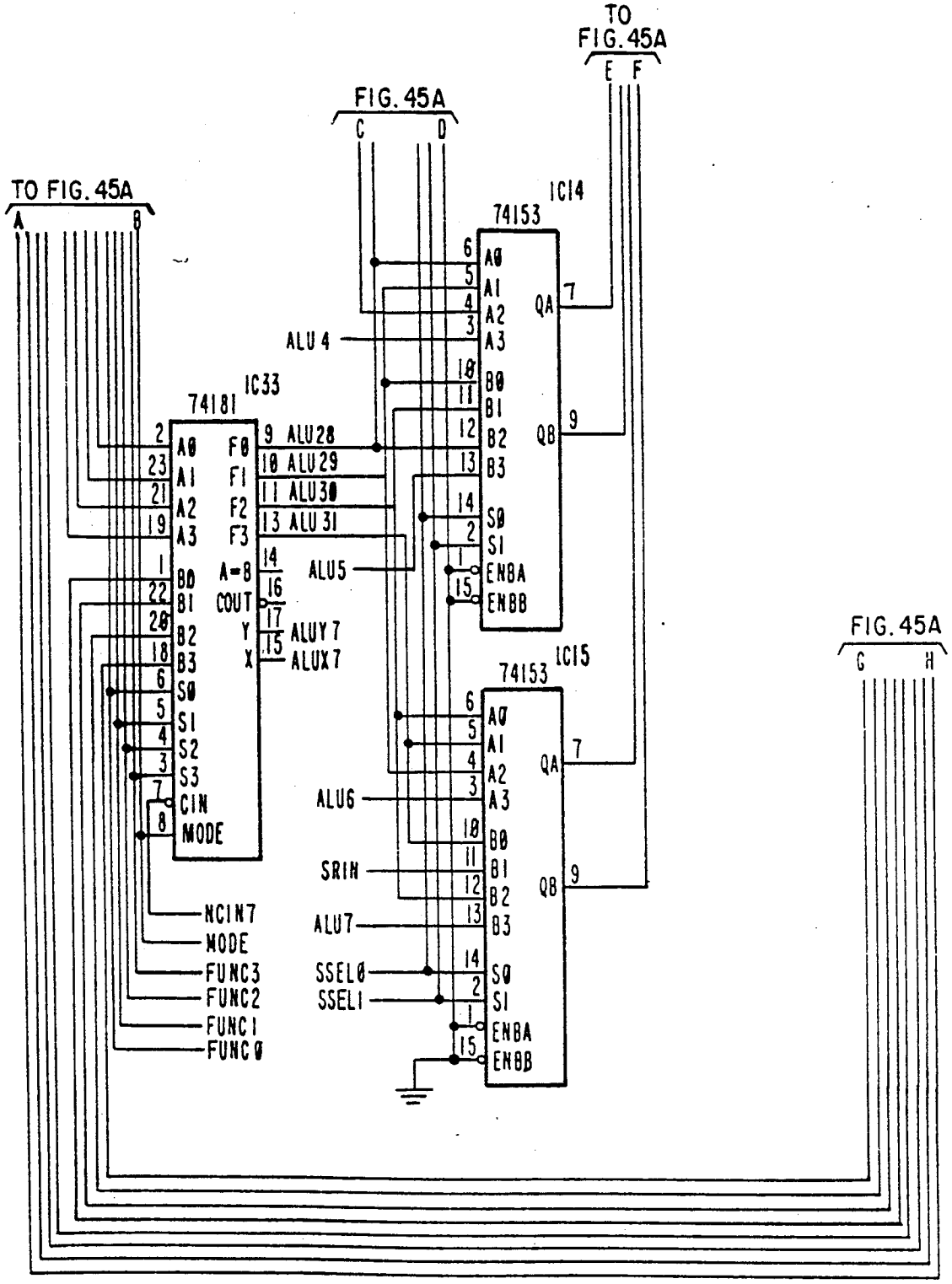
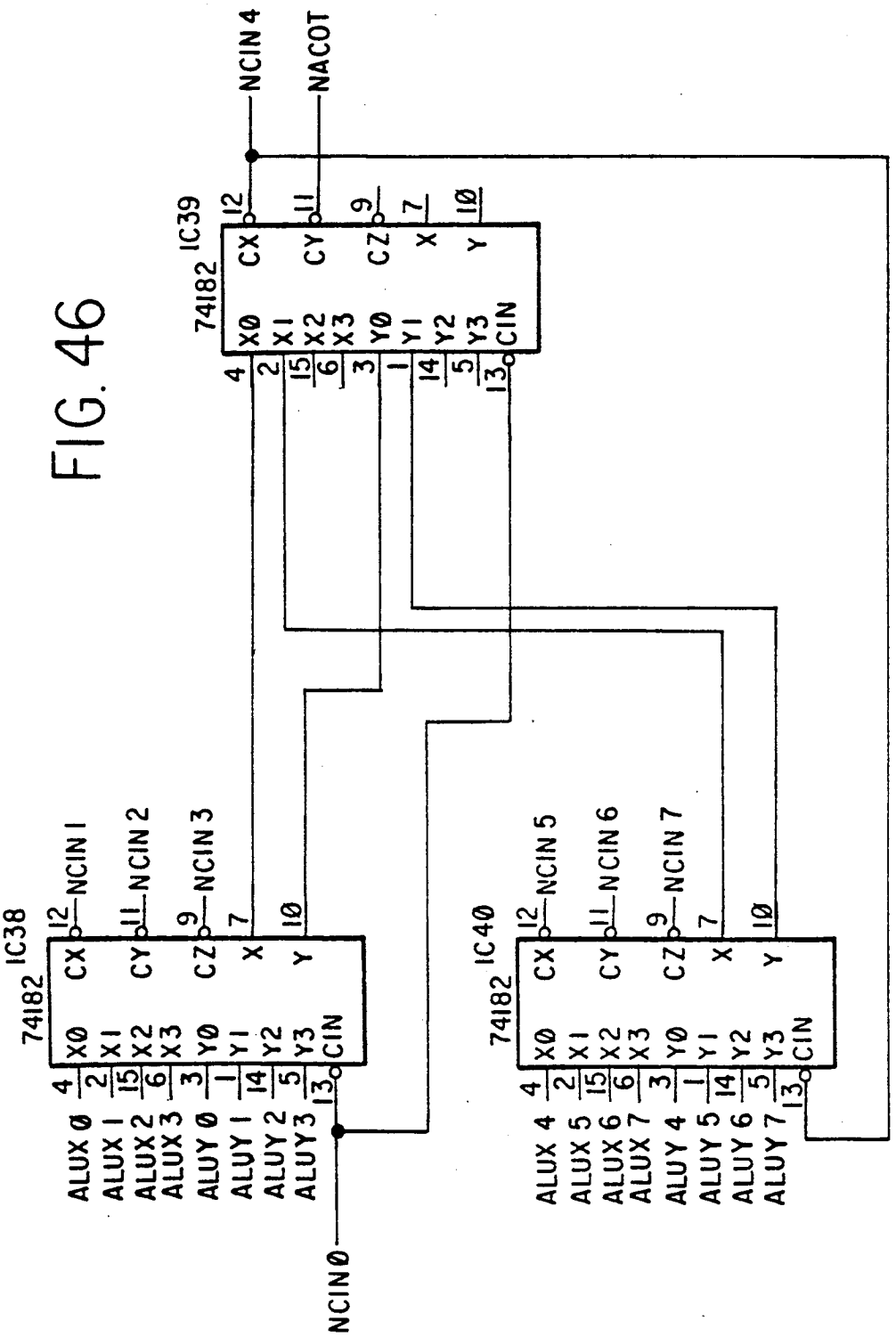

TO FIG. 45A
E F
FIG. 45A
C D
TO FIG. 45A
A B
IC14
74153
IC33
74181
ALU 4
ALU28
ALU29
ALU30
ALU31
ALU5
ALUY7
ALUX7
IC15
74153
FIG. 45A
G H
ALU6
SRIN
ALU7
SSEL0
SSEL1
NCIN7
MODE
FUNC3
FUNC2
FUNC1
FUNC0

FIG. 45B

FIG. 46


IC39
74182
NCIN 4
NACOT
IC38
NCIN 1
NCIN 2
NCIN 3
IC40
NCIN 5
NCIN 6
NCIN 7
NCIN 0
ALUX 0
ALUX 1
ALUX 2
ALUX 3
ALUY 0
ALUY 1
ALUY 2
ALUY 3
ALUX 4
ALUX 5
ALUX 6
ALUX 7
ALUY 4
ALUY 5
ALUY 6
ALUY 7

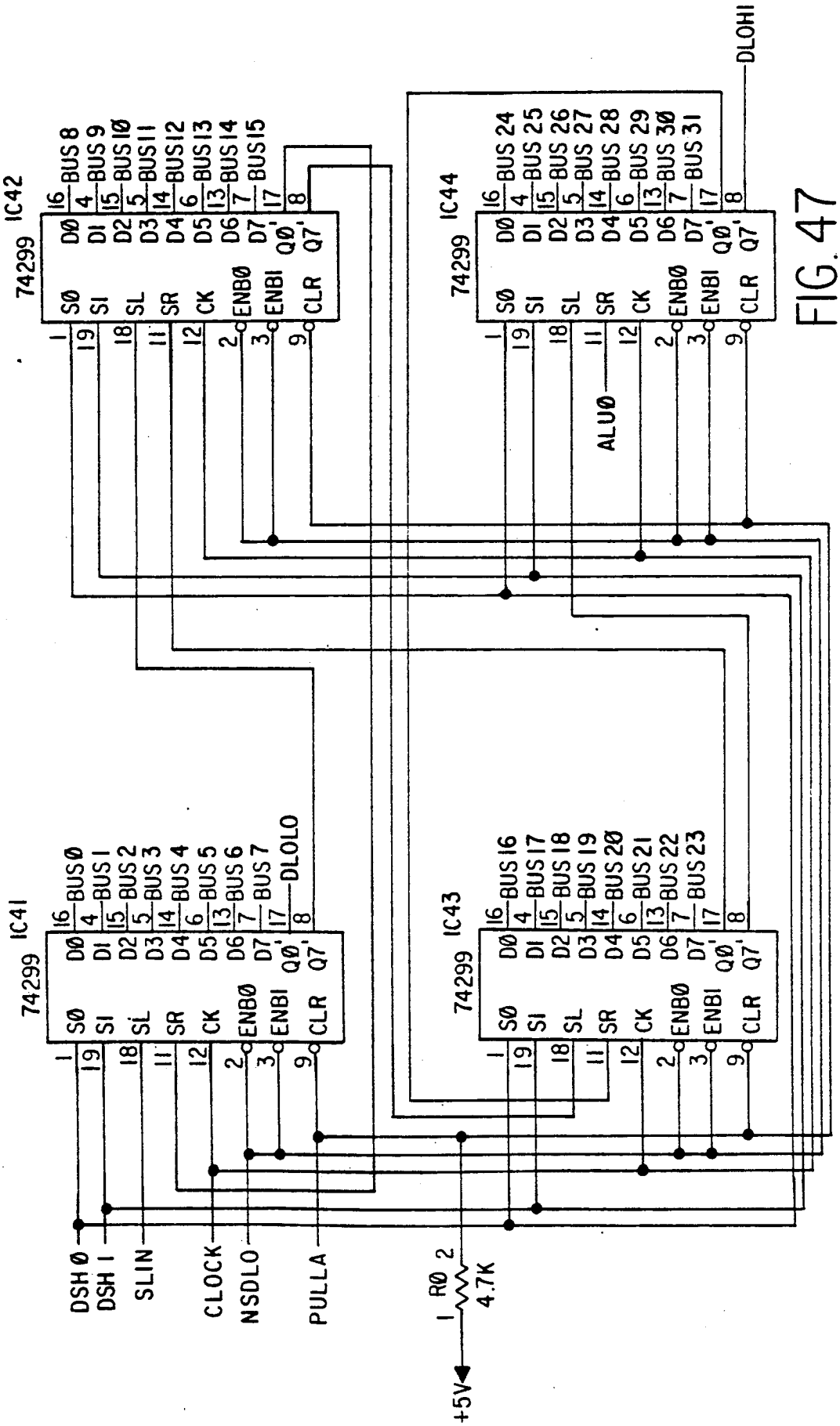

IC42
74299
S0
SI
SL
SR
CK
ENB0
ENBI
CLR
D0
D1
D2
D3
D4
D5
D6
D7
Q0'
Q7'
BUS 8
BUS 9
BUS 10
BUS 11
BUS 12
BUS 13
BUS 14
BUS 15
IC44
BUS 24
BUS 25
BUS 26
BUS 27
BUS 28
BUS 29
BUS 30
BUS 31
DLOHI
ALU0
IC41
BUS 0
BUS 1
BUS 2
BUS 3
BUS 4
BUS 5
BUS 6
BUS 7
DLOLO
IC43
BUS 16
BUS 17
BUS 18
BUS 19
BUS 20
BUS 21
BUS 22
BUS 23
DSH 0
DSH 1
SLIN
CLOCK
NSDLO
PULLA
R0
4.7K
+5V

FIG. 47

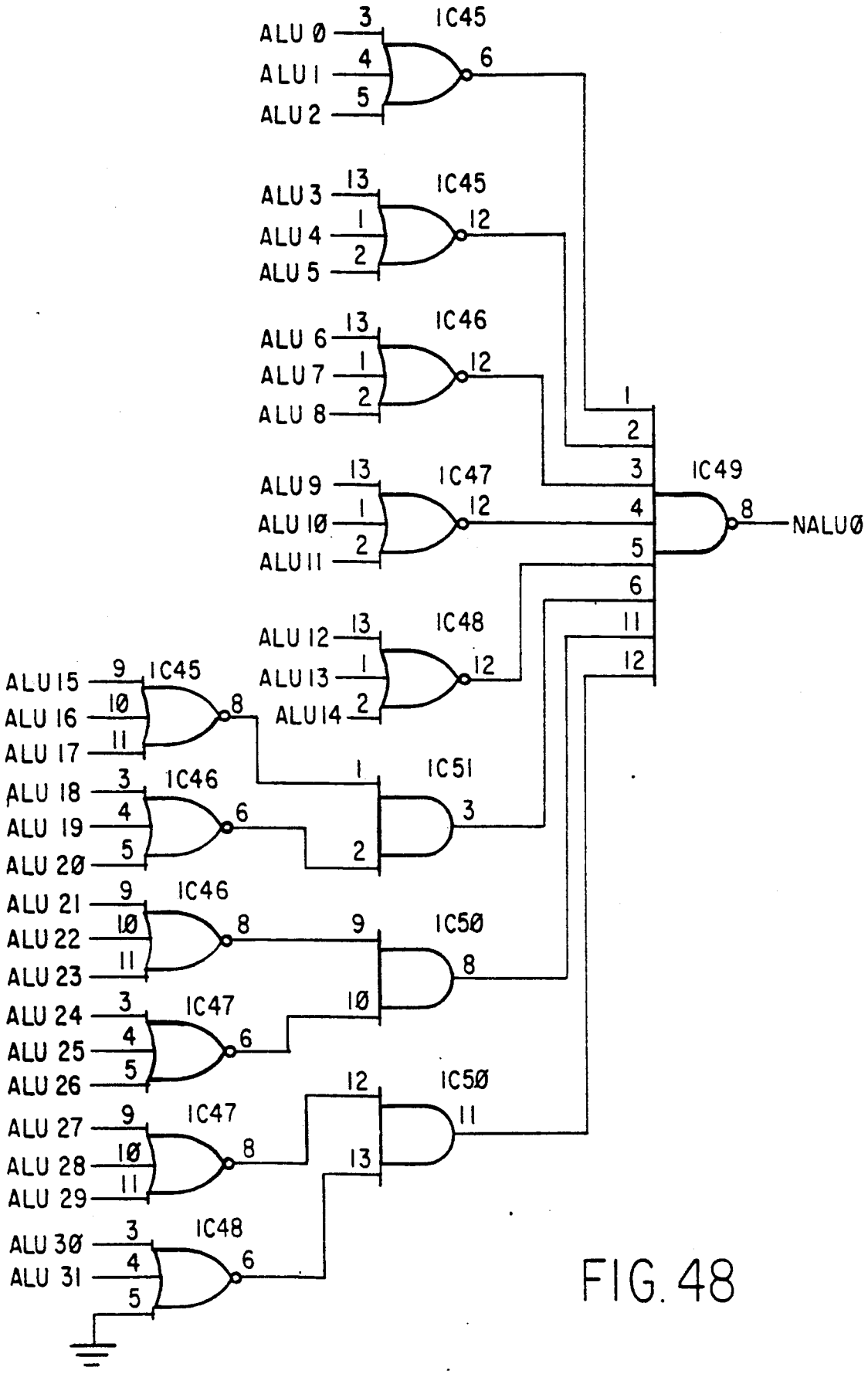

ALU 0
ALU 1
ALU 2
IC45
ALU 3
ALU 4
ALU 5
IC45
ALU 6
ALU 7
ALU 8
IC46
ALU 9
ALU 10
ALU 11
IC47
ALU 12
ALU 13
ALU 14
IC48
ALU 15
ALU 16
ALU 17
IC45
ALU 18
ALU 19
ALU 20
IC46
ALU 21
ALU 22
ALU 23
IC46
ALU 24
ALU 25
ALU 26
IC47
ALU 27
ALU 28
ALU 29
IC47
ALU 30
ALU 31
IC48
IC49
IC50
IC51
IC50
NALU0

FIG. 48

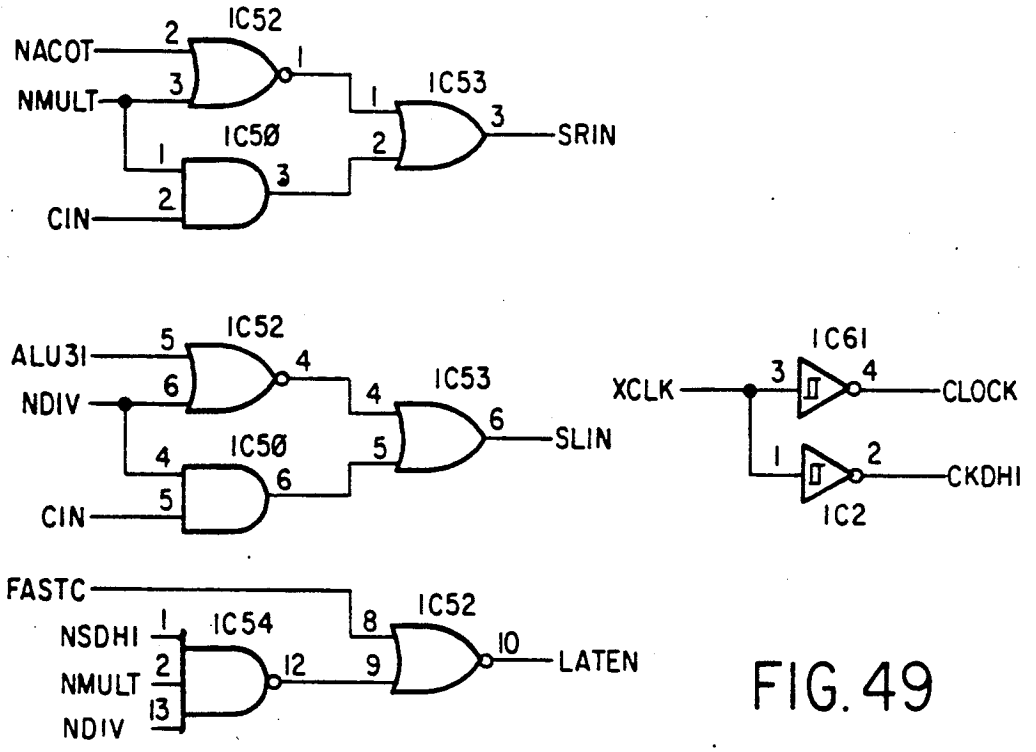

IC52
NACOT
NMULT
IC50
CIN
IC53
SRIN
ALU31
NDIV
SLIN
XCLK
IC61
CLOCK
CKDHI
IC2
FASTC
NSDHI
IC54
NMULT
NDIV
LATEN

FIG. 49

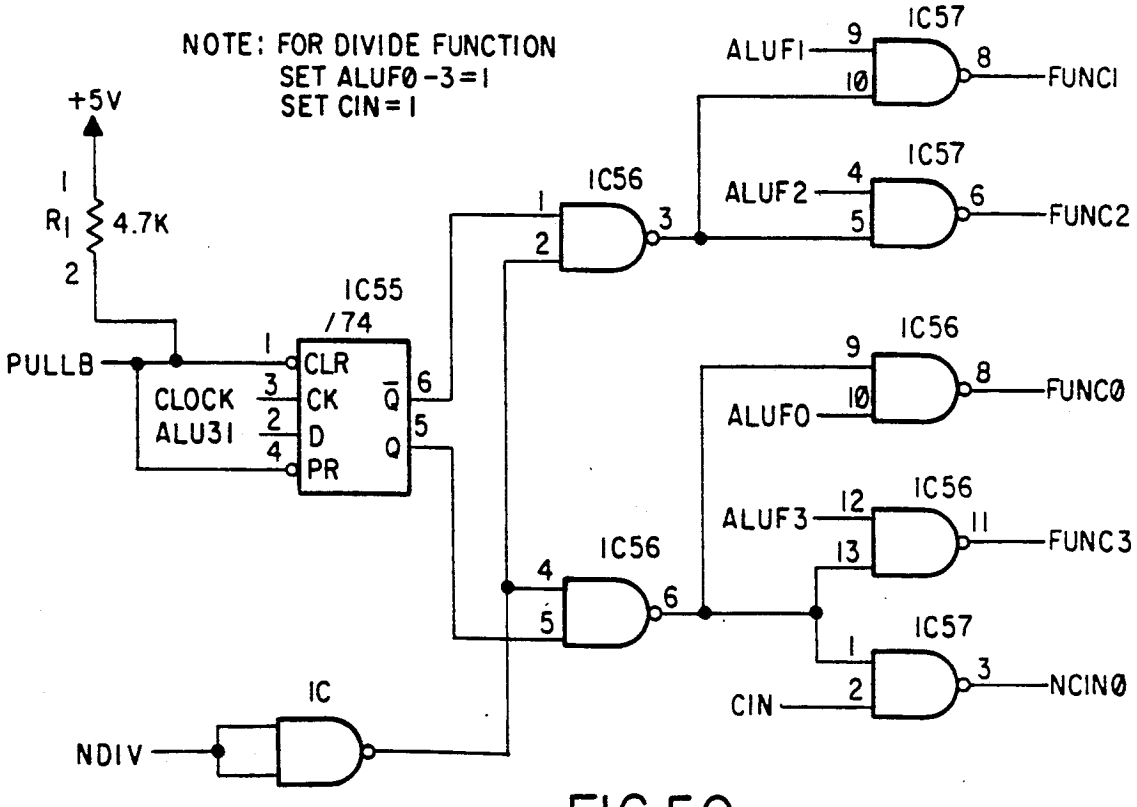

NOTE: FOR DIVIDE FUNCTION
SET ALUF0-3=1
SET CIN=1
+5V
R1
4.7K
PULLB
IC55
/74
CLR
CK
D
PR
Q
CLOCK
ALU31
IC56
IC
NDIV
ALUF1
IC57
FUNC1
ALUF2
FUNC2
ALUF0
FUNC0
ALUF3
FUNC3
CIN
NCIN0

FIG. 50

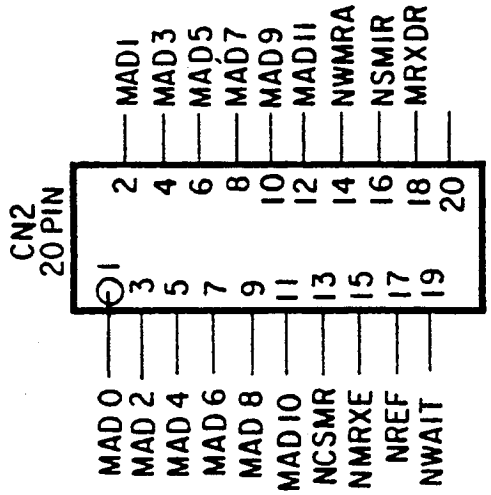
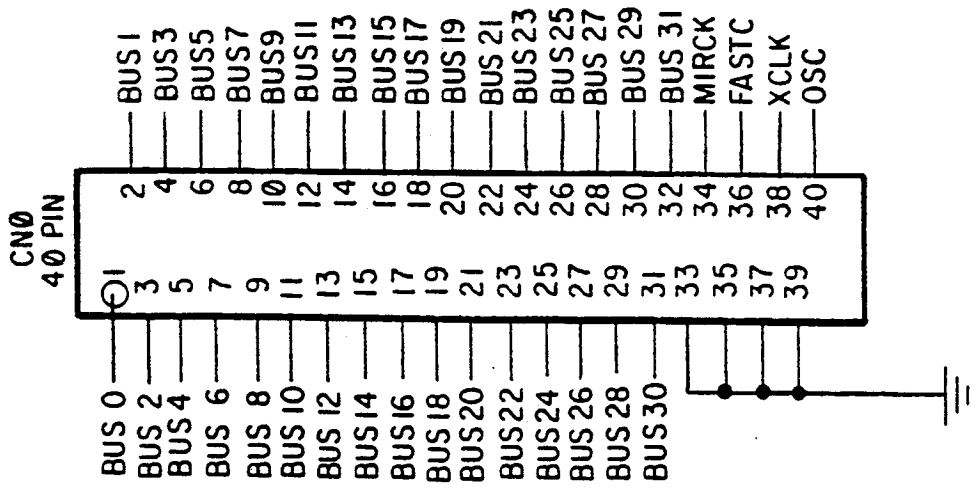

CN2
20 PIN
MAD 0
MAD 2
MAD 4
MAD 6
MAD 8
MAD 10
NCSMR
NMRXE
NREF
NWAIT
MAD1
MAD3
MAD5
MAD7
MAD9
MAD11
NWMRA
NSMIR
MRXDR

FIG. 51


CN0
40 PIN
BUS 0
BUS 2
BUS 4
BUS 6
BUS 8
BUS 10
BUS 12
BUS 14
BUS 16
BUS 18
BUS 20
BUS 22
BUS 24
BUS 26
BUS 28
BUS 30
BUS 1
BUS 3
BUS 5
BUS 7
BUS 9
BUS 11
BUS 13
BUS 15
BUS 17
BUS 19
BUS 21
BUS 23
BUS 25
BUS 27
BUS 29
BUS 31
MIRCK
FASTC
XCLK
OSC

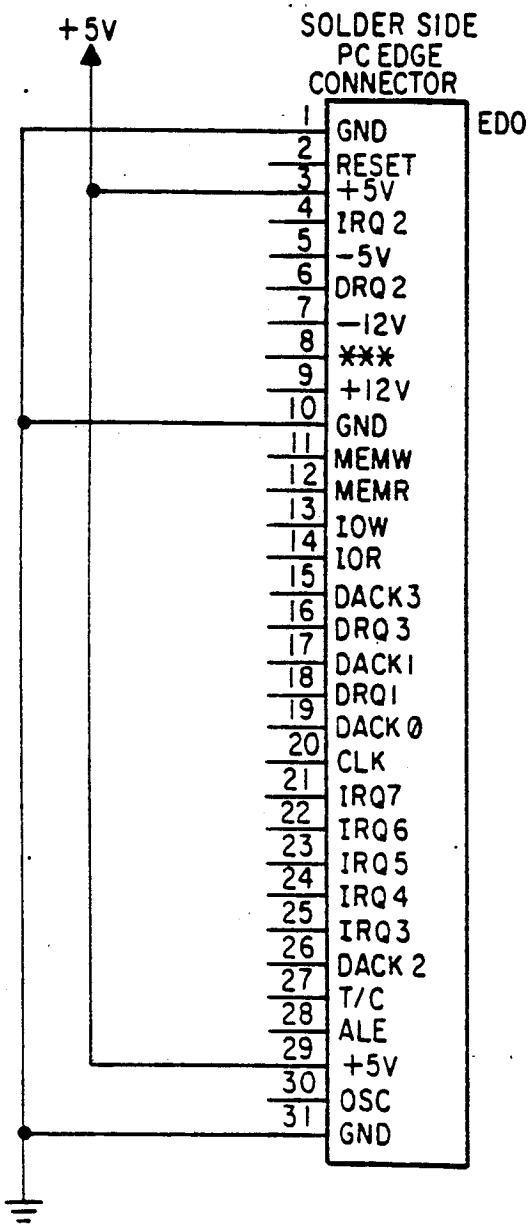

+5V
SOLDER SIDE
PC EDGE
CONNECTOR
EDO
1 GND
2 RESET
3 +5V
4 IRQ 2
5 -5V
6 DRQ 2
7 -12V
8 ***
9 +12V
10 GND
11 MEMW
12 MEMR
13 IOW
14 IOR
15 DACK3
16 DRQ 3
17 DACK1
18 DRQ1
19 DACK 0
20 CLK
21 IRQ7
22 IRQ6
23 IRQ5
24 IRQ4
25 IRQ3
26 DACK 2
27 T/C
28 ALE
29 +5V
30 OSC
31 GND

FIG. 53

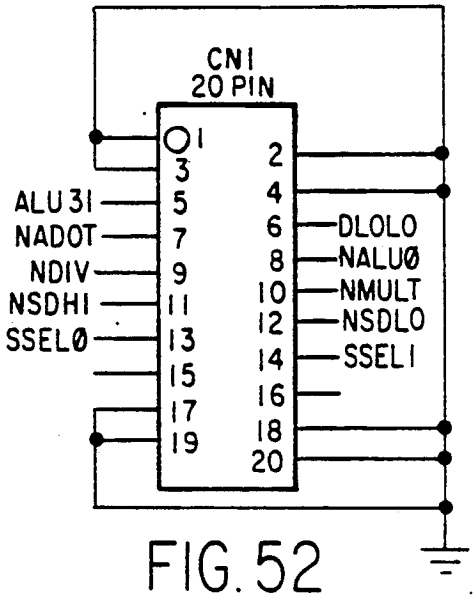

CN1
20 PIN
ALU 31
NADOT
NDIV
NSDHI
SSEL0
DLOLO
NALU0
NMULT
NSDLO
SSEL1

FIG. 52

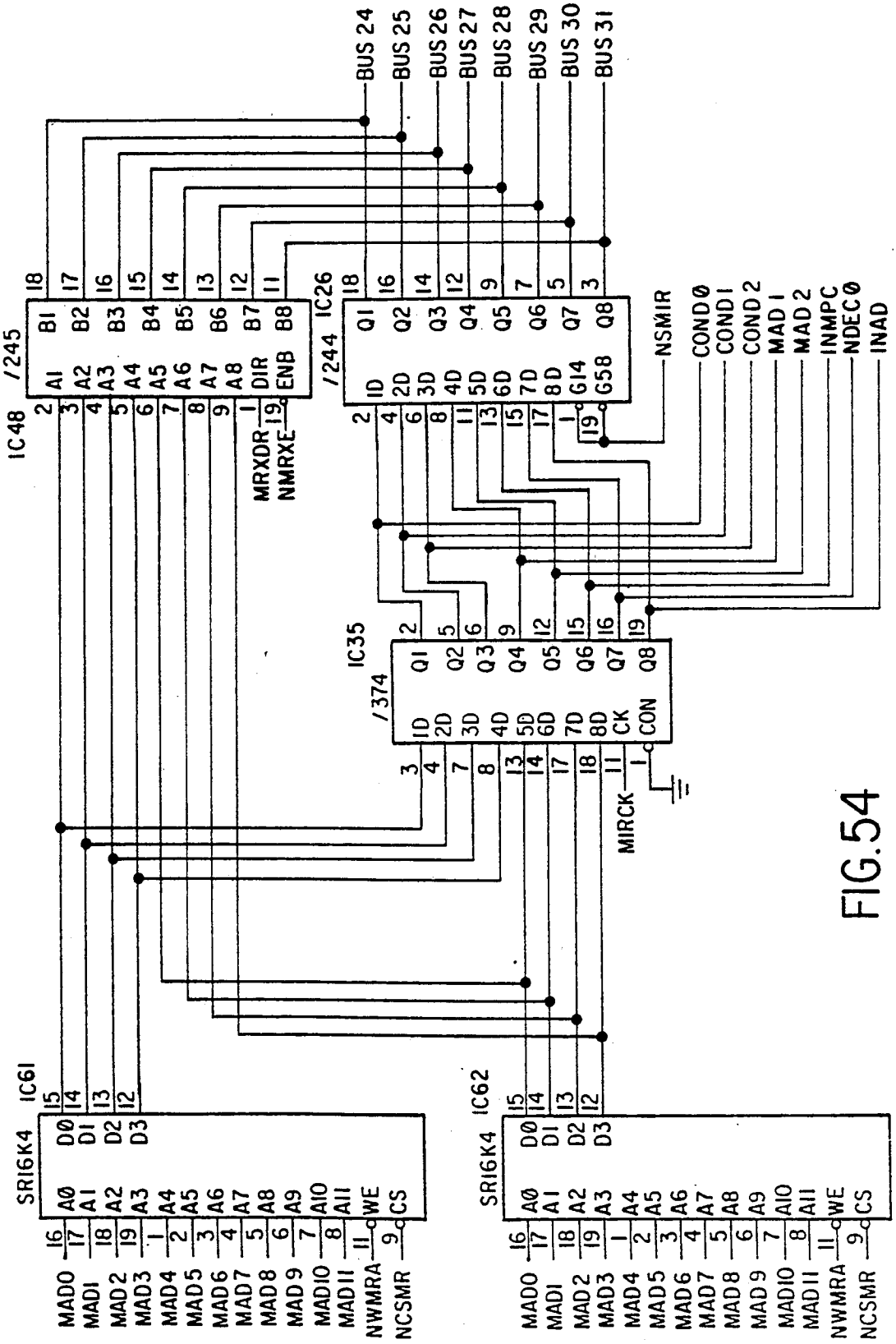

BUS 24
BUS 25
BUS 26
BUS 27
BUS 28
BUS 29
BUS 30
BUS 31
IC48
/245
IC26
/244
MRXDR
NMRXE
NSMIR
COND0
COND1
COND2
MAD1
MAD2
INMPC
NDEC0
INAD
IC35
/374
MIRCK
IC61
IC62
SR16K4
MAD0
MAD1
MAD2
MAD3
MAD4
MAD5
MAD6
MAD7
MAD8
MAD9
MAD10
MAD11
NWMRA
NCSMR

FIG.54

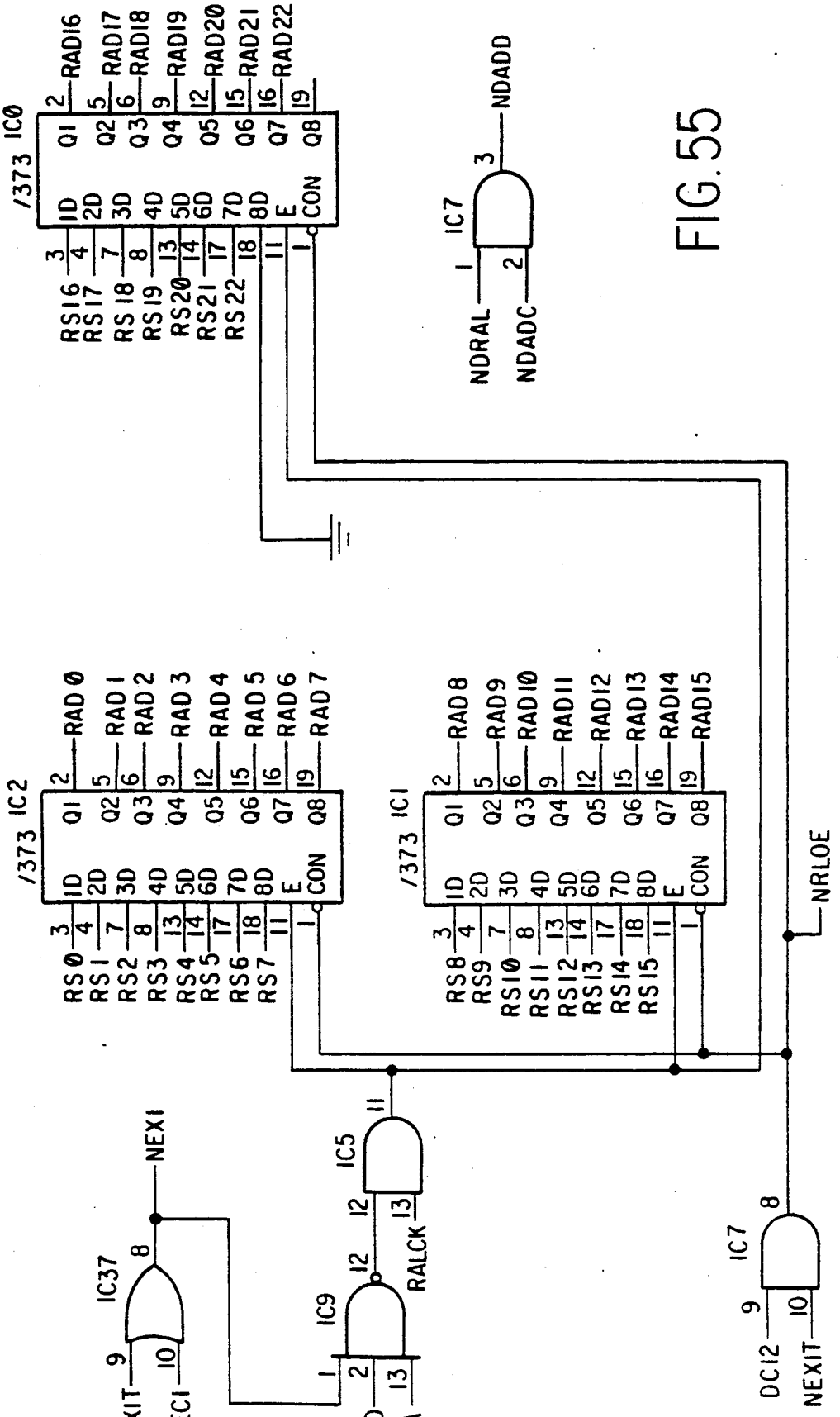
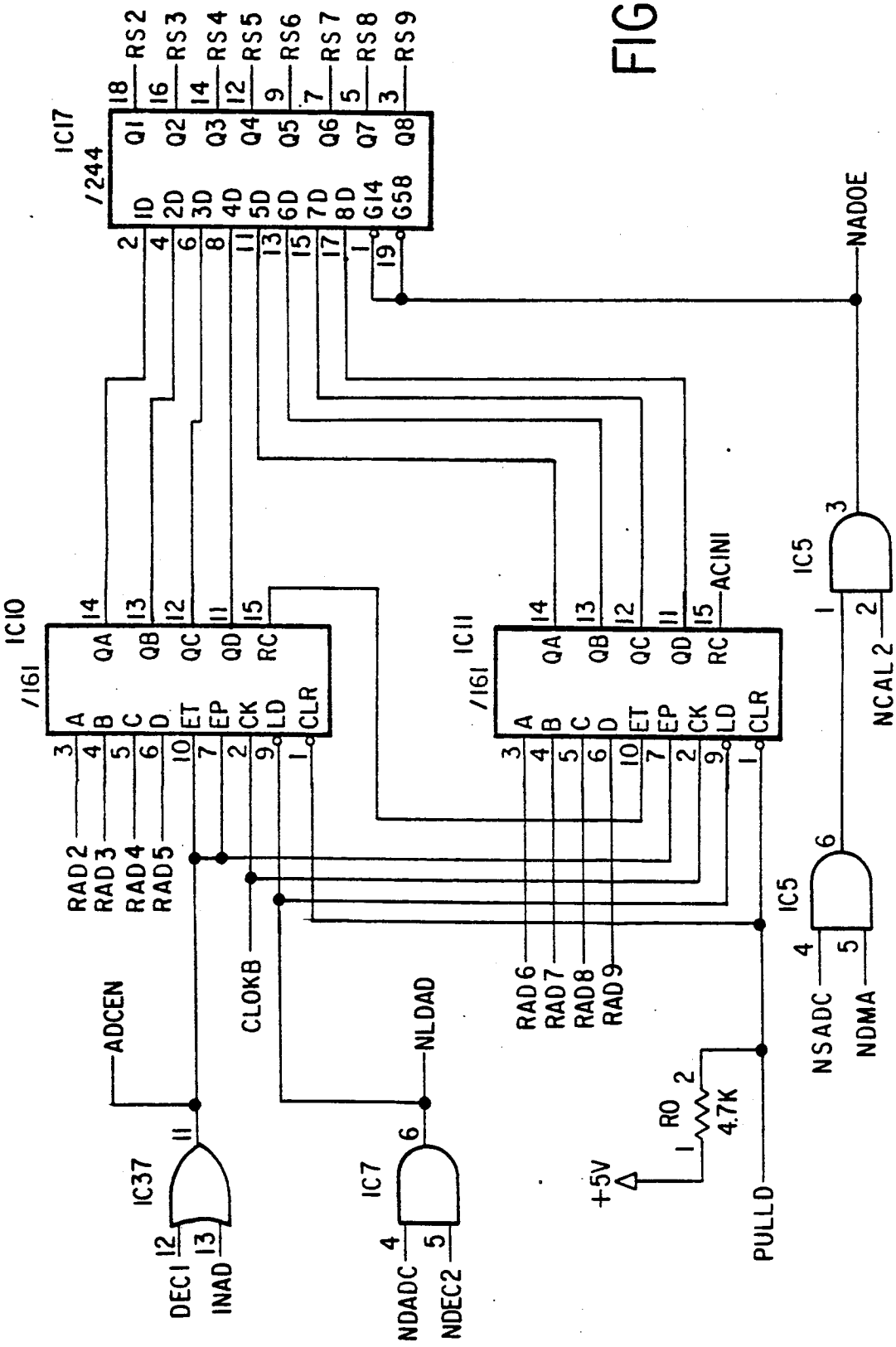

IC0
/373
RAD16
RAD17
RAD18
RAD19
RAD20
RAD21
RAD22
RS16
RS17
RS 18
RS 19
RS20
RS21
RS 22
IC7
NDADD
NDRAL
NDADC
IC2
RAD 0
RAD 1
RAD 2
RAD 3
RAD 4
RAD 5
RAD 6
RAD 7
RS 0
RS 1
RS2
RS3
RS4
RS5
RS6
RS7
IC1
RAD 8
RAD 9
RAD 10
RAD 11
RAD 12
RAD 13
RAD 14
RAD 15
RS8
RS9
RS10
RS11
RS12
RS13
RS14
RS15
NRLOE
IC5
RALCK
IC9
NDADD
NDMA
IC37
NEXI
NEXIT
NDECI
IC7
DC12
NEXIT

FIG. 55

FIG. 56


IC17
/244
RS2
RS3
RS4
RS5
RS6
RS7
RS8
RS9
NADOE
IC10
/161
IC11
/161
ACINI
IC5
NCAL2
RAD2
RAD3
RAD4
RAD5
RAD6
RAD7
RAD8
RAD9
CLOKB
ADCEN
NLDAD
IC37
IC7
DEC1
INAD
NDADC
NDEC2
+5V
RO
4.7K
PULLD
NSADC
NDMA

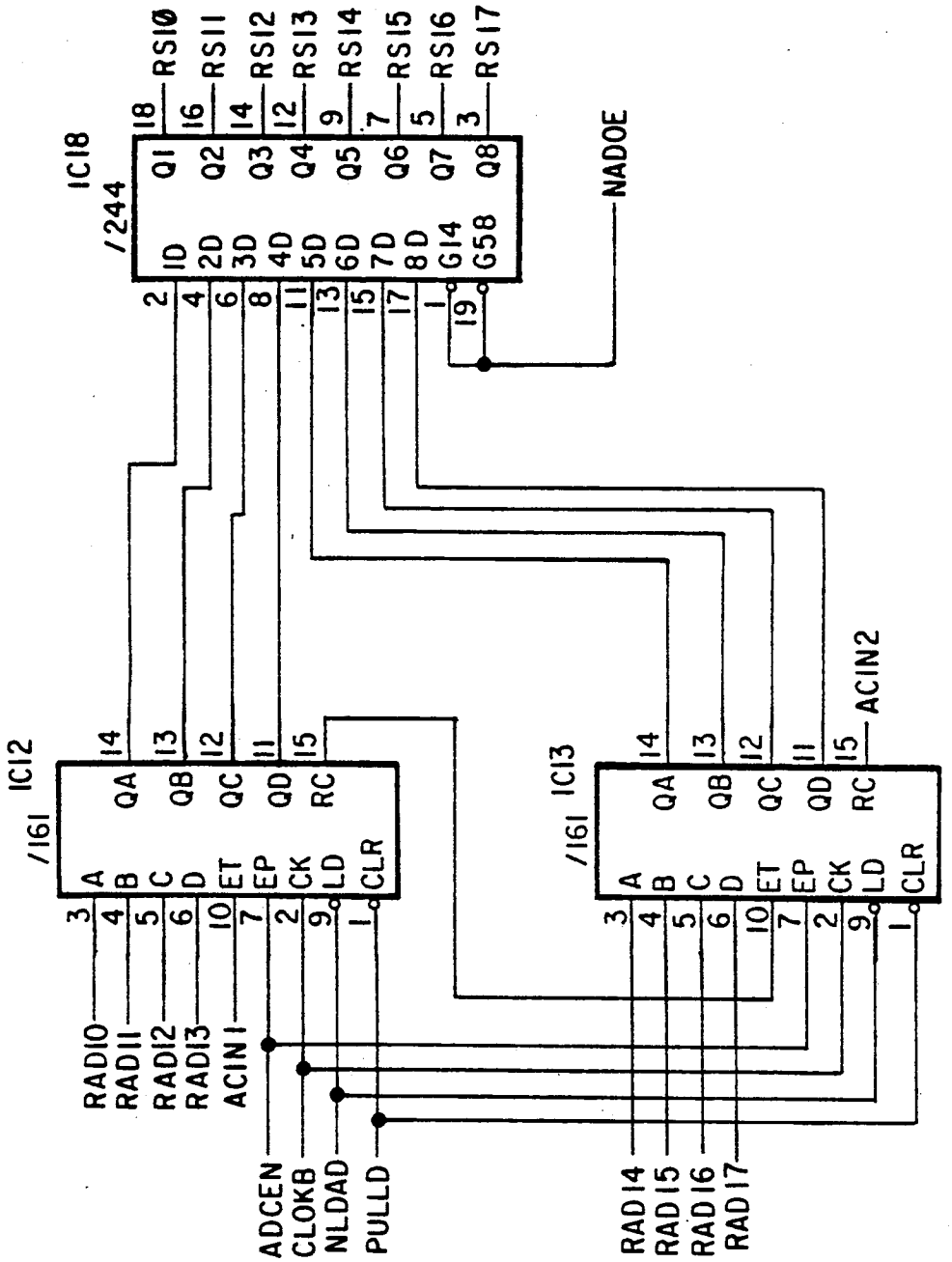

IC18
/244
RS10
RS11
RS12
RS13
RS14
RS15
RS16
RS17
NADOE
IC12
/161
IC13
/161
ACIN2
RAD10
RAD11
RAD12
RAD13
ACIN1
ADCEN
CLOKB
NLDAD
PULLD
RAD14
RAD15
RAD16
RAD17

FIG. 57

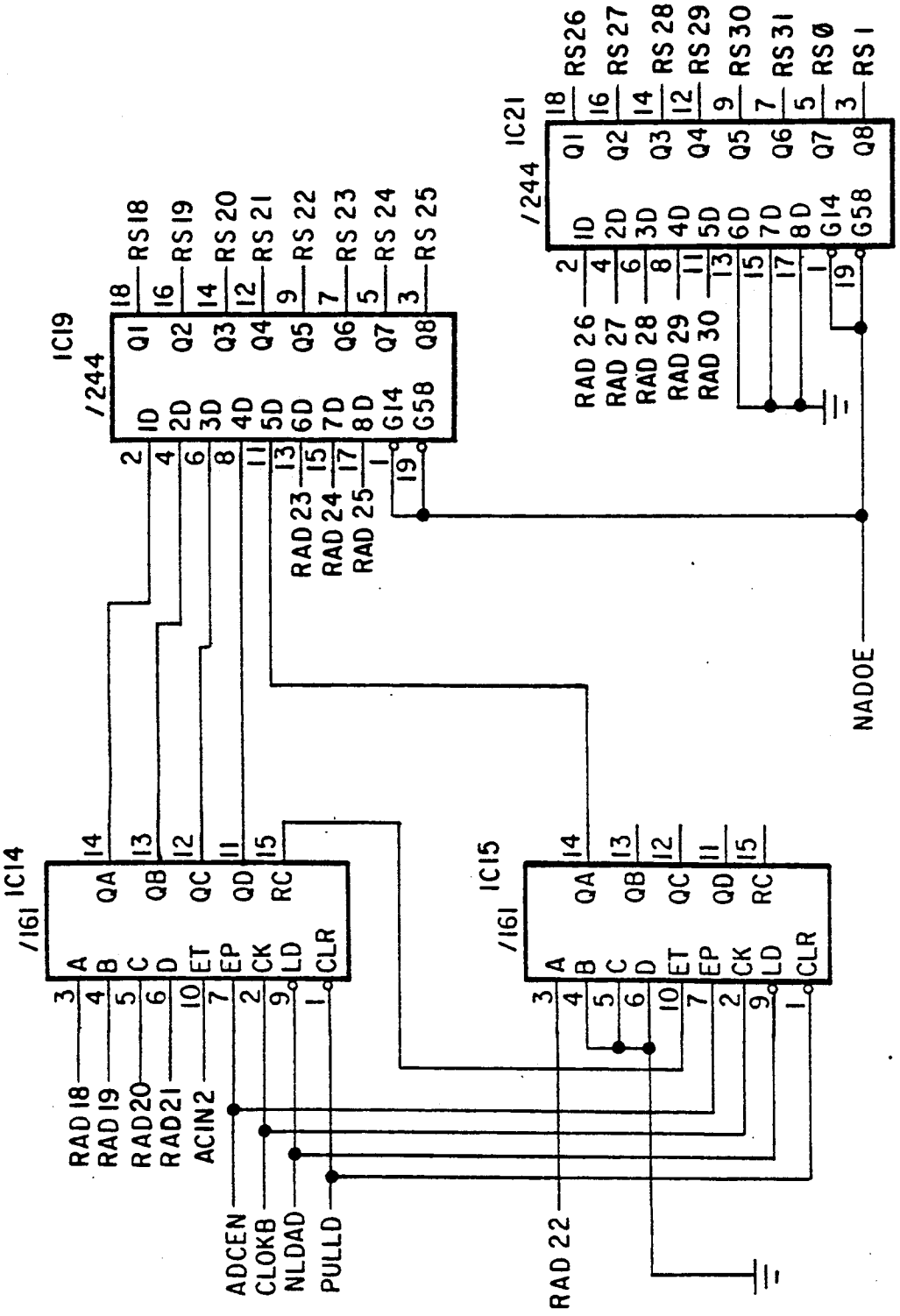

IC19
/244
IC21
/244
IC14
/161
IC15
/161
RS18
RS19
RS20
RS21
RS22
RS23
RS24
RS25
RS26
RS27
RS28
RS29
RS30
RS31
RS0
RS1
RAD 23
RAD 24
RAD 25
RAD 26
RAD 27
RAD 28
RAD 29
RAD 30
NADOE
RAD 18
RAD 19
RAD20
RAD21
ACIN2
ADCEN
CLOKB
NLDAD
PULLD
RAD 22

FIG. 58

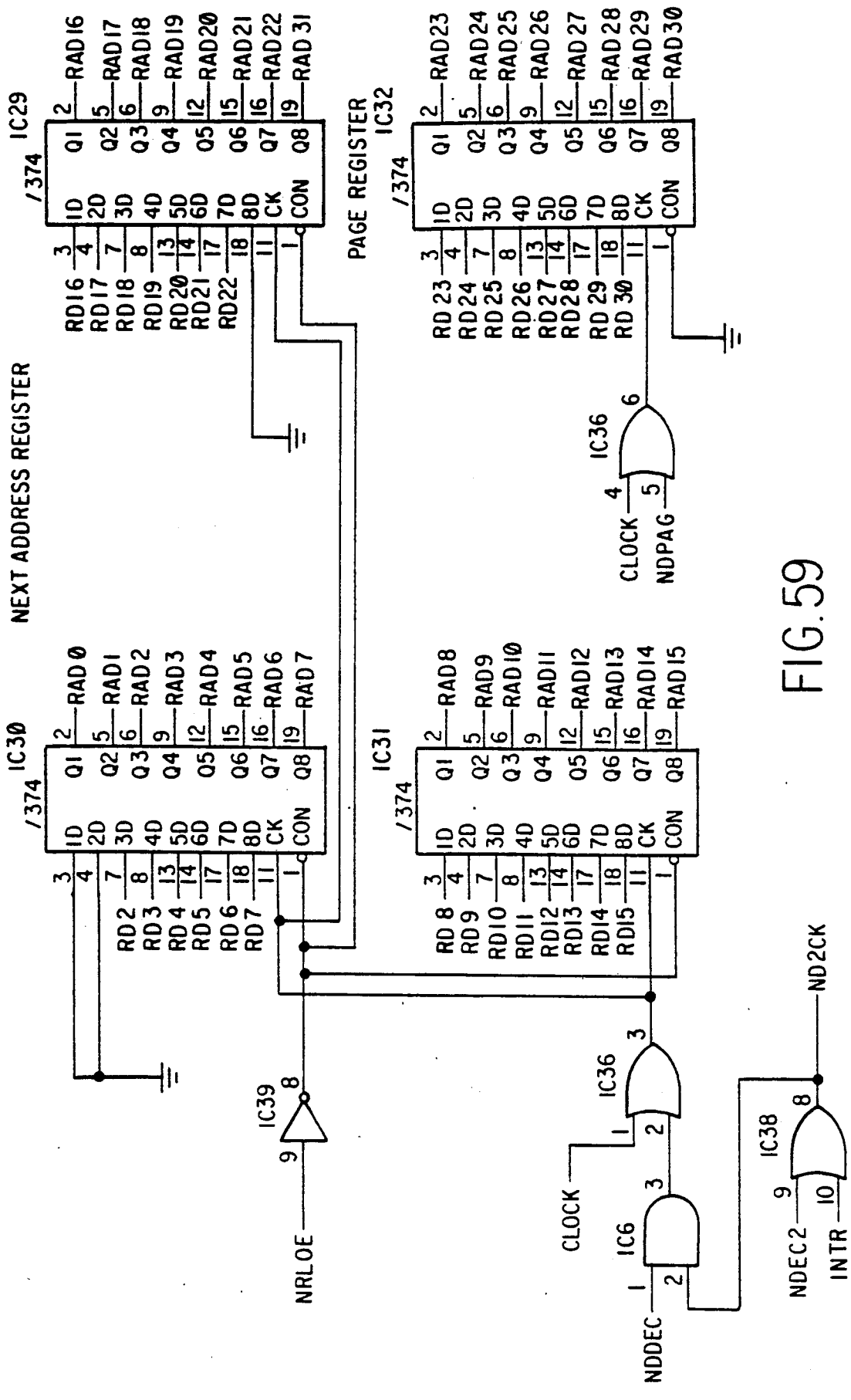

NEXT ADDRESS REGISTER
PAGE REGISTER
IC29
IC30
IC31
IC32
/374
RAD16
RAD17
RAD18
RAD19
RAD20
RAD21
RAD22
RAD31
RAD0
RAD1
RAD2
RAD3
RAD4
RAD5
RAD6
RAD7
RAD8
RAD9
RAD10
RAD11
RAD12
RAD13
RAD14
RAD15
RAD23
RAD24
RAD25
RAD26
RAD27
RAD28
RAD29
RAD30
IC36
IC39
IC6
IC38
CLOCK
NDPAG
NRLOE
NDDEC
NDEC2
INTR
ND2CK

FIG. 59

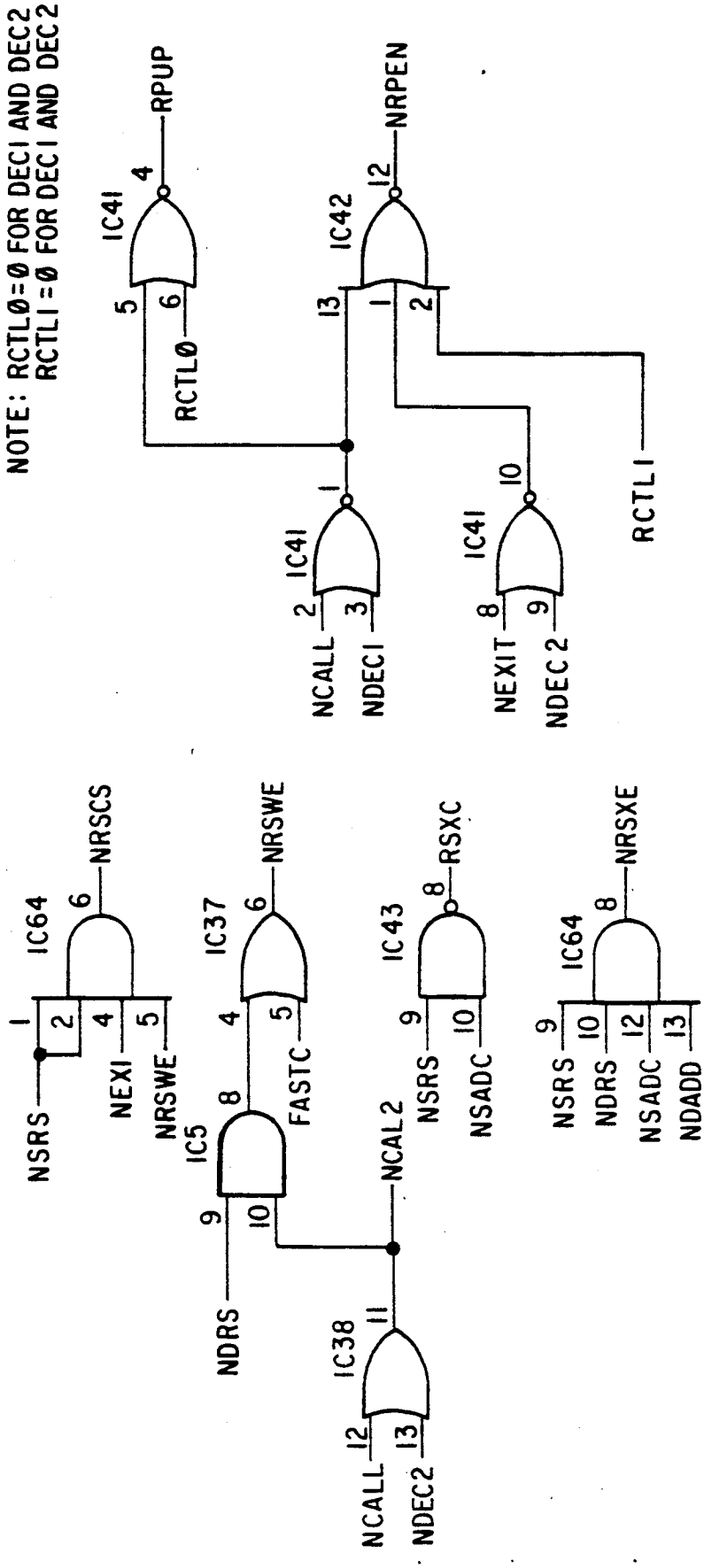

NOTE: RCTL0=0 FOR DEC1 AND DEC2
RCTL1=0 FOR DEC1 AND DEC2
IC41
RPUP
RCTL0
IC42
NRPEN
RCTL1
IC41
NCALL
NDEC1
IC41
NEXIT
NDEC2
IC64
NRSCS
NSRS
NEXI
NRSWE
IC37
NRSWE
IC5
FASTC
NDRS
NCAL2
IC43
RSXC
NSRS
NSADC
IC64
NRSXE
NSRS
NDRS
NSADC
NDADD
IC38
NCALL
NDEC2

FIG. 60

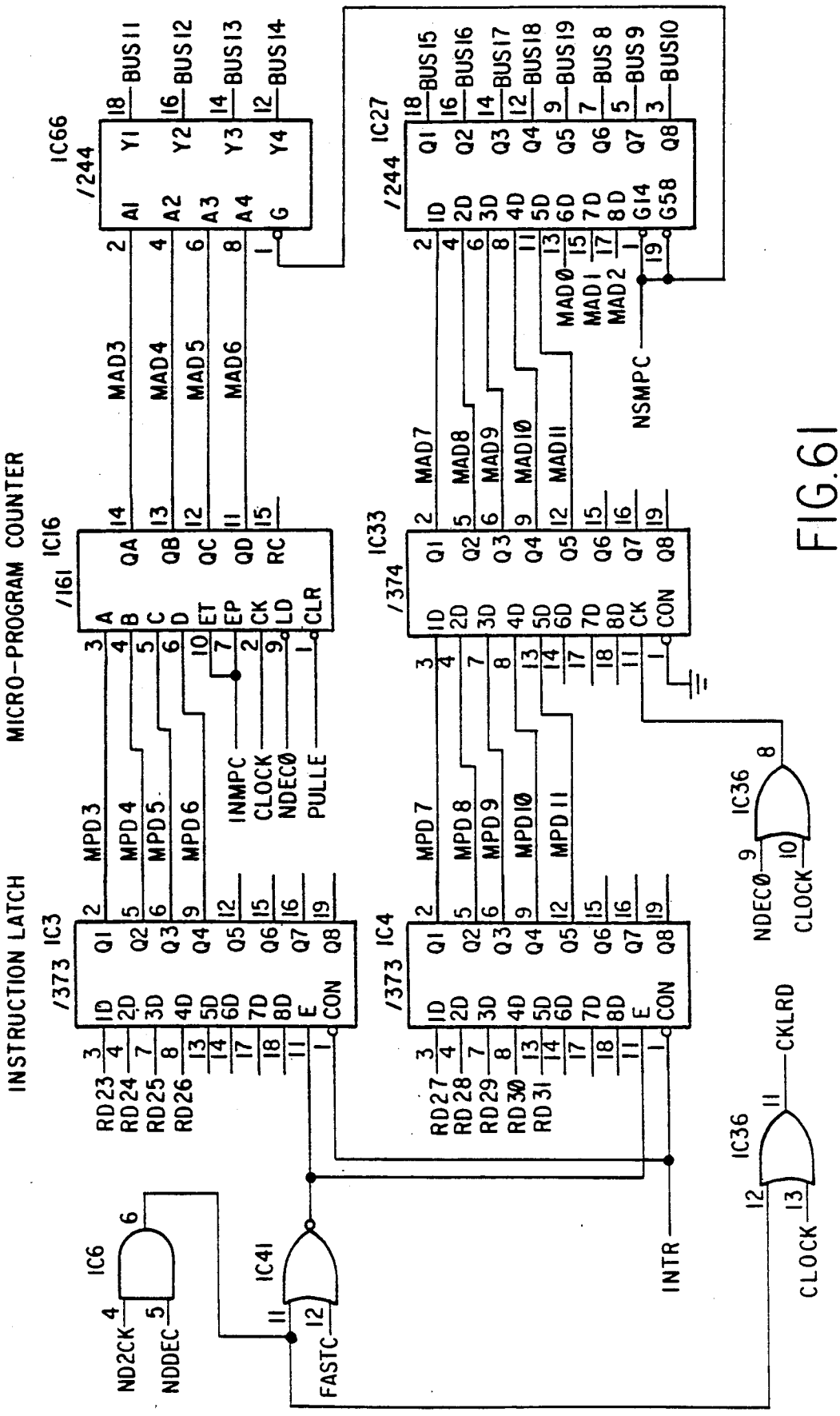

INSTRUCTION LATCH
MICRO-PROGRAM COUNTER
IC3 /373
IC4 /373
IC16 /161
IC33 /374
IC66 /244
IC27 /244
IC6
IC41
IC36
IC36
MPD3
MPD4
MPD5
MPD6
MPD7
MPD8
MPD9
MPD10
MPD11
RD23
RD24
RD25
RD26
RD27
RD28
RD29
RD30
RD31
INMPC
CLOCK
NDEC0
PULLE
MAD3
MAD4
MAD5
MAD6
MAD7
MAD8
MAD9
MAD10
MAD11
MAD0
MAD1
MAD2
NSMPC
BUS11
BUS12
BUS13
BUS14
BUS15
BUS16
BUS17
BUS18
BUS19
BUS 8
BUS 9
BUS10
ND2CK
NDDEC
FASTC
INTR
CKLRD
CLOCK

FIG.61

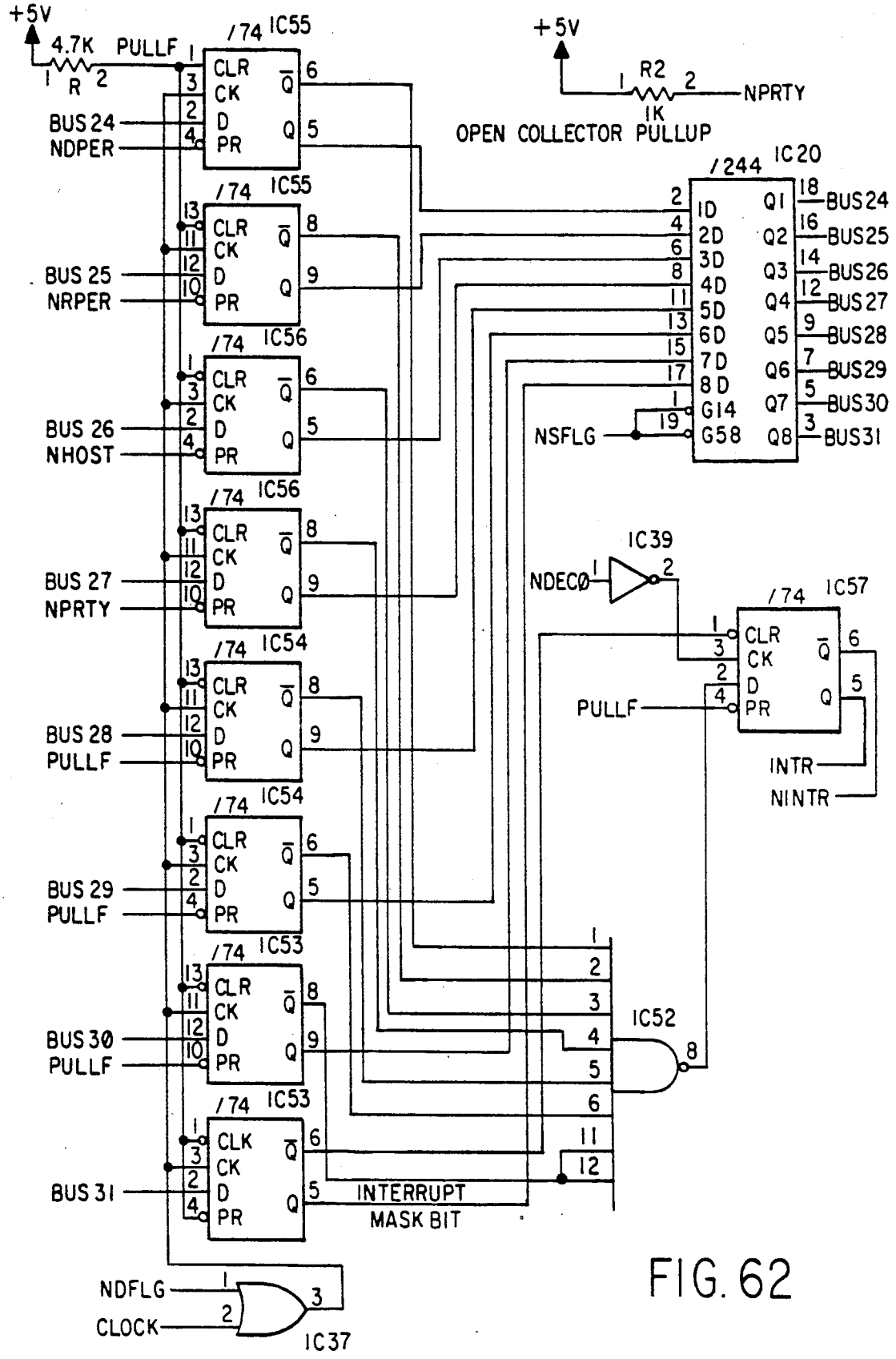

+5V
4.7K
PULLF
R
BUS 24
NDPER
BUS 25
NRPER
BUS 26
NHOST
BUS 27
NPRTY
BUS 28
PULLF
BUS 29
PULLF
BUS 30
PULLF
BUS 31
NDFLG
CLOCK
IC37
/74 IC55
/74 IC56
/74 IC54
/74 IC53
CLR
CK
D
PR
CLK
Q
+5V
R2
1K
NPRTY
OPEN COLLECTOR PULLUP
/244 IC20
ID
2D
3D
4D
5D
6D
7D
8D
G14
G58
NSFLG
BUS24
BUS25
BUS26
BUS27
BUS28
BUS29
BUS30
BUS31
IC39
NDEC0
/74 IC57
INTR
NINTR
IC52
INTERRUPT
MASK BIT

FIG. 62

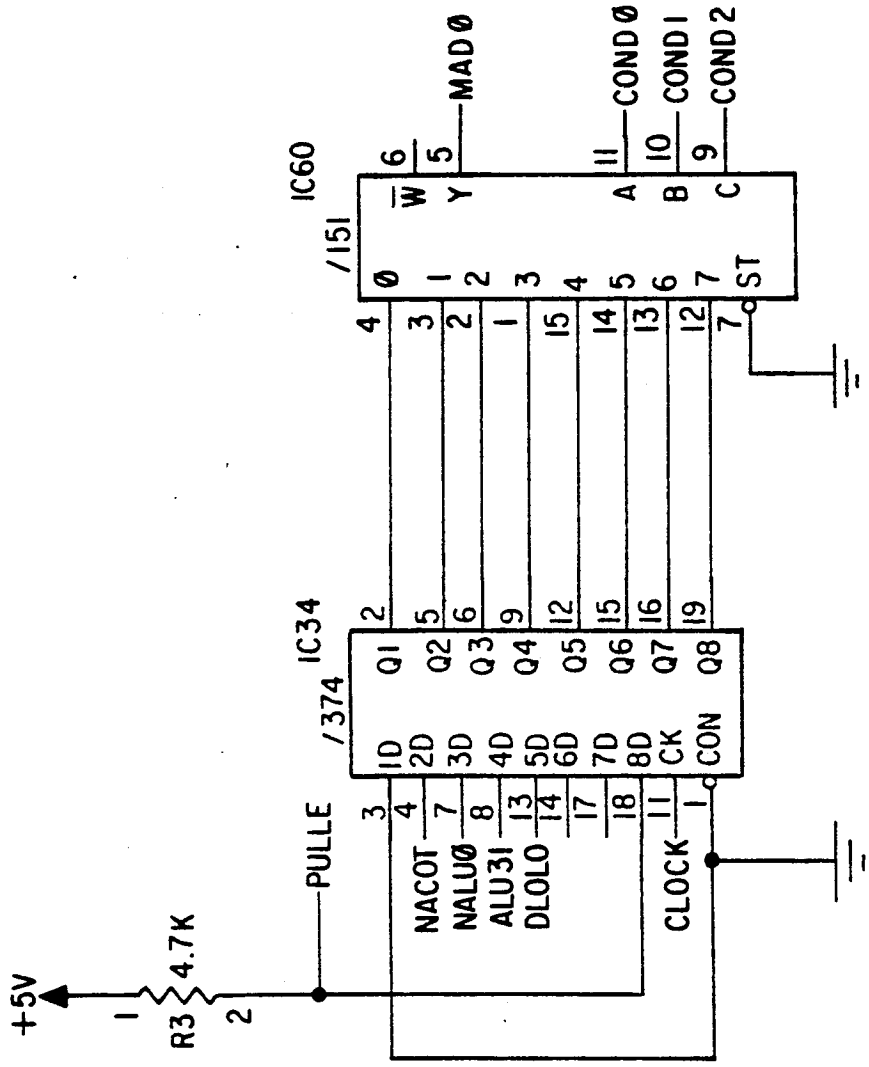

IC60
/151
W
Y
A
B
C
MAD0
COND0
COND1
COND2
0
1
2
3
4
5
6
7
ST
IC34
/374
1D
2D
3D
4D
5D
6D
7D
8D
CK
CON
Q1
Q2
Q3
Q4
Q5
Q6
Q7
Q8
PULLE
NACOT
NALU0
ALU31
DLOLO
CLOCK
+5V
R3
4.7K

FIG.63

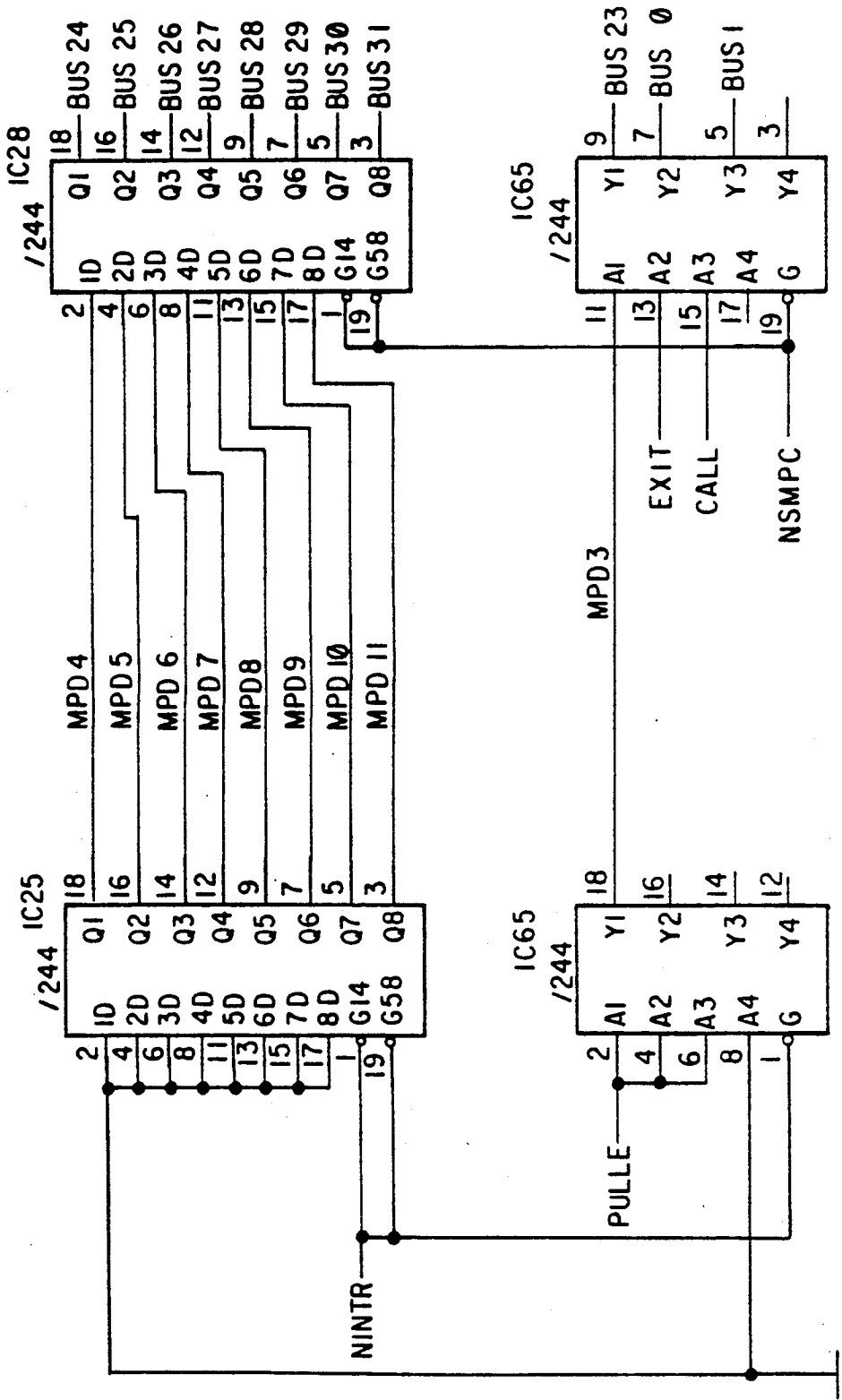

IC28
/244
BUS 24
BUS 25
BUS 26
BUS 27
BUS 28
BUS 29
BUS 30
BUS 31
IC65
/244
BUS 23
BUS 0
BUS 1
IC25
/244
MPD 4
MPD 5
MPD 6
MPD 7
MPD8
MPD9
MPD 10
MPD 11
IC65
/244
MPD3
EXIT
CALL
NSMPC
PULLE
NINTR

FIG.64

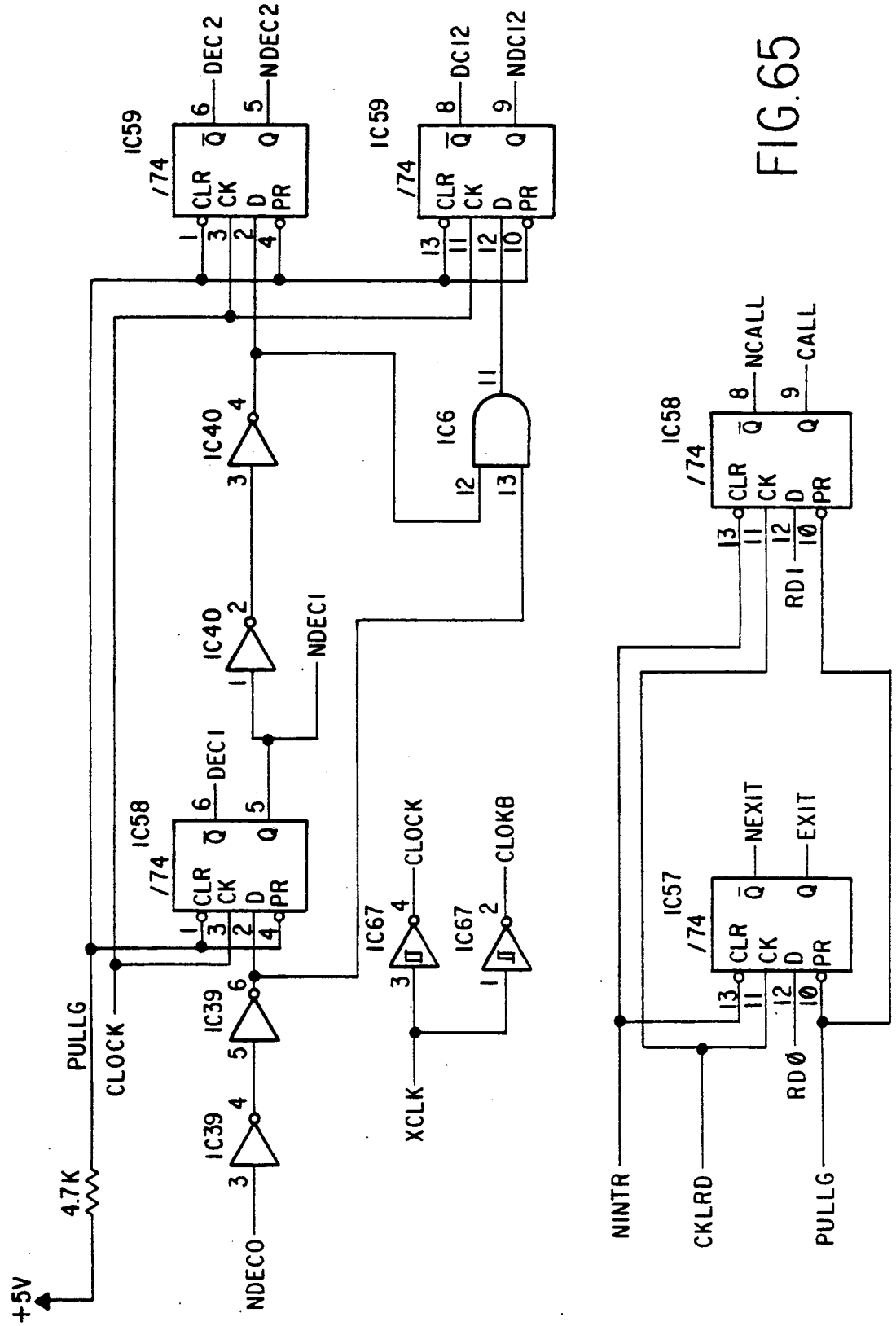

IC59
/74
CLR
CK
D
PR
Q
DEC2
NDEC2
DC12
NDC12
IC40
IC6
NDEC1
DEC1
IC58
IC67
CLOCK
CLOKB
XCLK
IC39
NDECO
4.7K
+5V
PULLG
CLOCK
NCALL
CALL
RD1
IC57
NEXIT
EXIT
RD0
NINTR
CKLRD

FIG.65

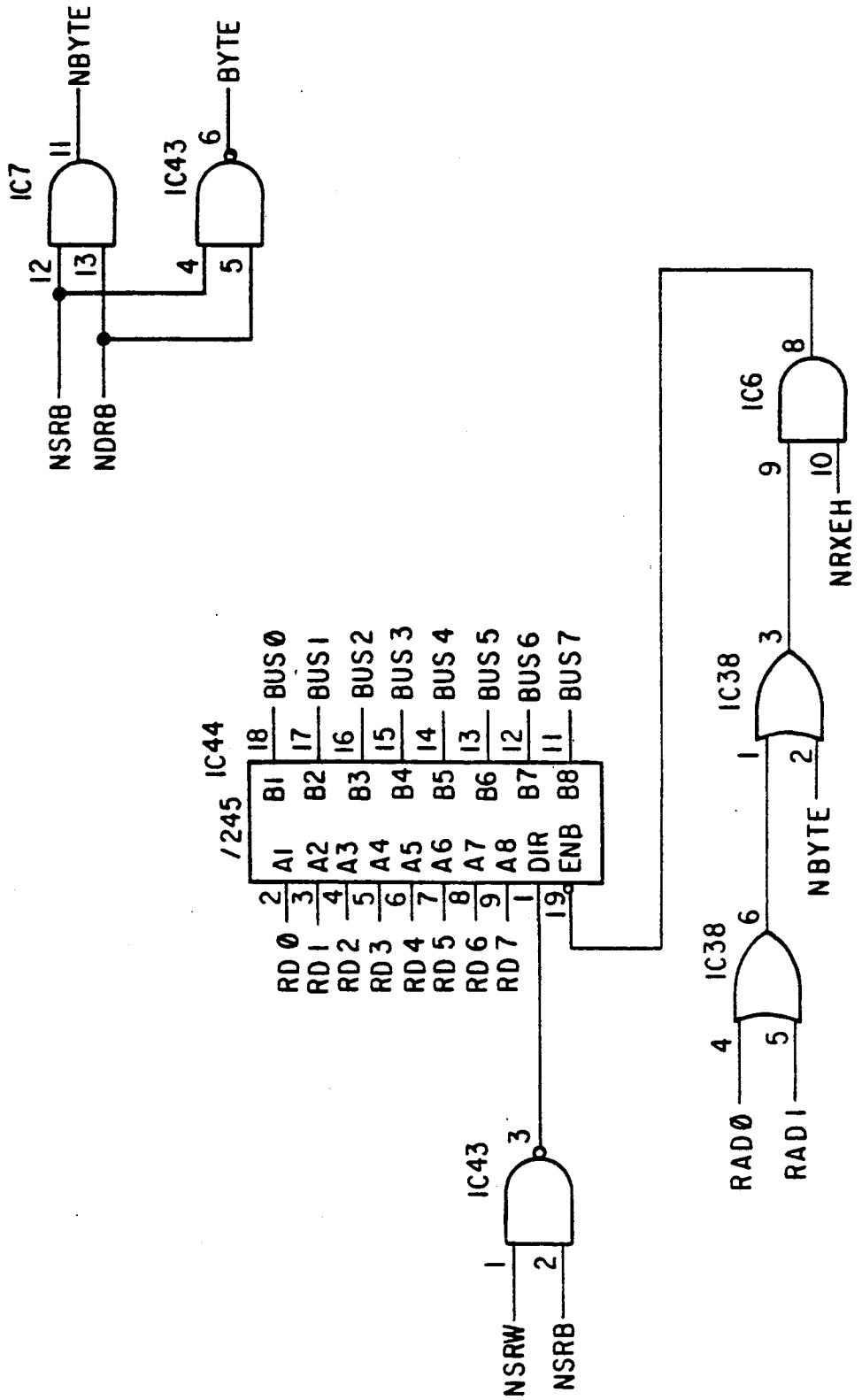

IC7
NBYTE
BYTE
IC43
NSRB
NDRB
IC44
/245
BUS 0
BUS 1
BUS 2
BUS 3
BUS 4
BUS 5
BUS 6
BUS 7
RD 0
RD 1
RD 2
RD 3
RD 4
RD 5
RD 6
RD 7
IC6
NRXEH
IC38
NBYTE
IC38
RAD0
RAD1
IC43
NSRW
NSRB

FIG. 66

FIG. 67

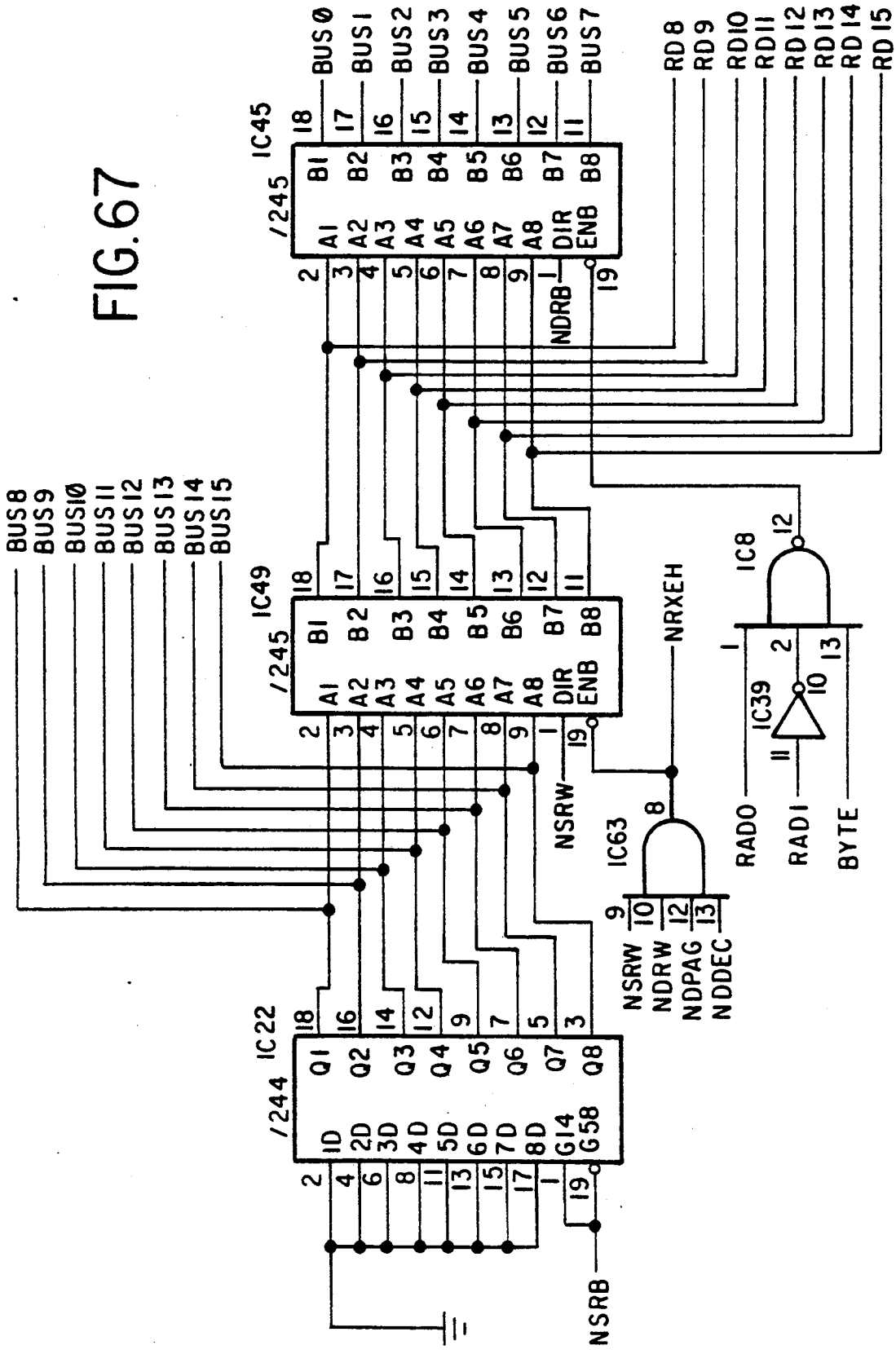

IC45
/245
IC49
/245
IC22
/244
IC8
IC39
IC63
BUS0
BUS1
BUS2
BUS3
BUS4
BUS5
BUS6
BUS7
BUS8
BUS9
BUS10
BUS11
BUS12
BUS13
BUS14
BUS15
RD8
RD9
RD10
RD11
RD12
RD13
RD14
RD15
NDRB
NSRW
NRXEH
NSRB
NSRW
NDRW
NDPAG
NDDEC
RAD0
RAD1
BYTE

FIG. 68

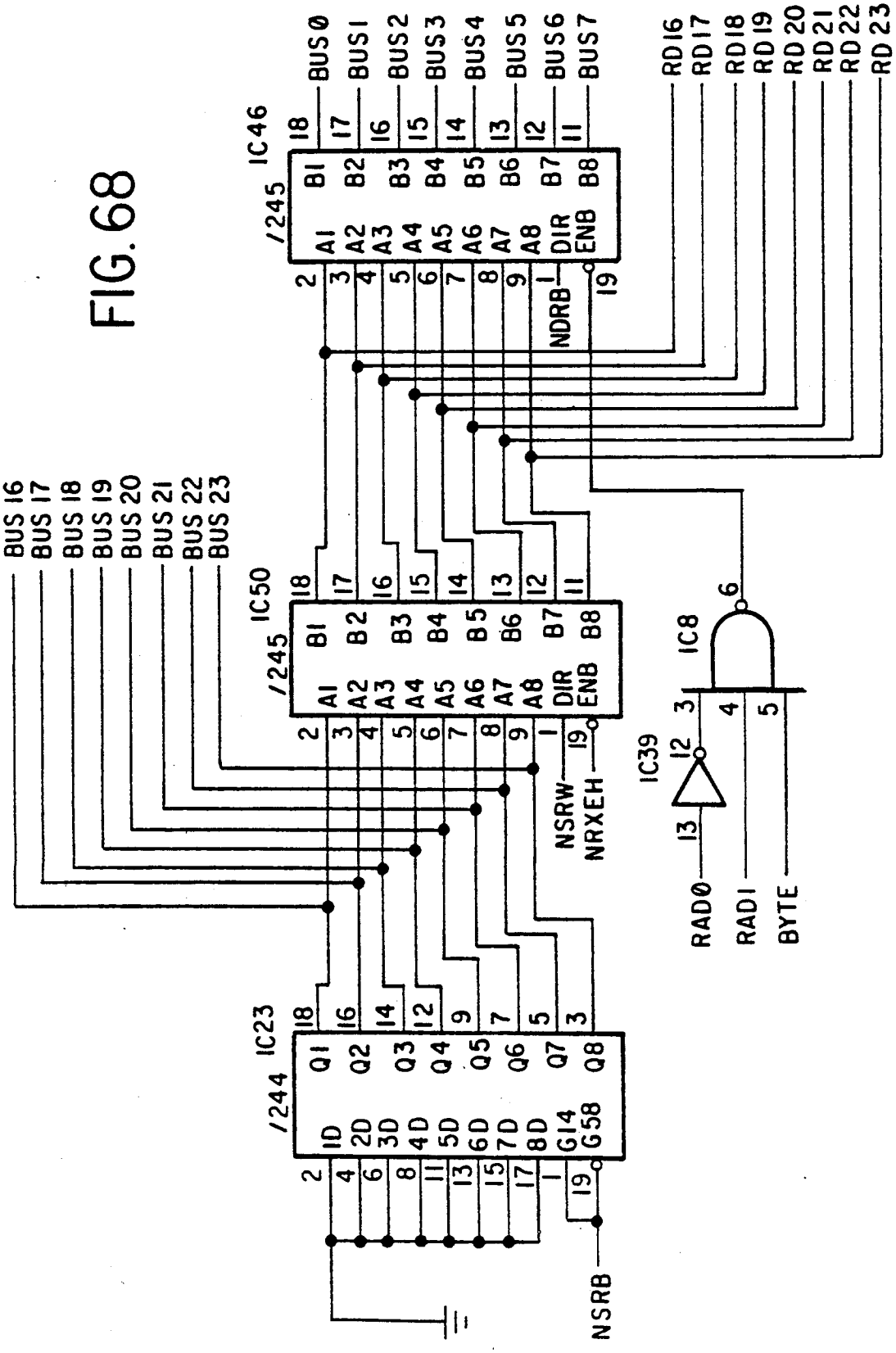
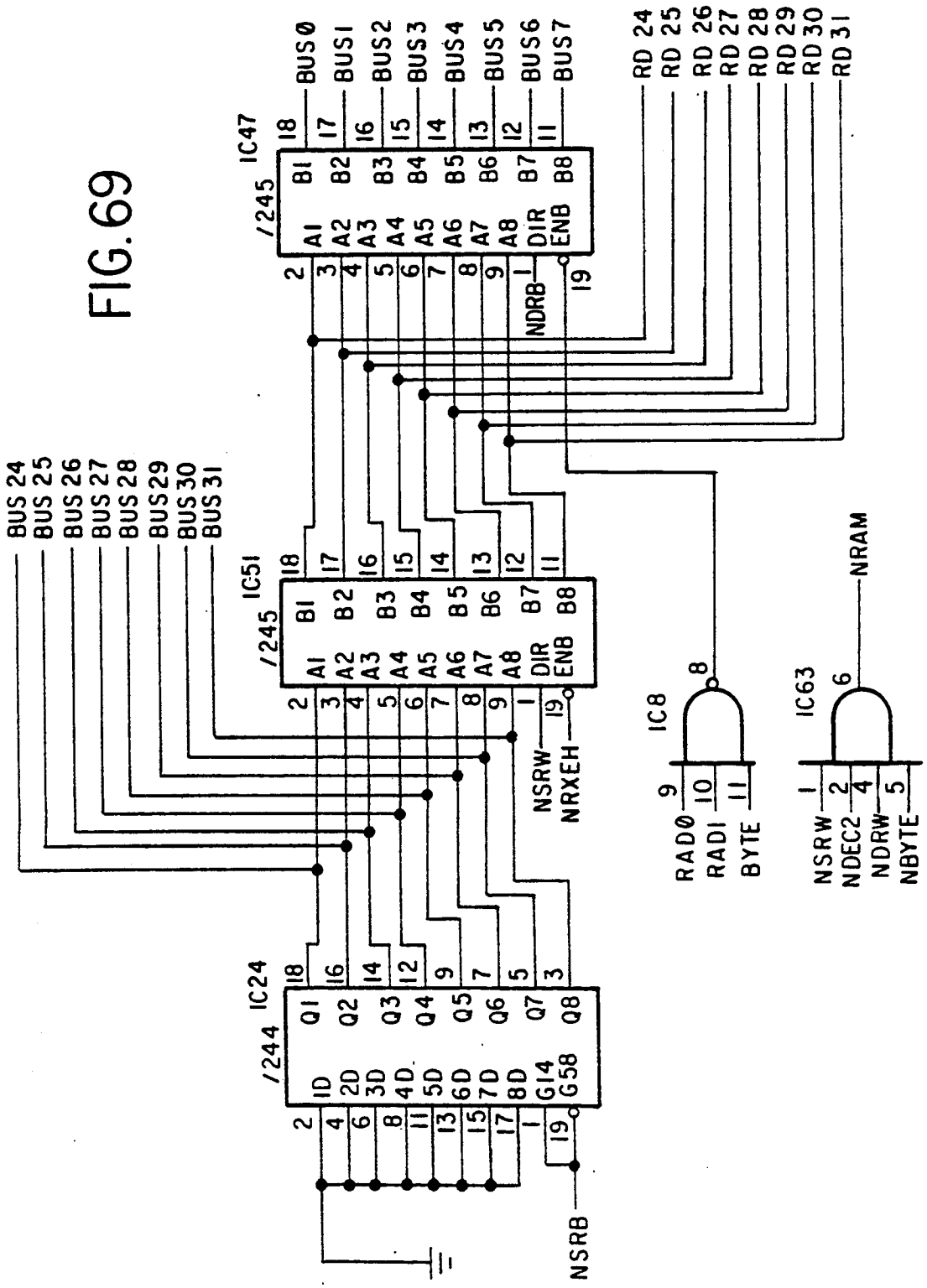

IC46
/245
IC50
/245
IC23
/244
IC8
IC39
BUS 0
BUS 1
BUS 2
BUS 3
BUS 4
BUS 5
BUS 6
BUS 7
BUS 16
BUS 17
BUS 18
BUS 19
BUS 20
BUS 21
BUS 22
BUS 23
RD16
RD17
RD18
RD19
RD20
RD21
RD22
RD23
NDRB
NSRW
NRXEH
NSRB
RAD0
RAD1
BYTE

FIG. 69


IC47
/245
BUS 0
BUS 1
BUS 2
BUS 3
BUS 4
BUS 5
BUS 6
BUS 7
RD 24
RD 25
RD 26
RD 27
RD 28
RD 29
RD 30
RD 31
NDRB
IC51
/245
BUS 24
BUS 25
BUS 26
BUS 27
BUS 28
BUS 29
BUS 30
BUS 31
NSRW
NRXEH
IC8
RAD0
RAD1
BYTE
IC63
NRAM
NSRW
NDEC2
NDRW
NBYTE
IC24
/244
NSRB

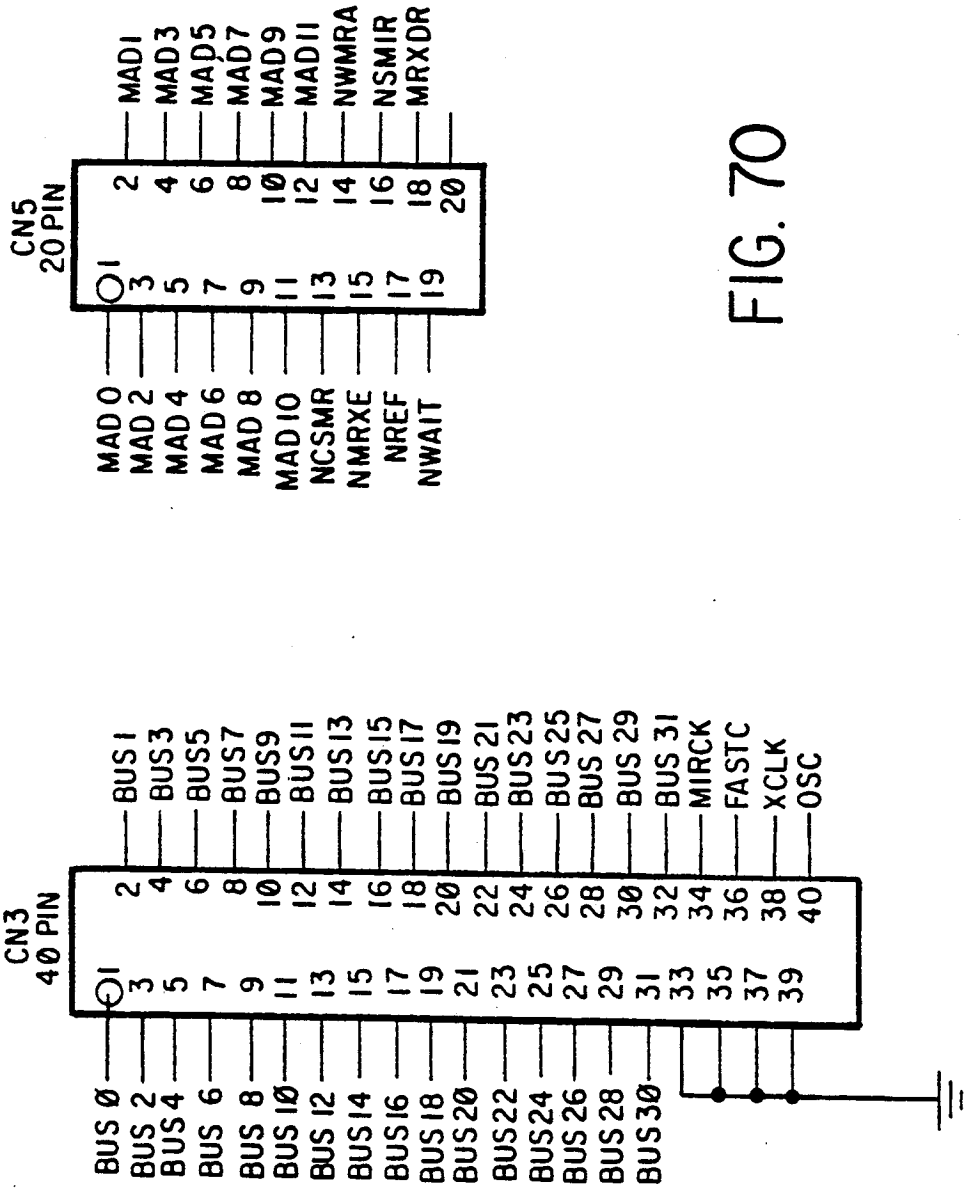

CN5
20 PIN
MAD 0
MAD 2
MAD 4
MAD 6
MAD 8
MAD 10
NCSMR
NMRXE
NREF
NWAIT
MAD1
MAD3
MAD5
MAD7
MAD9
MAD11
NWMRA
NSMIR
MRXDR
CN3
40 PIN
BUS 0
BUS 2
BUS 4
BUS 6
BUS 8
BUS 10
BUS 12
BUS 14
BUS 16
BUS 18
BUS 20
BUS 22
BUS 24
BUS 26
BUS 28
BUS 30
BUS 1
BUS 3
BUS 5
BUS 7
BUS 9
BUS 11
BUS 13
BUS 15
BUS 17
BUS 19
BUS 21
BUS 23
BUS 25
BUS 27
BUS 29
BUS 31
MIRCK
FASTC
XCLK
OSC

FIG. 70

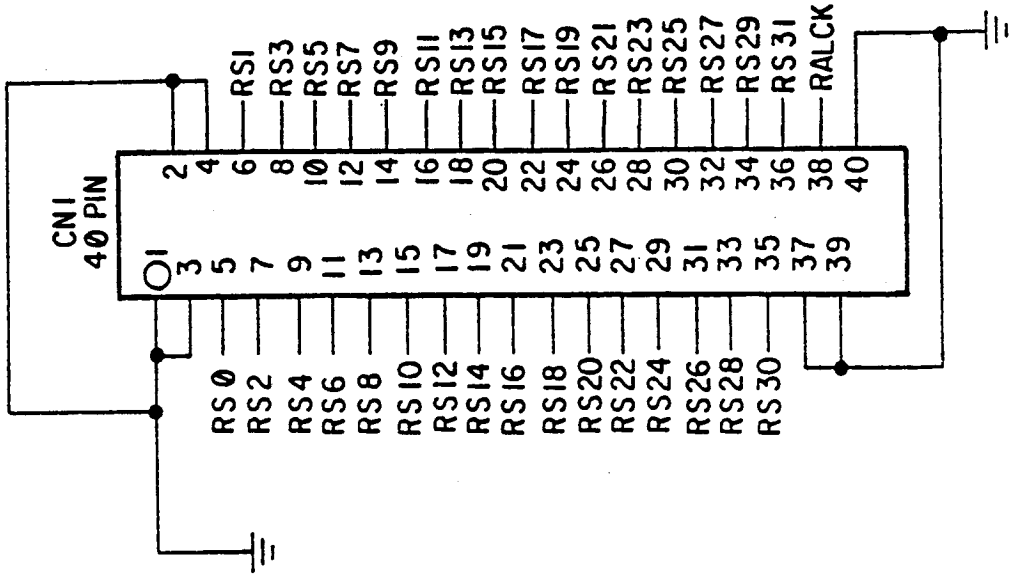

CN1
40 PIN
RS1
RS3
RS5
RS7
RS9
RS11
RS13
RS15
RS17
RS19
RS21
RS23
RS25
RS27
RS29
RS31
RALCK
RS0
RS2
RS4
RS6
RS8
RS10
RS12
RS14
RS16
RS18
RS20
RS22
RS24
RS26
RS28
RS30

FIG. 72

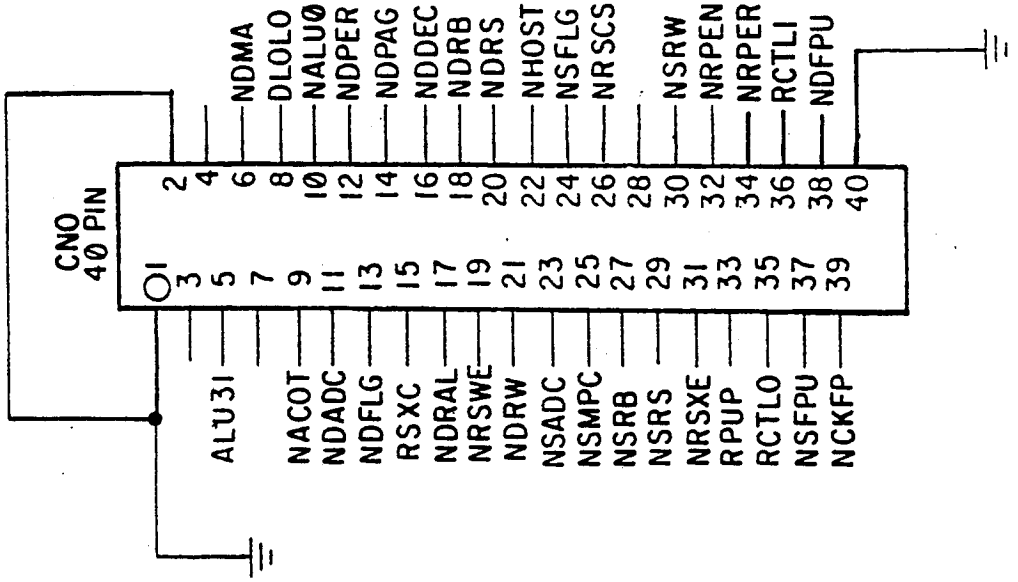

CNO
40 PIN
NDMA
DLOLO
NALU0
NDPER
NDPAG
NDDEC
NDRB
NDRS
NHOST
NSFLG
NRSCS
NSRW
NRPEN
NRPER
RCTLI
NDFPU
ALU31
NACOT
NDADC
NDFLG
RSXC
NDRAL
NRSWE
NDRW
NSADC
NSMPC
NSRB
NSRS
NRSXE
RPUP
RCTLO
NSFPU
NCKFP

FIG. 71

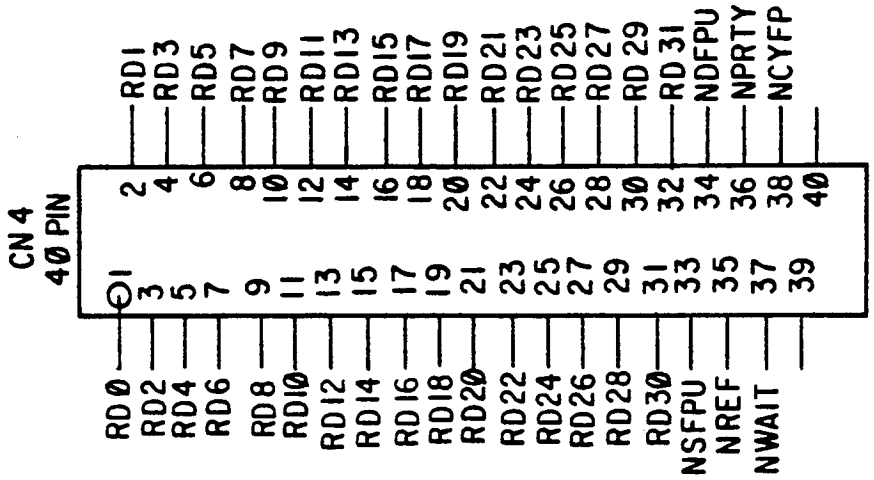

CN4
40 PIN
RD1
RD3
RD5
RD7
RD9
RD11
RD13
RD15
RD17
RD19
RD21
RD23
RD25
RD27
RD29
RD31
NDFPU
NPRTY
NCYFP
RD0
RD2
RD4
RD6
RD8
RD10
RD12
RD14
RD16
RD18
RD20
RD22
RD24
RD26
RD28
RD30
NSFPU
NREF
NWAIT

FIG. 74

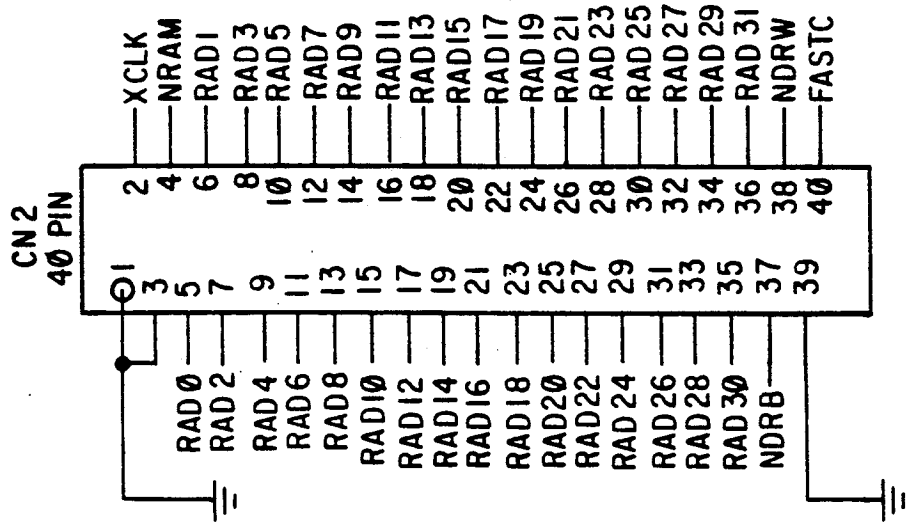

CN2
40 PIN
XCLK
NRAM
RAD1
RAD3
RAD5
RAD7
RAD9
RAD11
RAD13
RAD15
RAD17
RAD19
RAD21
RAD23
RAD25
RAD27
RAD29
RAD31
NDRW
FASTC
RAD0
RAD2
RAD4
RAD6
RAD8
RAD10
RAD12
RAD14
RAD16
RAD18
RAD20
RAD22
RAD24
RAD26
RAD28
RAD30
NDRB

FIG. 73

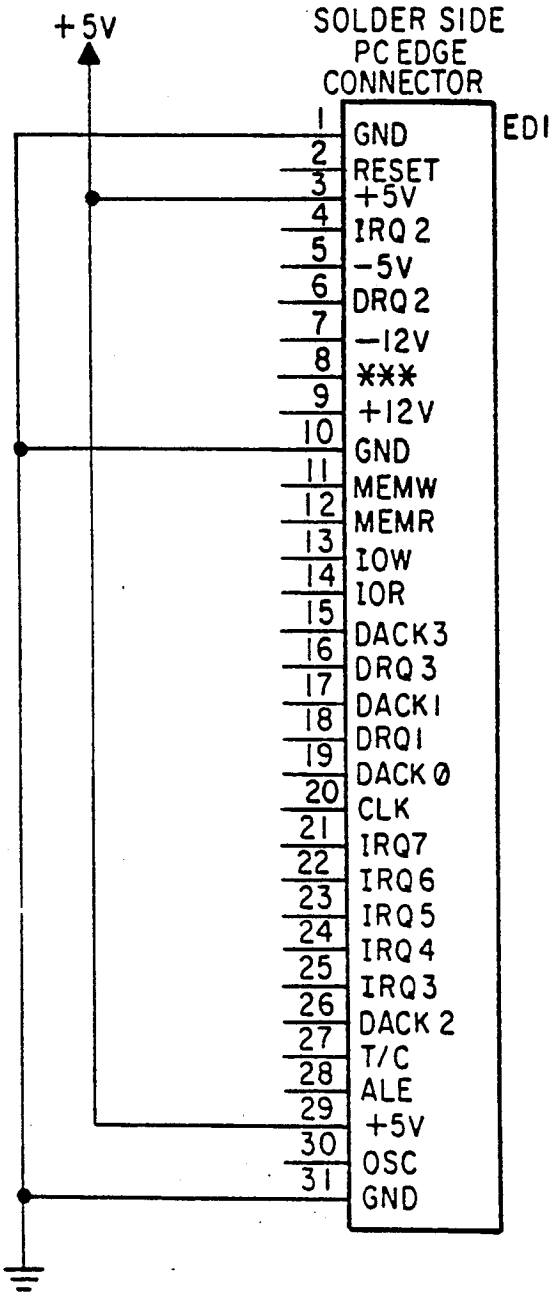

+5V
SOLDER SIDE
PC EDGE
CONNECTOR
EDI
1 GND
2 RESET
3 +5V
4 IRQ 2
5 -5V
6 DRQ 2
7 -12V
8 ***
9 +12V
10 GND
11 MEMW
12 MEMR
13 IOW
14 IOR
15 DACK3
16 DRQ 3
17 DACK1
18 DRQ1
19 DACK 0
20 CLK
21 IRQ7
22 IRQ6
23 IRQ5
24 IRQ4
25 IRQ3
26 DACK 2
27 T/C
28 ALE
29 +5V
30 OSC
31 GND

FIG. 75

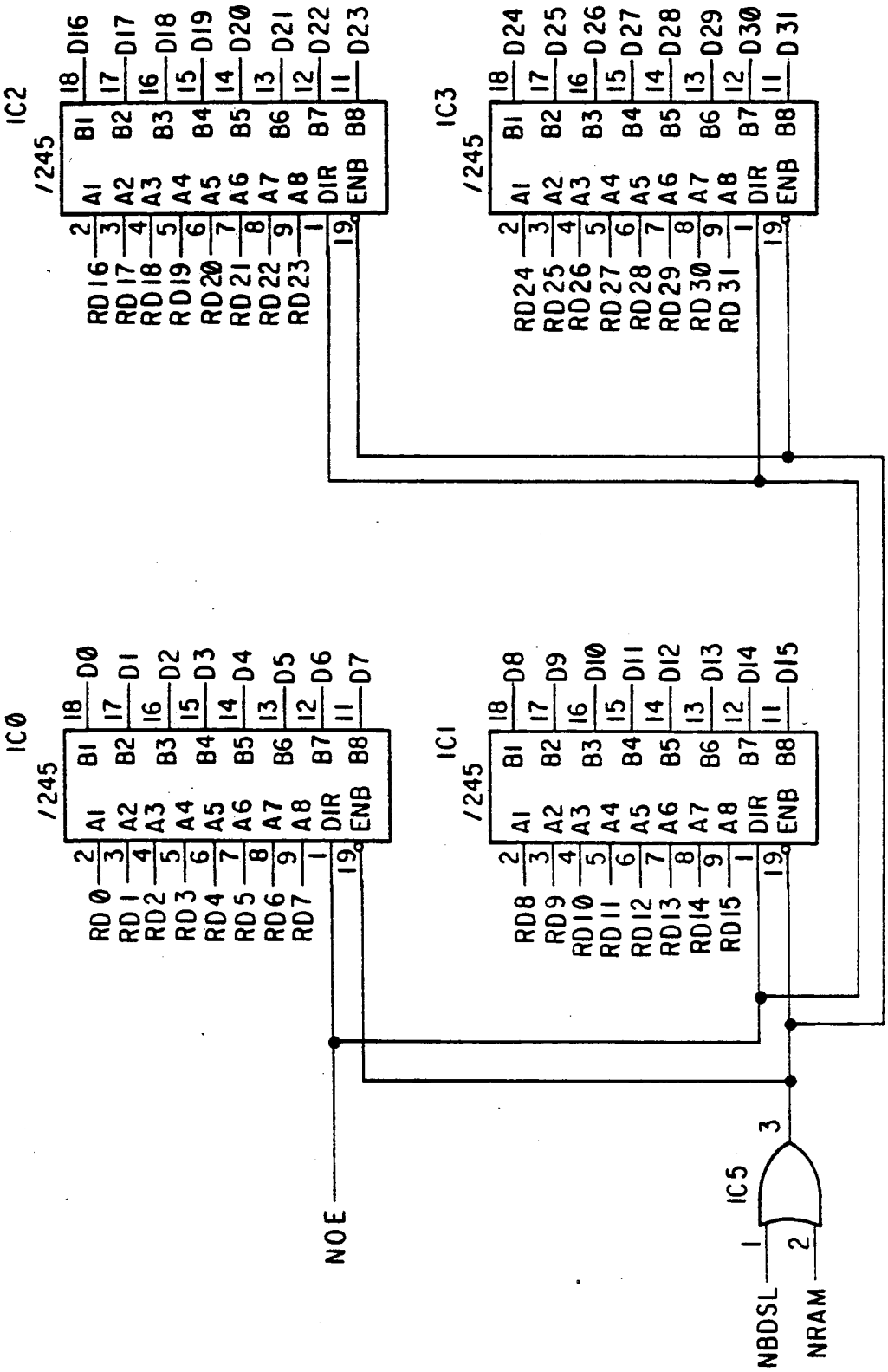


FIG. 76

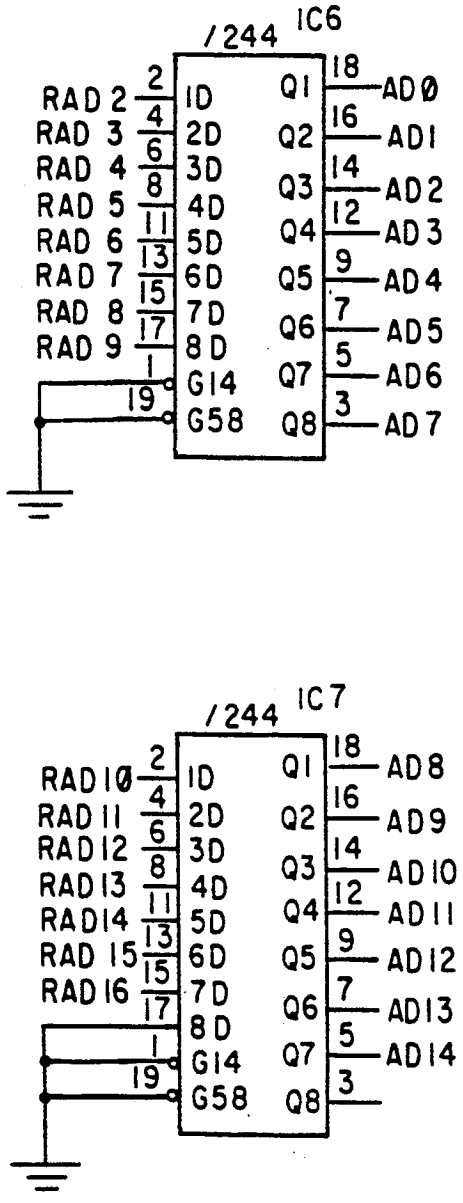

/244
IC6
RAD 2
RAD 3
RAD 4
RAD 5
RAD 6
RAD 7
RAD 8
RAD 9
1D
2D
3D
4D
5D
6D
7D
8D
G14
G58
Q1
Q2
Q3
Q4
Q5
Q6
Q7
Q8
AD0
AD1
AD2
AD3
AD4
AD5
AD6
AD7
/244
IC7
RAD10
RAD11
RAD12
RAD13
RAD14
RAD15
RAD16
AD8
AD9
AD10
AD11
AD12
AD13
AD14

FIG. 77

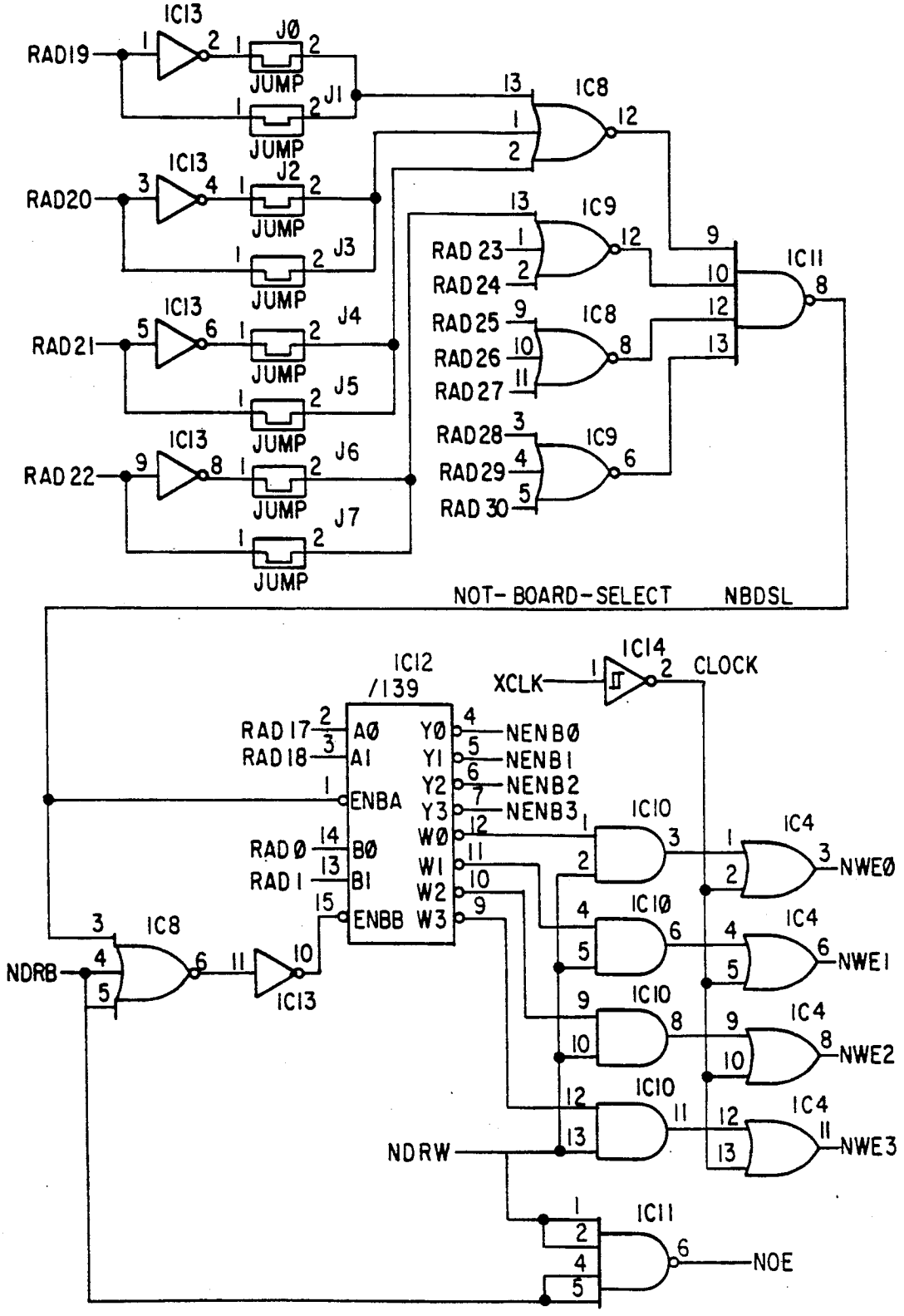

IC13
RAD19
J0
JUMP
J1
IC8
IC13
RAD20
J2
J3
IC9
RAD 23
RAD 24
IC11
IC13
RAD21
J4
J5
RAD25
RAD26
RAD27
IC13
RAD 22
J6
J7
RAD28
RAD29
RAD 30
NOT-BOARD-SELECT
NBDSL
IC12
/139
IC14
XCLK
CLOCK
RAD 17
RAD 18
A0
A1
ENBA
Y0
Y1
Y2
Y3
NENB0
NENB1
NENB2
NENB3
RAD 0
RAD 1
B0
B1
ENBB
W0
W1
W2
W3
IC10
IC4
NWE0
NWE1
NWE2
NWE3
IC8
NDRB
IC13
NDRW
IC11
NOE

FIG. 78

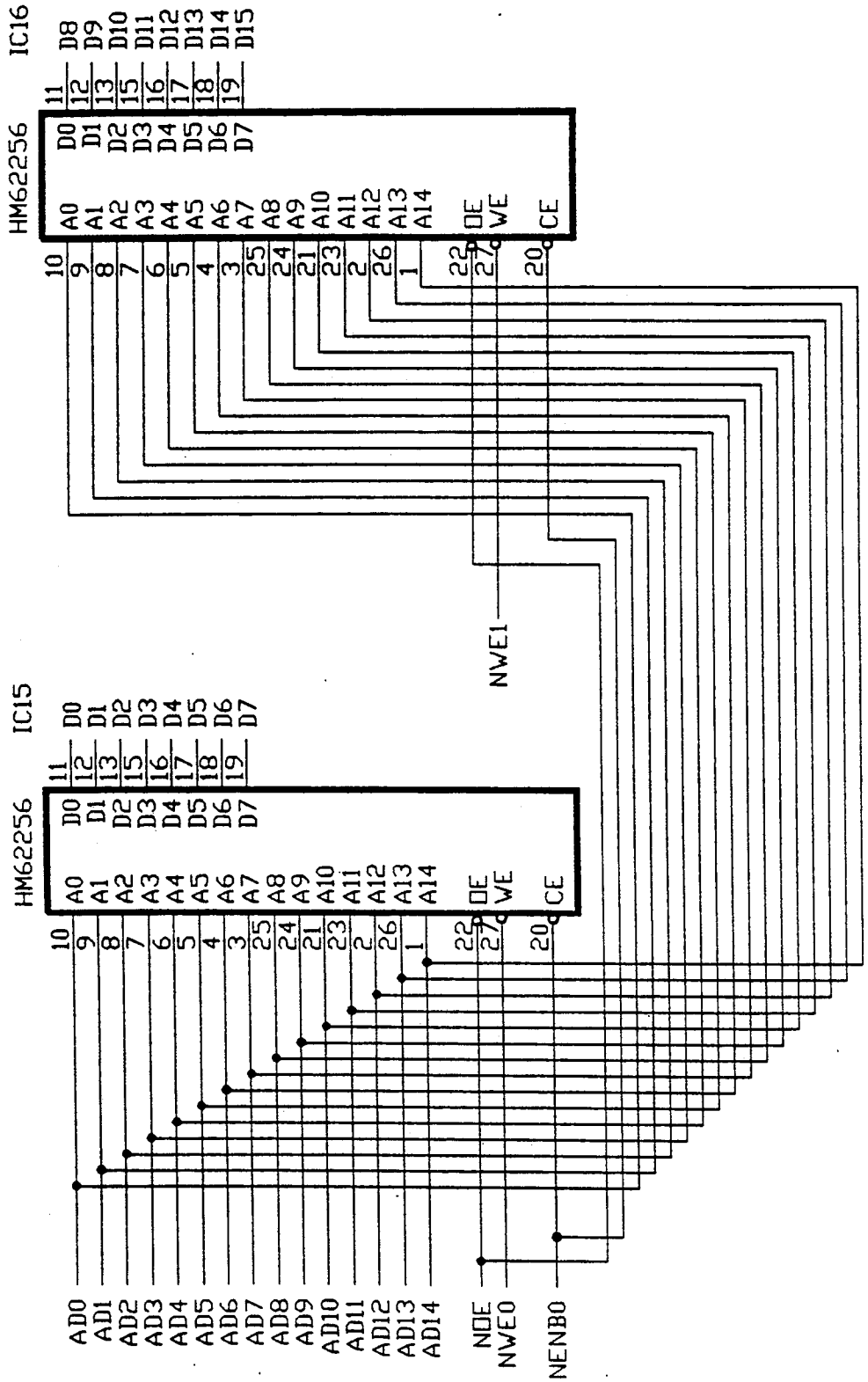

IC16
HM62256
D0 D1 D2 D3 D4 D5 D6 D7
A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 A10 A11 A12 A13 A14
OE WE CE
11 12 13 15 16 17 18 19
10 9 8 7 6 5 4 3 25 24 21 23 2 26 1 22 27 20
D8 D9 D10 D11 D12 D13 D14 D15
NWE1
IC15
HM62256
D0 D1 D2 D3 D4 D5 D6 D7
A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 A10 A11 A12 A13 A14
OE WE CE
11 12 13 15 16 17 18 19
10 9 8 7 6 5 4 3 25 24 21 23 2 26 1 22 27 20
AD0 AD1 AD2 AD3 AD4 AD5 AD6 AD7 AD8 AD9 AD10 AD11 AD12 AD13 AD14
NOE
NWE0
NENB0

FIG. 79

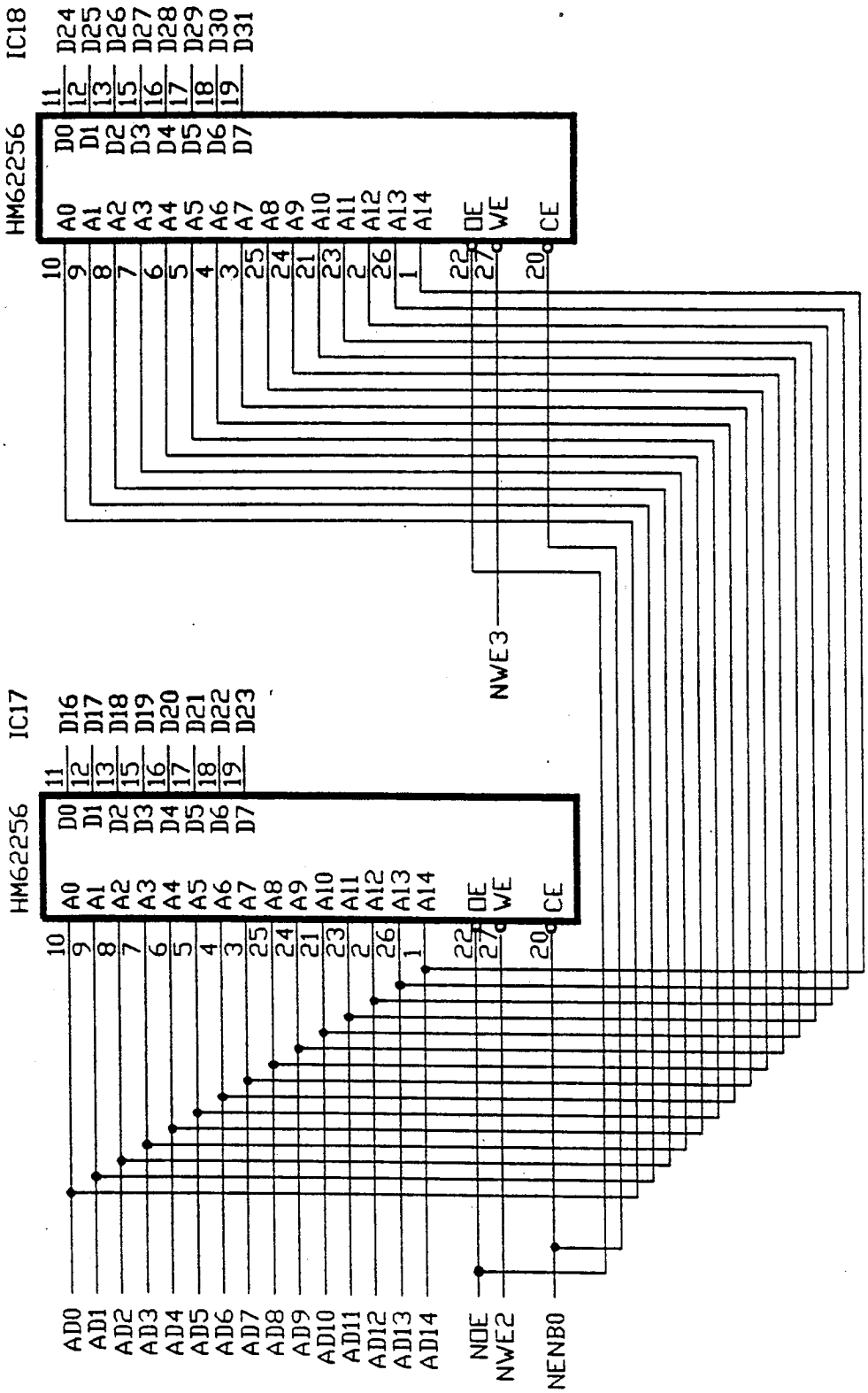

IC18
HM62256
D0 D1 D2 D3 D4 D5 D6 D7
D24 D25 D26 D27 D28 D29 D30 D31
A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 A10 A11 A12 A13 A14
OE WE CE
IC17
HM62256
D0 D1 D2 D3 D4 D5 D6 D7
D16 D17 D18 D19 D20 D21 D22 D23
A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 A10 A11 A12 A13 A14
OE WE CE
NWE3
AD0 AD1 AD2 AD3 AD4 AD5 AD6 AD7 AD8 AD9 AD10 AD11 AD12 AD13 AD14
NOE NWE2 NENB0

FIG.80

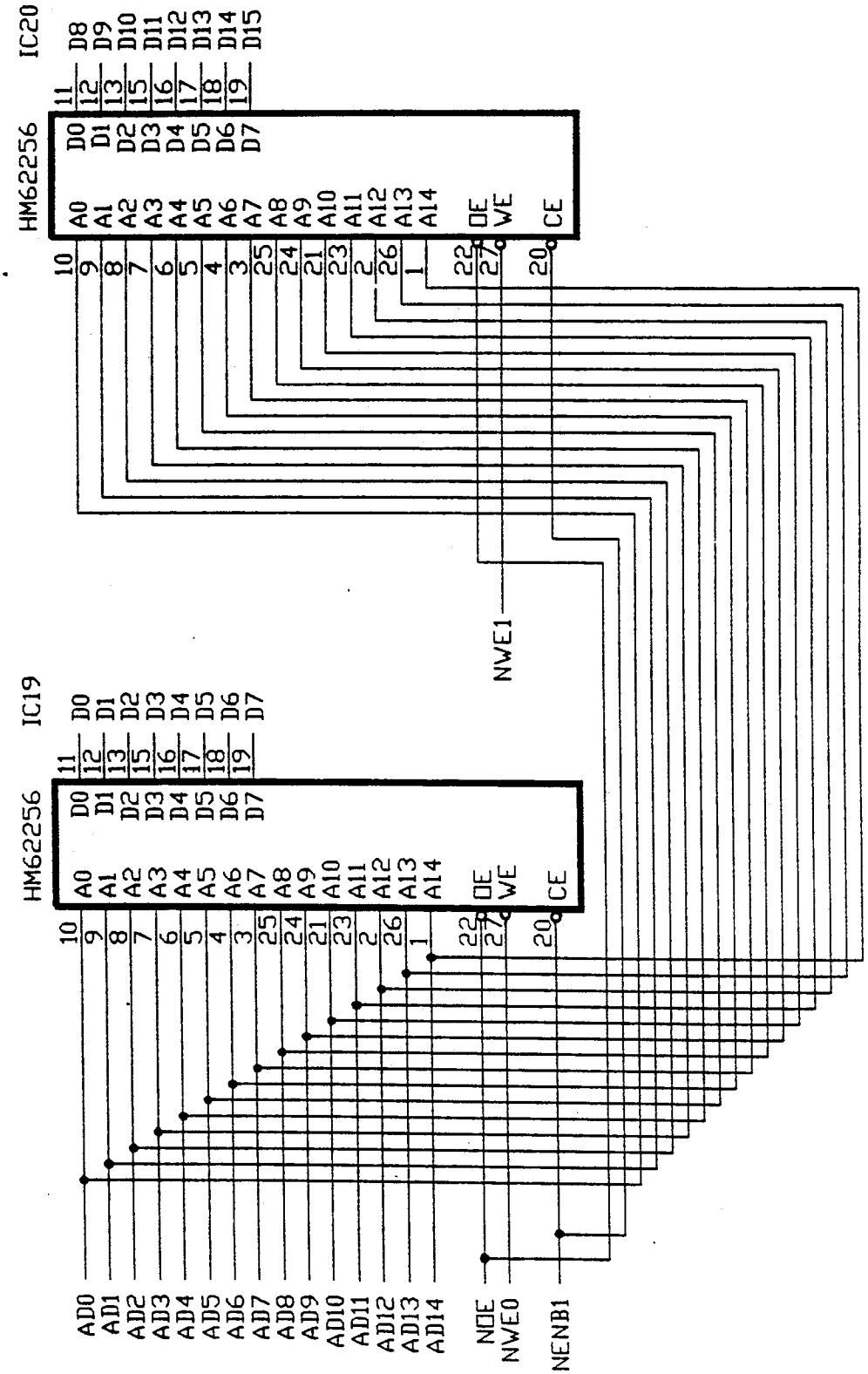

IC20
HM62256
D0 D1 D2 D3 D4 D5 D6 D7
D8 D9 D10 D11 D12 D13 D14 D15
A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 A10 A11 A12 A13 A14
OE WE CE
NWE1
IC19
HM62256
AD0 AD1 AD2 AD3 AD4 AD5 AD6 AD7 AD8 AD9 AD10 AD11 AD12 AD13 AD14
NOE NWE0 NENB1

FIG. 8I

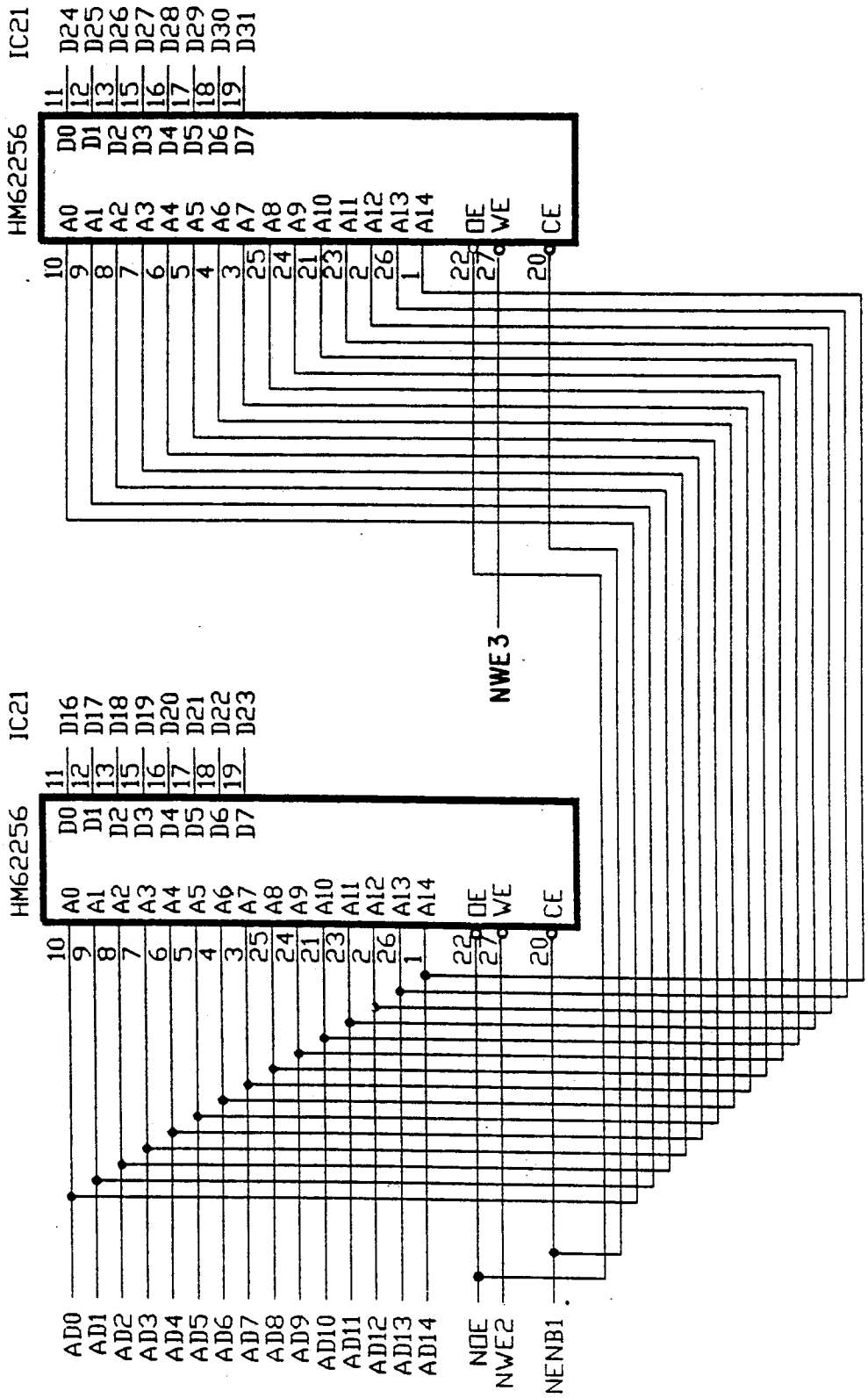

IC21
HM62256
D0 D1 D2 D3 D4 D5 D6 D7
D24 D25 D26 D27 D28 D29 D30 D31
A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 A10 A11 A12 A13 A14
OE WE CE
NWE3
IC21
HM62256
D16 D17 D18 D19 D20 D21 D22 D23
AD0 AD1 AD2 AD3 AD4 AD5 AD6 AD7 AD8 AD9 AD10 AD11 AD12 AD13 AD14
NOE
NWE2
NENB1

FIG. 82

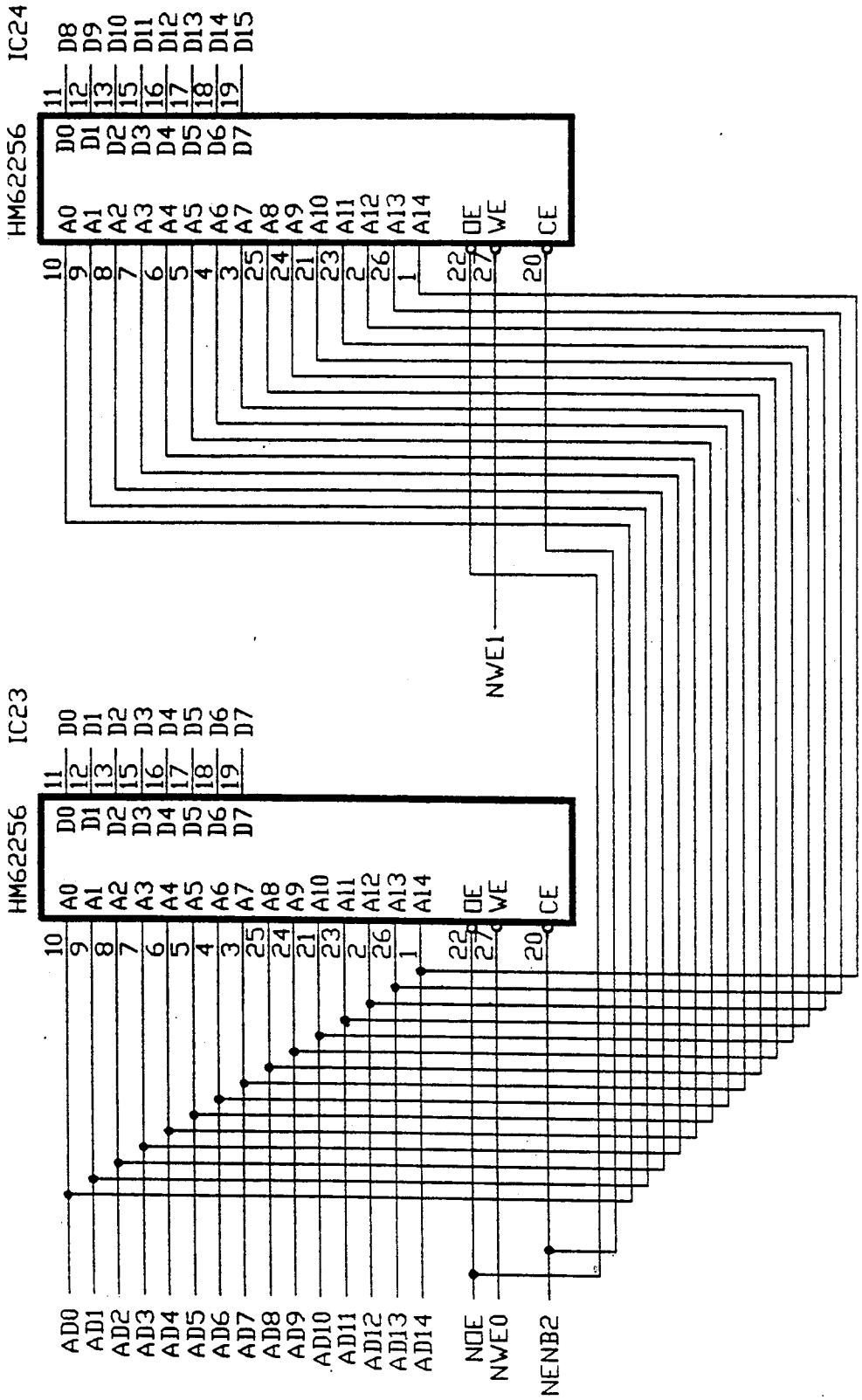

IC24
HM62256
D0 D1 D2 D3 D4 D5 D6 D7
A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 A10 A11 A12 A13 A14
OE WE CE
11 12 13 15 16 17 18 19
10 9 8 7 6 5 4 3 25 24 21 23 2 26 1 22 27 20
D8 D9 D10 D11 D12 D13 D14 D15
IC23
HM62256
D0 D1 D2 D3 D4 D5 D6 D7
A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 A10 A11 A12 A13 A14
OE WE CE
11 12 13 15 16 17 18 19
10 9 8 7 6 5 4 3 25 24 21 23 2 26 1 22 27 20
NWE1
AD0 AD1 AD2 AD3 AD4 AD5 AD6 AD7 AD8 AD9 AD10 AD11 AD12 AD13 AD14
NOE NWE0 NENB2

FIG. 83

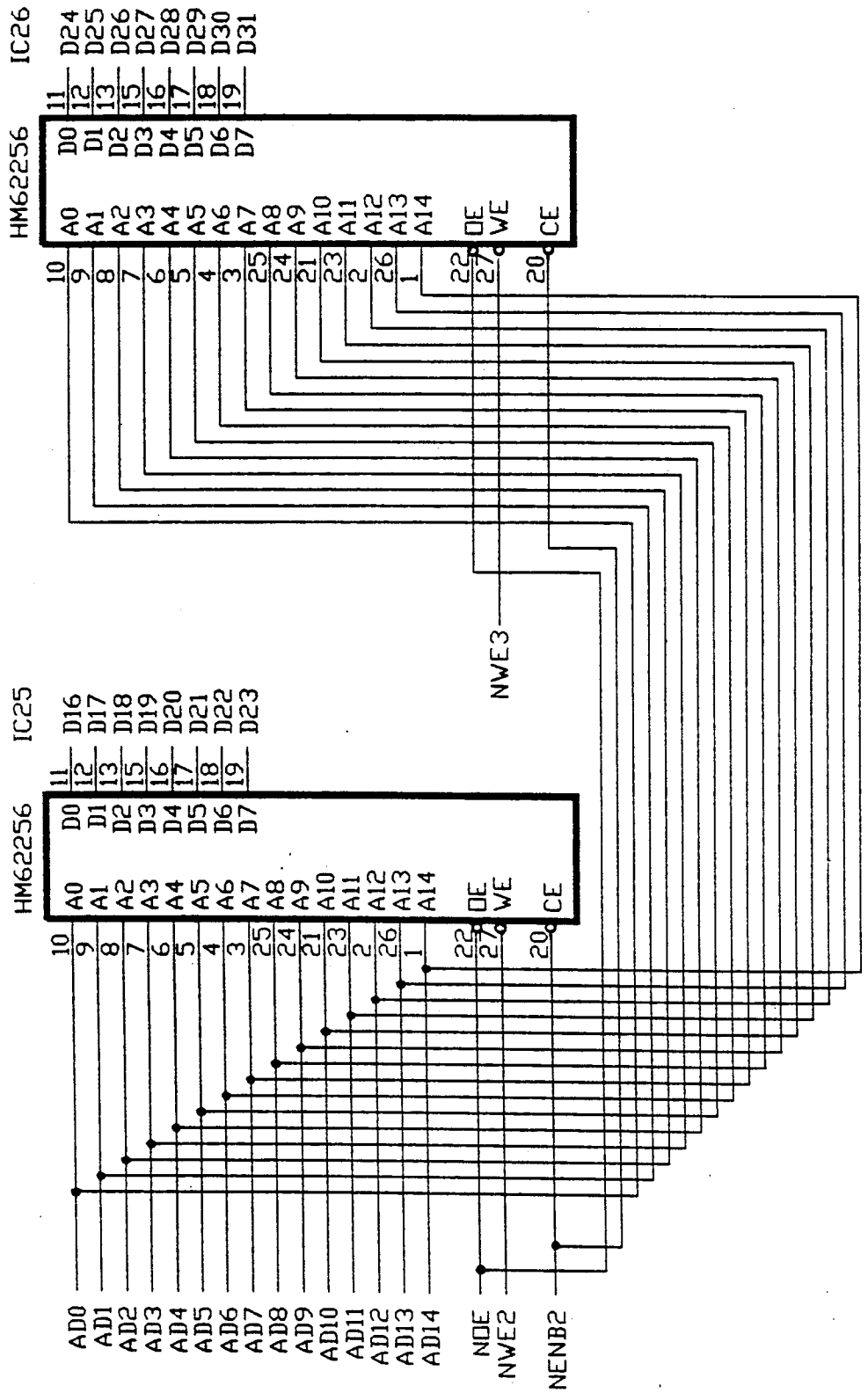

IC26
HM62256
D0 D1 D2 D3 D4 D5 D6 D7
11 12 13 15 16 17 18 19
D24 D25 D26 D27 D28 D29 D30 D31
A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 A10 A11 A12 A13 A14
10 9 8 7 6 5 4 3 25 24 21 23 2 26 1
OE WE CE
22 27 20
NWE3
IC25
HM62256
D0 D1 D2 D3 D4 D5 D6 D7
11 12 13 15 16 17 18 19
D16 D17 D18 D19 D20 D21 D22 D23
A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 A10 A11 A12 A13 A14
10 9 8 7 6 5 4 3 25 24 21 23 2 26 1
OE WE CE
22 27 20
AD0 AD1 AD2 AD3 AD4 AD5 AD6 AD7 AD8 AD9 AD10 AD11 AD12 AD13 AD14
NOE
NWE2
NENB2

FIG. 84

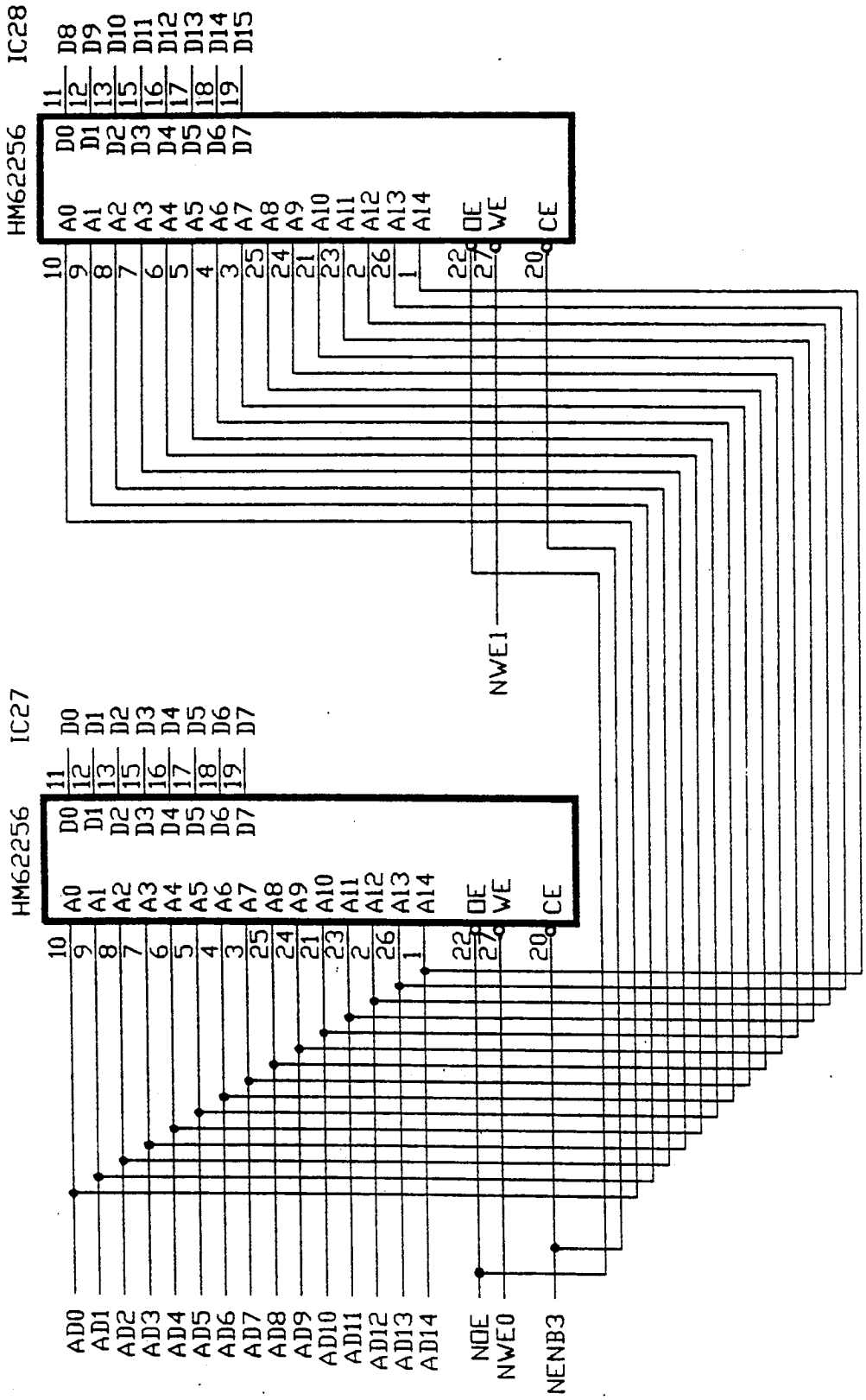

IC28
HM62256
D0 D1 D2 D3 D4 D5 D6 D7
11 12 13 15 16 17 18 19
D8 D9 D10 D11 D12 D13 D14 D15
A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 A10 A11 A12 A13 A14
10 9 8 7 6 5 4 3 25 24 21 23 2 26 1
OE WE CE
22 27 20
NWE1
IC27
HM62256
D0 D1 D2 D3 D4 D5 D6 D7
11 12 13 15 16 17 18 19
A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 A10 A11 A12 A13 A14
10 9 8 7 6 5 4 3 25 24 21 23 2 26 1
OE WE CE
22 27 20
AD0 AD1 AD2 AD3 AD4 AD5 AD6 AD7 AD8 AD9 AD10 AD11 AD12 AD13 AD14
NOE
NWE0
NENB3

FIG.85

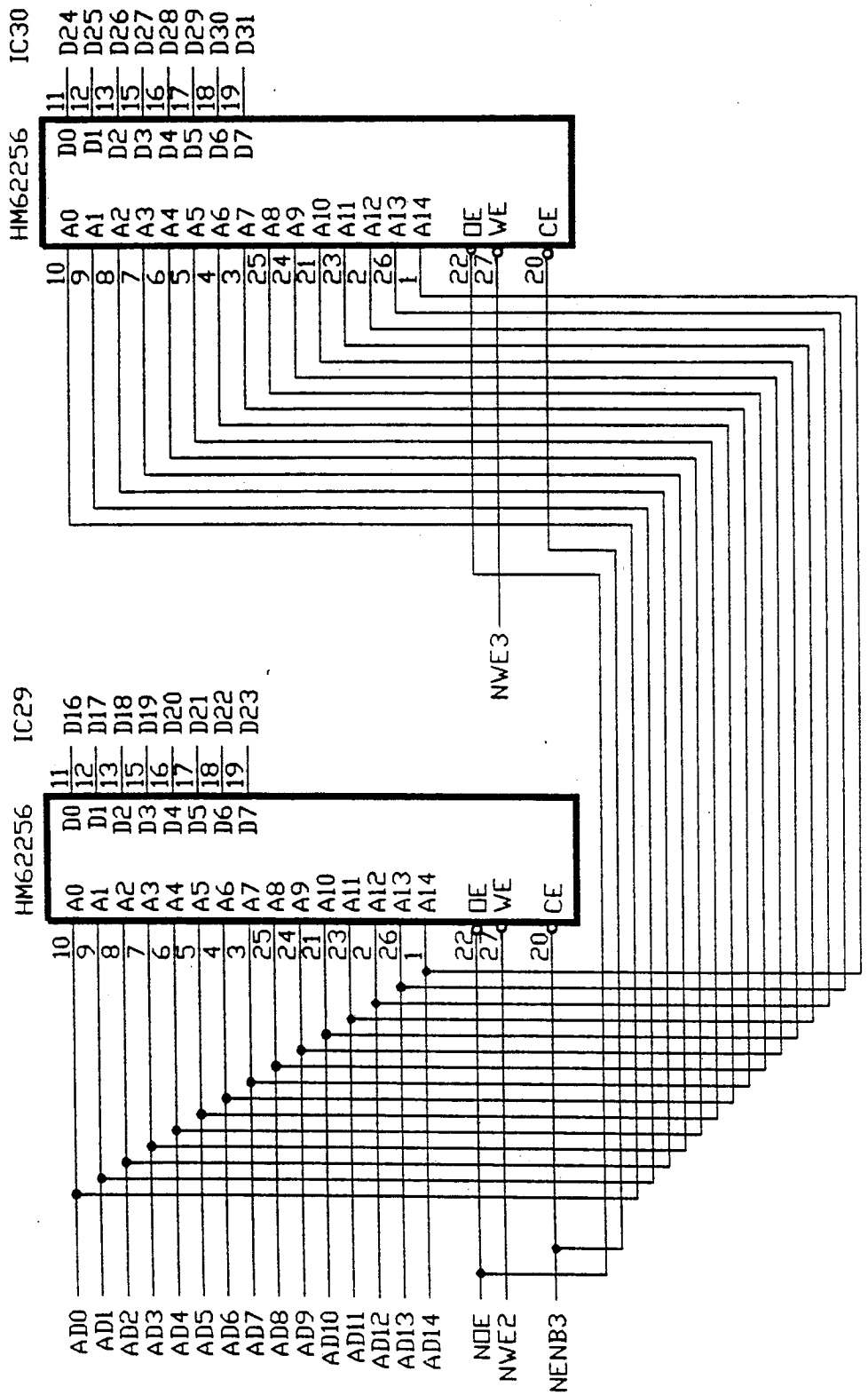

IC30
HM62256
D0 D1 D2 D3 D4 D5 D6 D7
D24 D25 D26 D27 D28 D29 D30 D31
A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 A10 A11 A12 A13 A14
OE WE CE
IC29
HM62256
D0 D1 D2 D3 D4 D5 D6 D7
D16 D17 D18 D19 D20 D21 D22 D23
A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 A10 A11 A12 A13 A14
OE WE CE
NWE3
AD0 AD1 AD2 AD3 AD4 AD5 AD6 AD7 AD8 AD9 AD10 AD11 AD12 AD13 AD14
NOE
NWE2
NENB3

FIG.86

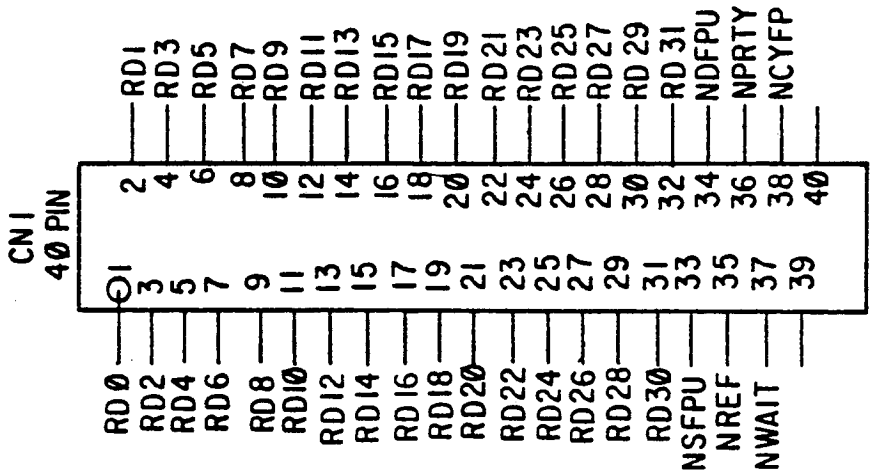

CN1
40 PIN
RD0
RD2
RD4
RD6
RD8
RD10
RD12
RD14
RD16
RD18
RD20
RD22
RD24
RD26
RD28
RD30
NSFPU
NREF
NWAIT
RD1
RD3
RD5
RD7
RD9
RD11
RD13
RD15
RD17
RD19
RD21
RD23
RD25
RD27
RD29
RD31
NDFPU
NPRTY
NCYFP

FIG. 88

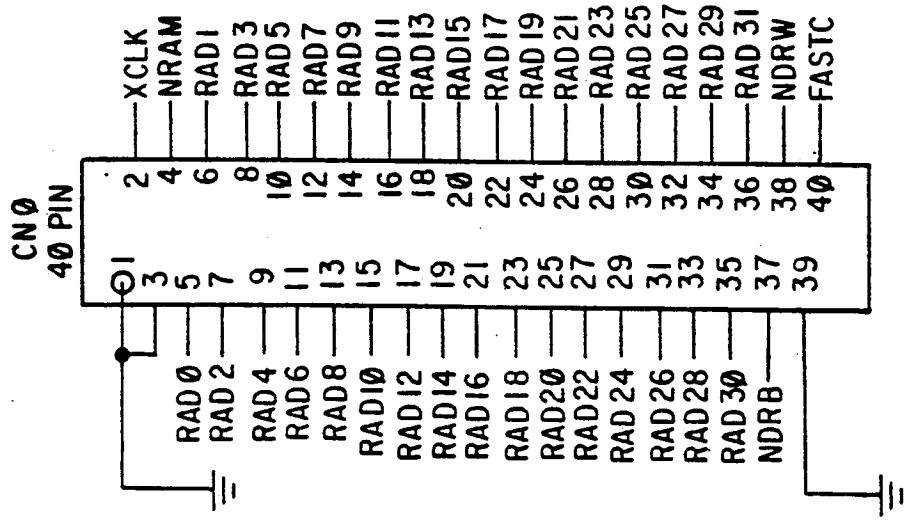

CN0
40 PIN
RAD0
RAD2
RAD4
RAD6
RAD8
RAD10
RAD12
RAD14
RAD16
RAD18
RAD20
RAD22
RAD24
RAD26
RAD28
RAD30
NDRB
XCLK
NRAM
RAD1
RAD3
RAD5
RAD7
RAD9
RAD11
RAD13
RAD15
RAD17
RAD19
RAD21
RAD23
RAD25
RAD27
RAD29
RAD31
NDRW
FASTC

FIG. 87

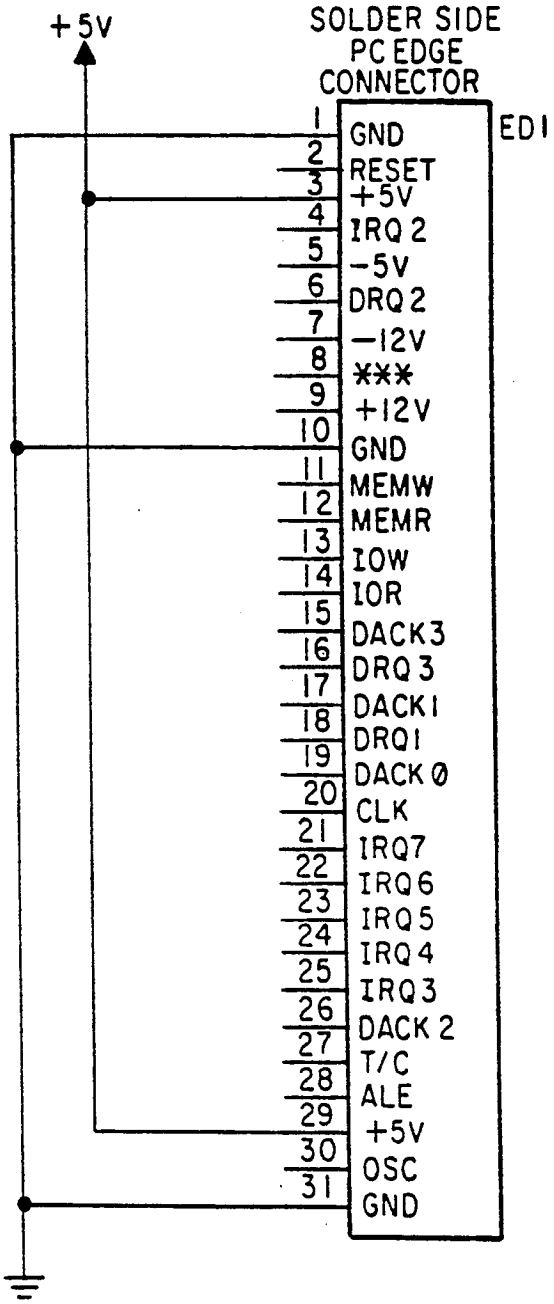

+5V
SOLDER SIDE
PC EDGE
CONNECTOR
ED1
1 GND
2 RESET
3 +5V
4 IRQ 2
5 -5V
6 DRQ 2
7 -12V
8 ***
9 +12V
10 GND
11 MEMW
12 MEMR
13 IOW
14 IOR
15 DACK3
16 DRQ 3
17 DACK1
18 DRQ1
19 DACK 0
20 CLK
21 IRQ7
22 IRQ6
23 IRQ5
24 IRQ4
25 IRQ3
26 DACK 2
27 T/C
28 ALE
29 +5V
30 OSC
31 GND

FIG. 89

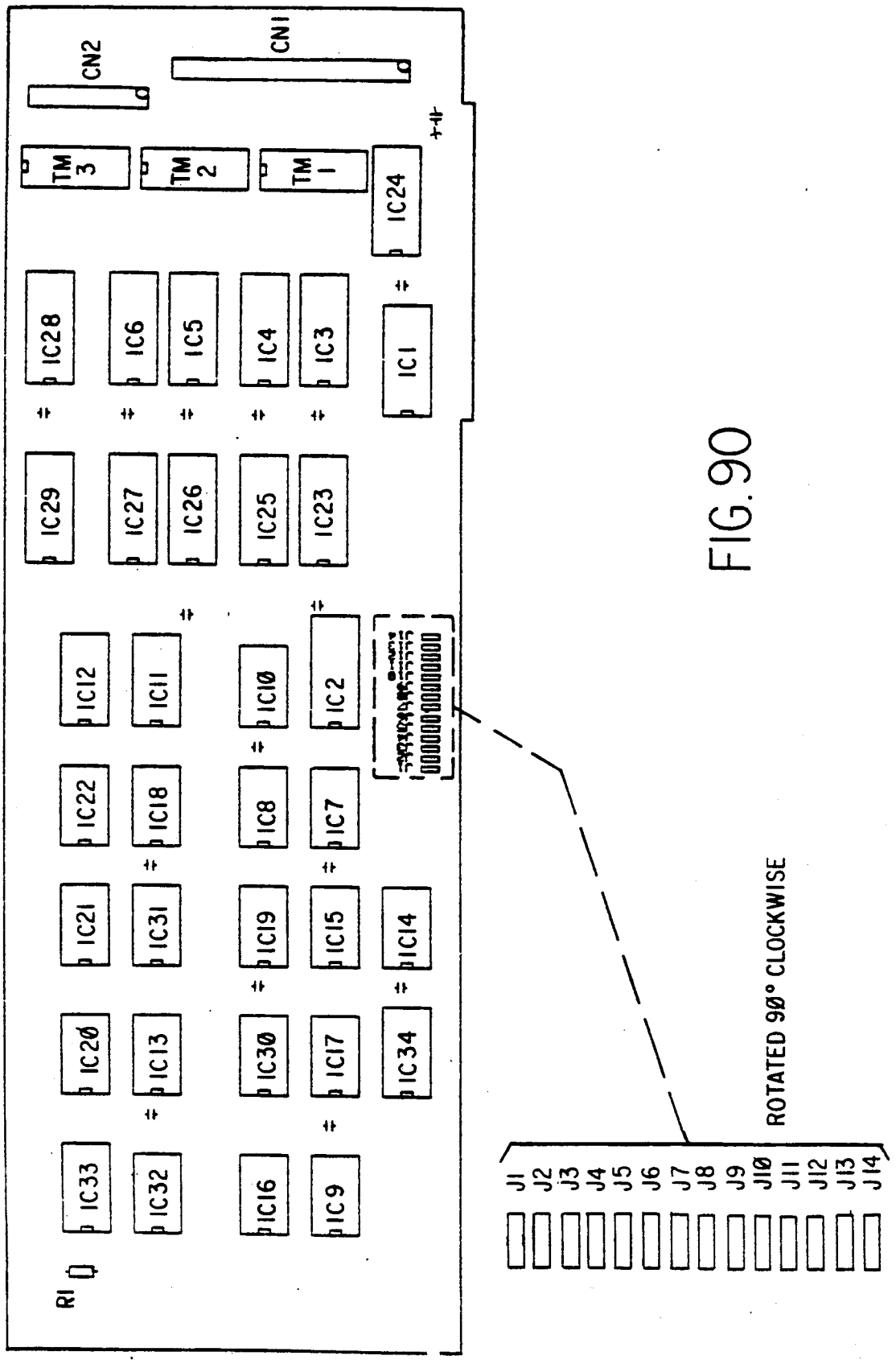

CN2
CN1
TM 3
TM 2
TM 1
IC24
IC28
IC6
IC5
IC4
IC3
IC1
IC29
IC27
IC26
IC25
IC23
IC12
IC11
IC10
IC2
IC22
IC18
IC8
IC7
IC21
IC31
IC19
IC15
IC14
IC20
IC13
IC30
IC17
IC34
IC33
IC32
IC16
IC9
RI
ROTATED 90° CLOCKWISE
J1 J2 J3 J4 J5 J6 J7 J8 J9 J10 J11 J12 J13 J14

FIG. 90

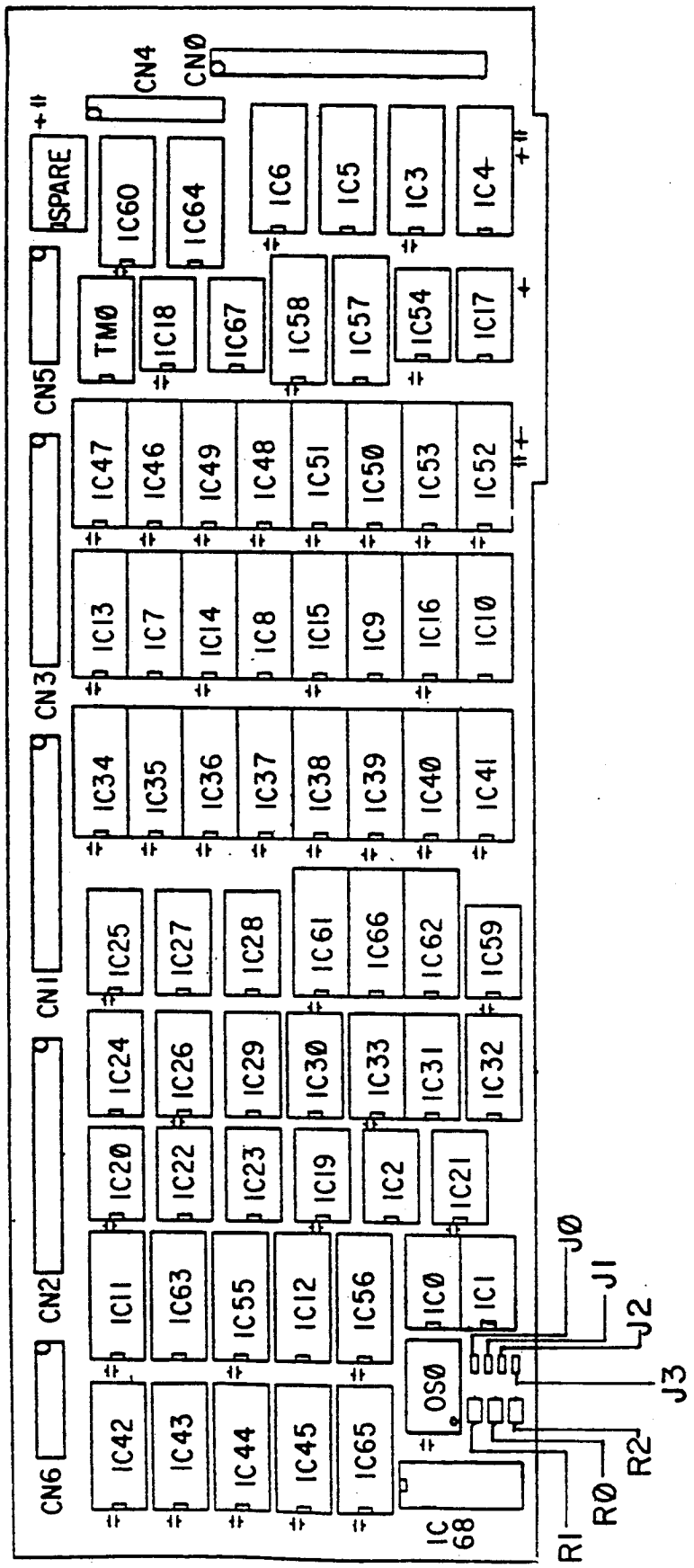

CN4
CN0
SPARE
IC60
IC64
IC6
IC5
IC3
IC4
TM0
IC18
IC67
IC58
IC57
IC54
IC17
CN5
IC47
IC46
IC49
IC48
IC51
IC50
IC53
IC52
CN3
IC13
IC7
IC14
IC8
IC15
IC9
IC16
IC10
CN1
IC34
IC35
IC36
IC37
IC38
IC39
IC40
IC41
IC25
IC27
IC28
IC61
IC66
IC62
IC59
CN2
IC24
IC26
IC29
IC30
IC33
IC31
IC32
IC20
IC22
IC23
IC19
IC2
IC21
CN6
IC11
IC63
IC55
IC12
IC56
IC0
IC1
J0
J1
J2
J3
IC42
IC43
IC44
IC45
IC65
OS0
R2
R0
IC 68
R1

FIG. 91

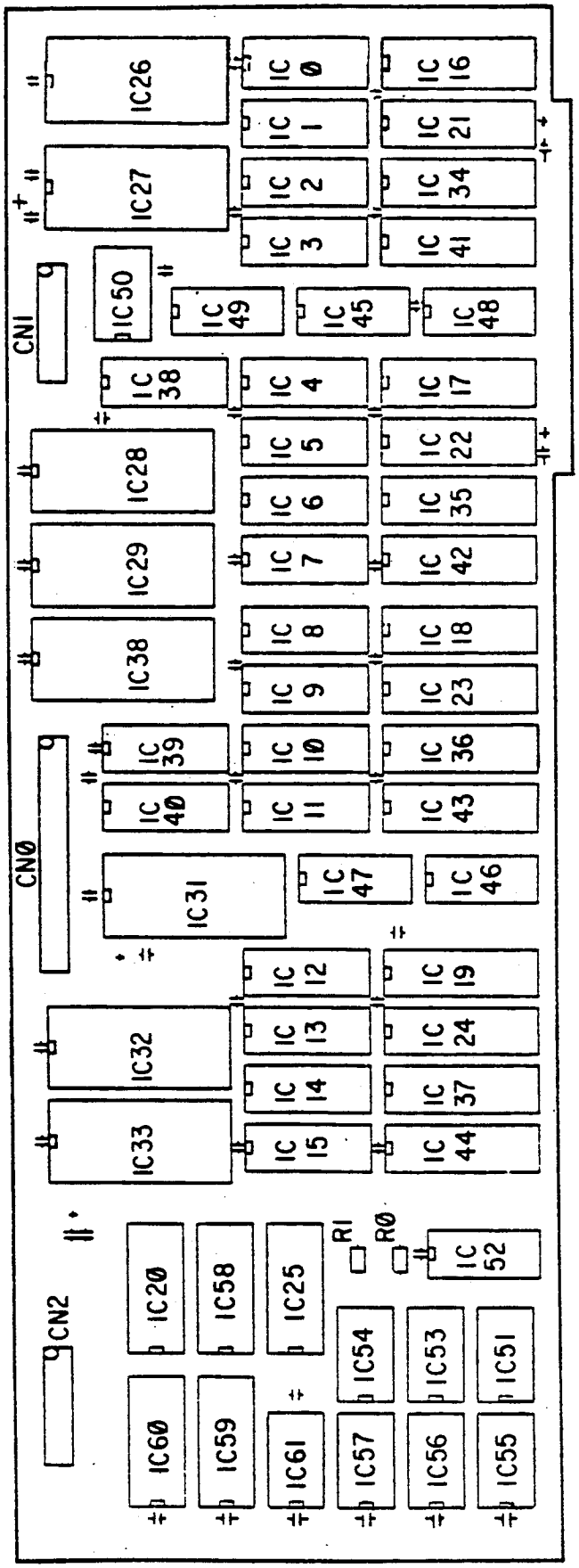

IC26
IC27
IC28
IC29
IC38
IC31
IC32
IC33
IC50
IC20
IC58
IC25
IC60
IC59
IC61
IC54
IC53
IC51
IC57
IC56
IC55
IC52
CN0
CN1
CN2
RI
R0
IC 0
IC 1
IC 2
IC 3
IC 4
IC 5
IC 6
IC 7
IC 8
IC 9
IC 10
IC 11
IC 12
IC 13
IC 14
IC 15
IC 16
IC 17
IC 18
IC 19
IC 21
IC 22
IC 23
IC 24
IC 34
IC 35
IC 36
IC 37
IC 41
IC 42
IC 43
IC 44
IC 45
IC 46
IC 47
IC 48
IC 49
IC 38
IC 39
IC 40

FIG.92

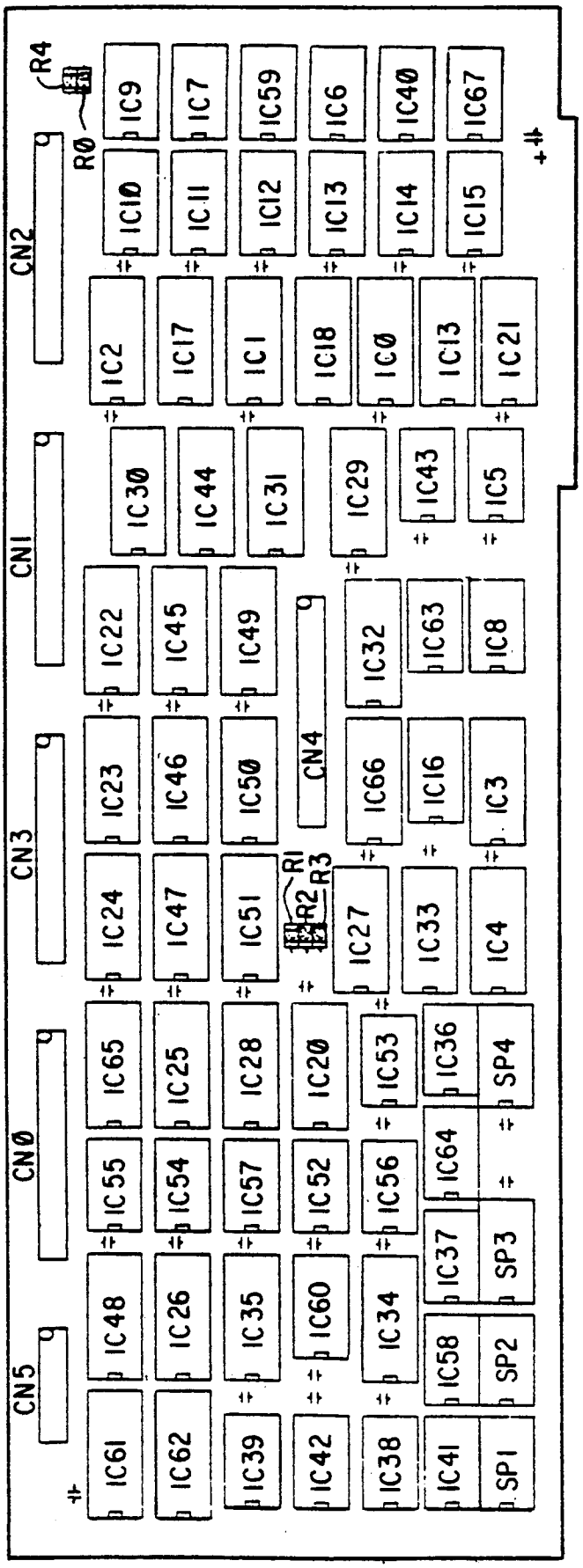

CN0
CN1
CN2
CN3
CN4
CN5
R0
R1
R2
R3
R4
IC0
IC1
IC2
IC3
IC4
IC5
IC6
IC7
IC8
IC9
IC10
IC11
IC12
IC13
IC14
IC15
IC16
IC17
IC18
IC20
IC21
IC22
IC23
IC24
IC25
IC26
IC27
IC28
IC29
IC30
IC31
IC32
IC33
IC34
IC35
IC36
IC37
IC38
IC39
IC40
IC41
IC42
IC43
IC44
IC45
IC46
IC47
IC48
IC49
IC50
IC51
IC52
IC53
IC54
IC55
IC56
IC57
IC58
IC59
IC60
IC61
IC62
IC63
IC64
IC65
IC66
IC67
SP1
SP2
SP3
SP4

FIG. 93

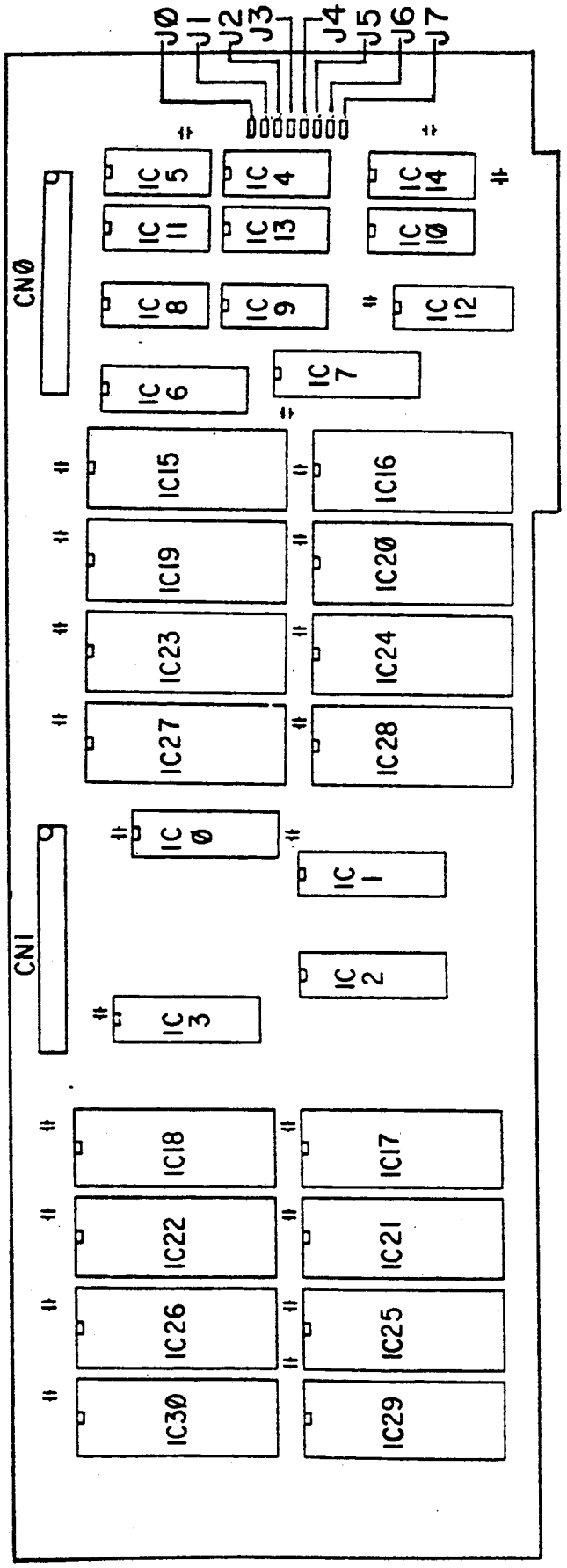

J0
J1
J2
J3
J4
J5
J6
J7
CN0
CN1
IC 0
IC 1
IC 2
IC 3
IC 4
IC 5
IC 6
IC 7
IC 8
IC 9
IC 10
IC 11
IC 12
IC 13
IC 14
IC15
IC16
IC17
IC18
IC19
IC20
IC21
IC22
IC23
IC24
IC25
IC26
IC27
IC28
IC29
IC30

FIG. 94

STACK-MEMORY-BASED WRITABLE INSTRUCTION SET COMPUTER HAVING A SINGLE DATA BUS

BACKGROUND AND SUMMARY OF THE INVENTION

This invention relates to general purpose data processors, and in particular, to such data processors having a writable instruction set with a hardware stack.

This invention is based upon the groundwork laid by our previous CPU/16 patent application Ser. No. 031,473 filed on Mar. 24, 1987, also assigned to the same assignee.

Since the advent of computers, attempts have been made to make computers smaller, with increased memory, and with faster operation. Recently, minicomputers and microcomputers have been built which have the memory capacity of original mainframe computers. Most of these computers are referred to as "complex instruction set" computers. Because of the use of complex instruction sets, these computers tend to be relatively slow in operation as compared to computers designed for specific applications. However, they are able to perform a wide variety of programs because of their ability to process instruction sets corresponding to the source programs run on them.

More recently, "reduced instruction set" computers have been developed which can execute programs more quickly than the complex instruction set computers. However, these computers tend to be limited in that the instruction sets are reduced to only those instructions which are used most often. Infrequently used instructions are eliminated to reduce hardware complexity and to increase hardware speed. Such computers provide limited semantic efficiency in applications for which they are not designed. These large semantic gaps cannot be filled easily. Emulation of complex but frequently used instructions is always a less efficient solution and significantly reduces the initial speed advantage of such machines. Thus, such computers provide limited general applicability.

The present invention provides a computer having general purpose applicability by increasing flexibility while providing substantially improved speed of operation by minimizing complexity as compared to conventional computers. The invention provides this in a way which uses simple, commonly available components. Further the invention minimizes hardware and software tool costs.

More specifically, the present invention provides a computer having a main program memory, a writable micro-program memory, an arithmetic logic unit, and a stack memory, all connected to a single common data bus. In a preferred embodiment, this invention provides a computer interface for use with a host computer. Further, more specifically, both a data stack and a subroutine return address stack are provided, each associated with a pointer which may be set to any element in the corresponding stack without affecting the contents of the stack. Further, there is a direct communication link between the return stack and the main program memory addressing logic, and a direct link between the main program memory and the microcode memory which is separate from the data bus. This provides overlapped instruction fetching and executing, and allows the processing of subroutine calls in parallel with other operations. This parallel capability provides for zero-time-cost (i.e. "free") subroutine calls not possible with other computer architectures.

A major innovation of the present invention over previous writable instruction set, hardware stack computers is the use of a fixed-length machine instruction format that contains an operation code, a jump or return address, and subroutine calling control bits. This innovation, when combined with the direct connection of the return address stack to memory, the use of a hardware data stack, and other design considerations, allows the machine to process subroutine calls, subroutine returns and unconditional branches in parallel with normal instruction processing. Programs which follow modern software doctrine use a large number of small subroutines with frequent subroutine calls. The impact of processing subroutine calls in parallel with other computations is to encourage following modern software doctrine by eliminating the considerable execution speed penalty imposed by other machines for invoking a subroutine.

As a result of the combination of a next instruction address with the opcode for each instruction, the preferred embodiment does not have a program counter in the traditional sense. Except for subroutine return instructions, each instruction contains the address of the next instruction to be executed. In the case of a subroutine return, the next instruction address is obtained from the top value on the return address stack. While this technique is commonly employed at the micro-program level, it has never been used in a high-level language machine. In particular, it has never been used on any machine for the express purpose of processing subroutine calls in parallel with other high level machine operations.

A consequence of the availability of "free" subroutine calls combined with a writable instruction set is a shift of paradigm from the programmer's point of view, opening the as yet unexploited possibility of new methods for writing programs. Conventional computers are viewed by the programmer as executing sequential arrangements of instructions with occasional branches or subroutine calls. Each list is conceived of as directly executing machine functions (although a layer of interpretation may be hidden from the programmer by the hardware.) In a writable instruction set computer with hardware stacks and zero-cost subroutine calls, programs are viewed as a tree-structured database of instructions, in which the "root" of the tree consists of a group of pointers to sub-tree nodes, each sub-tree node consists of another group of pointers to further nodes, and so on out to the tree "leaves" which contain instructions instead of pointers. Flow of control is not viewed as along sequences of instructions, but rather as flow traversing a tree structure, from roots to leaves and then up and down the tree structure in a manner to visit the leaves in sequential order. In the case of this preferred embodiment, the tree structure nodes consist of subroutine call pointers, and the leaves consist of effectively subroutine calls into microcoded primitives. Due to the capability of combining an instruction opcode with a subroutine call, greater efficiency is realized with this design than with what could be realized with a pure tree machine that could only execute operations or process subroutine calls (but not both) with each instruction.

A preferred ALU made in accordance with the invention has a register (the data hi register) on one input for holding intermediate results. On the other input side

is a transparent latch (implemented in the preferred embodiment with standard 74ALS373 integrated circuits) that can either pass data through from the data bus, or retain data present on the bus on the previous clock cycle. This retention capability, along with the capability to direct the contents of the ALU register directly to the bus, allows exchanging the data hi register with the data stack or other registers in two clock cycles instead of the three clock cycles which would be required without this innovation. Since exchanging the top two elements of the data stack is a common operation, this results in a substantial increase in processing speed with very little hardware cost over having multiple intermediate storage registers.

In the preferred embodiment of the invention, a four-way decoder is used to control individual 8-bit banks of the 32-bit program memory. This, combined with data flow logic in the interface between the program memory and the data bus, allows individual access to modification of any byte value in program memory with a single write operation. Conventional computers require a full width memory read, 8-bit modification of the data within a temporary holding register, and a full width memory write operation to update a byte in memory, resulting in substantially slower speeds for such operations. While the preferred embodiment employs this new technique to modify 8 bits of a 32 bit word, this technique is generally applicable to accessing any subset of bits within any length of memory word.

The combination of appropriate software shown in Appendix A that exploits the simultaneous processing of conditional branching opcodes with subroutine calls and the use of hardware stacks combine to form an exceptionally efficient expert system inference engine. An expert system rule base typically is formed by a nested list of "rules" which can invoke other rules via subroutine calls that are only activated under certain conditions. The capability of the preferred embodiment to simultaneously process each rule-oriented subroutine call while evaluating the conditions under which the subroutine call will either be allowed to proceed or will be aborted greatly speeds up processing of expert system programs. Expert systems can run at speeds of over 600,000 inferences per second on the preferred embodiment using a 150ns clock cycle, which is a substantial improvement over existing general purpose computers, and in fact over most special purpose computers.

It will be seen that such a computer offers substantial optimization of throughput while maintaining flexibility. It is also predicted that use of such a machine will positively influence programs and programming languages to have improved structure and lower development cost by not penalizing the modern software principle of breaking programs up into small subroutines.

These and other advantages and features of the invention will be more clearly understood from a consideration of the drawings and the following detailed description of the preferred embodiment.

BRIEF DESCRIPTION OF THE DRAWINGS

Referring to the associated sheets of drawings:

FIGS. **1** and **2** are a system block diagram showing a preferred embodiment made according to the present invention;

FIGS. **3** through **89** show the detailed schematics of the embodiment of FIGS. **1** and **2** organized into groups of compone s placed on five separate printed circuit boards in t preferred embodiment, and;

FIGS. **90** through **95** show a preferred placement of the integrated circuits for FIGS. 3 through 89 on 5 expansion boards for use in conjunction with a host computer.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT SYSTEM HARDWARE

Referring initially to FIG. **1** and FIG. **2**, a system overview of the hardware of a writable instruction set computer **100** made according to the present invention is shown. Computer **100** includes a single 32-bit system data bus **101**. An interface assembly **102** is coupled to bus **101** for interfacing with a host computer **103**, which for the preferred embodiment is an IBM PC/AT, made by International Business Machines, Inc., or equivalent personal computer. Assembly **102** includes a bus interface transceiver **104**, an 8-bit status register **105** for requesting host services, and an 8 bit service request register **106** for the host to request services of computer **100**. In the preferred embodiment, the host interface adapter **107** provides the necessary 8 bit host to 32 bit computer data sizing changes. Hosts in other embodiments would not necessarily be restricted to an 8-bit interface.

Memory stack means are provided in the form of a data stack **108** and a return address stack **109**. Each stack is organized in the preferred embodiment as 4 kilowords of 32 bits per word. Each stack has an associated pointer. Specifically, a data stack pointer **110** is associated with data stack **108**, and a return stack pointer **111** is associated with return stack **109**. As can be seen, each stack pointer receives as input the low 12 bits from bus **101** and has its output connected to the address input of the corresponding stack, as well as through a transmitter **112** or **113** to bus **101**. The data stack data inputs and outputs are buffered through transceiver **114** to provide for better current driving capability. The return stack data may be read from or written to the data bus **101** through the transceiver **116**. In addition, the return stack data may be read from the address counter **117** or written to the address latch **118**.

The RAM address latch **118** and the next address register **119** are the two possible sources for the low 23 bits of address to the program memory (RAM) **121**. The bits **23–30** of program memory address are provided by a page register **120**, allowing up to 2 gigabytes of addressable program memory organized as a group of non-overlapping **8** megabyte pages. When fetching an instruction based on an unconditional branch or subroutine call specified by the address field of the previous instruction, the next address register **119** is used to address memory **121**. For subroutine calls, the contents of the address counter **117** are loaded with the address of the calling program, incremented by 4, and saved in the return stack **109** for use upon subroutine return. The return pointer **111** is decremented before writing to return stack **109**.

Upon subroutine return, return stack **109** provides an address through RAM address latch **118** to address program RAM **121**. RAM address latch **118** retains the address while return stack pointer **111** is incremented to pop the return address off the return stack. In jump, subroutine call, and subroutine return operations, the instruction fetched from program RAM **121** is stored in next address register **119** and the instruction latch **125** at the end of the fetching operation. Thus, each instruction

directly addresses the next instruction through the next address register **119** and program RAM **121**.

It should be noted that the address counter **117** and next address register **119** are not used as a program counter in the conventional sense. In conventional computers, the program counter is a hardware device used as the primary means of generating addresses for program memory whose normal operation is to increment in some manner while accessing sequential instructions. In computer **100**, the next address register **119** is a simple holding register that is used to hold the address of the next instruction to be fetched from memory. The value of the next address register **119** is determined by an address field contained within the previous instruction executed, NOT from incrementing the previous register value. The address counter **117** is not directly involved in computing instruction addresses; it is only used to generate subroutine return addresses. Thus, computer **100** uses address information in each instruction to determine the address of the next instruction to be executed for high level language programs.

Program RAM **121** is organized as a 32-bit program memory addressable for full-words only on evenly divisible by 4 byte addresses. Computer **100** provides a minimum quantity of 512 kilobytes of program memory, with expansion of up to 8 megabytes of program memory possible. A minor modification of the memory expansion boards, employed to allow for decoding more boards, allows use of up to 2 gigabytes of program memory. Program memory words of 32 bits are read from or written to the data bus **101** through transceiver **123**. Additionally, single byte values with the high 24 bits set to **0** may be read and written to any byte (within each 32-bit word) in memory through the byte addressing and data routing block **122**.

Provisions have been made to incorporate a microcode-controlled floating point math coprocessor **124** into the design, but such a processor has not yet been implemented in the preferred embodiment. The floating point coprocessor **124** would take its instructions not from a separate microcode memory, as is the usual design practice, but rather directly from program memory.

The thirty-two bit arithmetic logic unit (ALU) **126** has its A input connected to a data high register (DHI) **127** and its B input connected to the data bus **101** through a transparent latch **128**. The output of the ALU **126** is connected to a multiplexer **129** that provides for data pass-through, single bit shift left and shift right operations, and a byte rotate right operation. The output of ALU **126** is always fed back into the DHI register **127**. The DHI register **127** is connected to data bus **101** through a data transmitter **130**.

A data low register (DLO) **131** is connected via a bidirectional path to the data bus **101**, and its shift in/out signals are connected to the multiplexer **129** to provide a 64-bit shifting capability.

The opcode portion of program RAM **121** is connected to instruction latch **125** for the purpose of holding the next opcode to be executed by the machine. This instruction latch **125** is decoded according to existing interrupt information from interrupt register **126** and conditional branching information from the condition code register **127** to form the contents of the micro-program counter **129**. The micro-program counter **129** forms a 12 bit address into micro- program memory **131**. The three low bits of the address into micro-program memory **131** are generated from a combination of the micro-address constant inputs and decoding of the condition select field to allow for conditional branching. The contents of the output of the decoding/address logic **128** and the micro-program counter **129** may be read to data bus **101** for diagnostic and interrupt processing,purposes through bus driver **130**.

Micro-program memory **131** is a 32-bit high speed memory of 4 kilowords in length. Its data may be read or written to data bus **101** through transceiver **132**, providing a writable instruction set capability. During program execution, its data is fed into the micro-instruction register **133** to provide control signals for operation. Micro-instruction register **133** may be read to data bus **101** through transmitter **134** for diagnostic purposes.

The detailed schematics of the various integrated circuits forming computer **100** are shown in FIGS. **3–89**. Narrative text preceding each group of figures gives descriptions of each signal mnemonic used in the schematics. Other than to identify general features of these circuits, they will not be described in detail, the detail being ascertainable from the hardware themselves. However, some general comments are in order.

Computer **100** in its preferred embodiment is designed for construction on five boards which take five expansion slots in a personal computer. It is addressed with conventional 8088 microprocessor IN and OUT port instructions. It uses 32-bit data paths and 32- bit horizontal microcode (of which bits only **30** are actually used.) It operates on a jumper- and crystal-oscillator controlled micro-instruction cycle period which is preferably set at 150 ns. Most of the logic is the 74ALS series. The ALU is composed of eight 74F181 integrated circuits with carry-lookahead logic. Stack and microcode memory chips are 35 ns CMOS 4-bit chips. Program memory is 120 ns low power CMOS 8-bit memory chips. Since simple primitives are only two clock cycles long, this gives a best case operating speed of 3.3 million basic high level stack operations per second (MOPS). In actual program operation, the average instruction would take just over two cycles, exclusive of complex micro-instructions such as multiplication, division, block memory moves, etc. This, combined with the fact that subroutine calls are zero-cost operations when combined in an instruction with an opcode, gives an average operational speed of approximately 3.5 MOPS.

Variable benchmarks show speed increases of 5 to 10 times over an 80286 running at 8 MHz with zero-wait-state memory. An expert system benchmark shows an even more impressive performance of in excess of 640,000 logical inferences per second.

Instruction decoding requires a 2-cycle minimum on a microcode word definition.

SUMMARY OF FIGURES

The following is a summary of the figures that will be referred to in the detailed description of the preferred embodiment. The figures are organized into general block diagrams and five groups corresponding to the five printed circuit boards in the preferred embodiment.

FIGURE NUMBER	FILE NAME	DESCRIPTION OF CONTENTS
SYSTEM BLOCK DIAGRAM		
1	SBLOCK	ALU AND MEMORY ADDRESS BLOCK DIAGRAM

-continued

FIGURE NUMBER	FILE NAME	DESCRIPTION OF CONTENTS
2	MBLOCK	INSTRUCTION DECODING AND HOST INTERFACE BLOCK DIAGRAM
HOST ADAPTER BOARD		
3	HOST1	HOST ADDRESS DECODER
4	HOST2	READ/WRITE DECODER
5	HOST3	DMA CONTROL LOGIC
6	HOST4	DATA WIDTH CONVERTER FROM HOST
7	HOST5	DATA WIDTH CONVERTER TO HOST
8	HOST6	DATA WIDTH CONVERTER CONTROL LOGIC
9	HOST7	HOST DATA BUS BUFFER
10	HOST8	CONTROL SIGNAL TRANSMITTER - 1
11	HOST9	32-BIT DATA SIGNAL BUS TERMINATORS
12	HOST10	CONTROL SIGNAL TRANSMITTER - 2
13	CON1	HOST EDGE CONNECTOR
14	CON3	HOST TO CPU/32 RIBBON CABLES
The signal descriptions for the host adapter (HOST) board are listed in Appendix D on pages 1 and 2.		
HOST INTERFACE & STACK MEMORY BOARD		
15	MRAM1	MICRO-PROGRAM (0-7)
16	MRAM2	MICRO-PROGRAM (8-15)
17	INT1	STATUS & SERVICE REQUEST REGS
18	INT2	DATA BUFFER TO/FROM HOST
19	INT3	CONTROL SIGNAL SIGNAL BUFFER - 1
20	INT4	CONTROL SIGNAL BUFFER - 2
21	MISC1	SYSTEM CLOCK GENERATOR/OSCILLATOR
22	MISC2	CLOCK CONDITIONING
23	MISC3	BUS SOURCE & DEST DECODERS
24	MISC4	MRAM CONTROL LOGIC
25	STACK1	DATA STACK POINTER
26	STACK2	DATA STACK RAM (0-7)
27	STACK3	DATA STACK RAM (8-15)
28	STACK4	DATA STACK RAM (16-23)
29	STACK5	DATA STACK RAM (24-31)
30	STACK6	RETURN STACK POINTER
31	STACK7	RETURN STACK RAM (0-7)
32	STACK8	RETURN STACK RAM (8-15)
33	STACK9	RETURN STACK RAM (16-23)
34	STAK10	RETURN STACK RAM (24-31)
35	CON2	DATA & CONTROL BUS RIBBON CABLES
36	CON3	HOST TO CPU/32 RIBBON CABLES
37	CON4	DATA TO INTERFACE BOARD RIBBON CABLE
38	CON5	INTERFACE TO ADDRESS BOARD RIBBON CABLE "A"
39	CON6	INTERFACE TO ADDRESS BOARD RIBBON CABLE "B"
40	CON9	PC-BUS POWER/GND
The signal descriptions for the host interface and stack memory (INT) board are listed in Appendix D on pages 3-6.		
ALU & DATA PATH BOARD		
41	MRAM3	MICRO-PROGRAM BITS (16-23)
42A, 42B	DATA1	ALU (0-7)
43A, 43B	DATA2	ALU (8-15)
44A, 44B	DATA3	ALU (16-23)
45A, 45B	DATA4	ALU (24-31)
46	DATA5	ALU CARRY-LOOKAHEAD
47	DATA6	DLO REGISTER
48	DATA7	ALU ZERO DETECT
49	DATA8	SHIFT INPUT CONDITIONING
50	DATA9	ALU FUNCTION CONDITIONING FOR DIVISION
51	CON2	DATA & CONTROL BUS RIBBON CABLES
52	CON4	DATA TO INTERFACE BOARD RIBBON CABLE
53	CON9	PC-BUS POWER/GND

-continued

FIGURE NUMBER	FILE NAME	DESCRIPTION OF CONTENTS
The signal descriptions for the ALU and data path (DATA) board are listed in Appendix D on pages 7-9.		
MEMORY ADDRESS & MICROCODE CONTROL BOARD		
54	MRAM4	MICRO-PROGRAM BITS (24-31)
	ADDR1	intentionally omitted
	ADDR2	intentionally omitted
55	ADDR3	RAM ADDRESS LATCH
56	ADDR4	ADDRESS COUNTER (2-9)
57	ADDR5	ADDRESS COUNTER (10-17)
58	ADDR6	ADDRESS COUNTER (18-31) & (0-1)
59	ADDR7	NEXT ADDRESS & PAGE REGISTERS
60	ADDR8	RETURN STACK CONTROL LOGIC
61	CONT1	INSTRUCTION REGISTER & MICRO-PROGRAM COUNTER
62	CONT2	INTERUPT FLAG REGISTER
63	CONT3	CONDITION CODE REGISTER
64	CONT4	INTERRUPT MICRO-ADDRESS REGISTER
65	CONT5	MISC CONTROL LOGIC
66	RAM1	RAM DATA TO BUS INTERFACE (0-7)
67	RAM2	RAM DATA TO BUS INTERFACE (8-15)
68	RAM3	RAM DATA TO BUS INTERFACE (16-23)
69	RAM4	RAM DATA TO BUS INTERFACE (24-31)
70	CON2	DATA & CONTROL BUS RIBBON CABLES
71	CON5	INTERFACE TO ADDRESS BOARD RIBBON CABLE "A"
72	CON6	INTERFACE TO ADDRESS BOARD RIBBON CABLE "B"
73	CON7	ADDRESS TO RAM BOARDS RIBBON CABLE "A"
74	CON8	ADDRESS TO RAM BOARDS RIBBON CABLE "B"
75	CON9	PC-BUS POWER/GND
The signal instructions for the memory address and microcode control (ADDR) board are listed in Appendix D on pages 10-13.		
MEMORY BOARD		
(Note that up to sixteen memory boards may be used within one system)		
76	MEM1	RAM DATA BUFFER
77	MEM2	RAM ADDRESS BUFFER
78	MEM3	READ/WRITE/OUTPUT CONTROL LOGIC
79	MEM4	RAM BANK 0 BITS (0-15)
80	MEM5	RAM BANK 0 BITS (16-31)
81	MEM6	RAM BANK 1 BITS (0-15)
82	MEM7	RAM BANK 1 BITS (16-31)
83	MEM8	RAM BANK 2 BITS (0-15)
84	MEM9	RAM BANK 2 BITS (16-31)
85	MEM10	RAM BANK 3 BITS (0-15)
86	MEM11	RAM BANK 3 BITS (16-31)
87	CON7	ADDR TO MEMORY BOARD RIBBON CABLE "A"
88	CON8	ADDR TO MEMORY BOARD RIBBON CABLE "B"
89	CON9	PC-BUS POWER/GND
The signal instructions for the memory (MEM) board are listed in Appendix D on page 14.		

DETAILED NARRATIVE FOR THE FIGURES

The Host Interface Adapter. FIGS. 3-14 describe the host interface adapter card (referred to as the "host" card.) The host card included in the preferred embodiment is suited for use in an IBM PC computer or compatible, but other functionally similar embodiments are possible for use with other host computers.

FIG. 3 shows the host address bus decoding logic used to activate the board for operation during a host 103 IN or OUT port operation. Jumpers J1 through J14

are used to select the decoded address to any bank of eight ports in the port address space. FIG. 4 shows the decoders IC11 and IC12 which generate control signals based on the lowest bits of the port addresses. In common usage, the preferred embodiment uses eight output ports and three input ports as follows:

PORT	FUNCTION
OUTPUT	
300	DATA BUS (AUTOMATICALLY SEQUENCED FOR 4 BYTES)
301	MIR (WRITE 4 TIMES JUST LIKE WRITE0)
302	SINGLE STEP BOARD CLOCK
303	START BOARD
304	STOP BOARD
305	SET DMA MODE
306	RESET DATA BUS SEQUENCER & DMA MODE
307	SERVICE REQUEST REG & INTERUPT
INPUT	
300	DATA BUS (AUTOMATICALLY SEQUENCED FOR 4 BYTES)
301	MIR (READ 4 TIMES JUST LIKE READ0)
302	STATUS REGISTER (8 BITS)

FIG. 5 shows the generation of control signals and direct memory access (DMA) handshaking signals for the host interface. The host board is capable of accepting high-speed DMA transfers to or from host computer 103 memory directly to and from computer 100 memory. FIGS. 6–12 show the data paths for conversion between an 8-bit host 103 data bus and the 32-bit data bus 101, as well as the buffering for data and control signals on the ribbon cables connecting the host card to the interface card described next. FIGS. 13–14 show the connector arrangements for the host card to host computer bus connector and for the host card to interface card connectors.

The Interface And Stack Card. The interface and stack card (called the interface card) described by FIGS. 15–40 performs a dual function: It serves as the control for bus transfers from the host card and within computer 100 over data bus 101, and provides both the data stack means 108 and the return stack means 109. FIGS. 15–16 show storage for bits 0–15 of the microcode memory and the micro-instruction register. The micro-instruction format is discussed in Appendix B.

FIG. 17 shows the service request register IC58 which is used by the host computer 103 to request one of 255 possible programmable service types from the computer 100. Also shown is the status register IC57 which is used by computer 100 to signal a request for service from host computer 103. FIGS. 18–20 show data and control signal buffers between the host card and the interface card.

FIGS. 21–22 show the clock generating circuits for computer 100. Jumpers J0 through J3 in FIG. 21, along with a socket to change the crystal oscillator used for OS0 allow selection of a wide range of oscillator frequencies. The preferred frequency for the preferred embodiment is 5.0 million Hertz. FIG. 22 shows that a fast clock FASTC is generated that is several nanoseconds ahead in phase of the system clock XCLK for the purpose of satisfying hold times of chips that require data to be valid after the clock rising edge. FIG. 23 shows the data bus 101 source and destination decoders. The devices in this figure generate signals to select only one device to drive data bus 101 and one device to receive data from bus 101. FIG. 24 shows miscellaneous control gates for microcode memory and the micro-instruction register.

FIGS. 25–28 show the data stack means. The data stack has a 12-bit up/down counter that may be incremented, decremented, or loaded from data bus 101 at the end of every clock cycle. The use of fast static RAM chips for the stack memory itself allows the data stack 108 to be read or written and then the stack pointer 110 to be changed on each clock cycle. FIGS. 30–34 show the return stack means. The implementation of the return stack 109 and return stack pointer 111 is very similar to that of the data stack 108 and data stack pointer 110.

FIGS. 35–40 show connector arrangements for transmitting and receiving signals from other cards in the system and from the host adapter card.

The Data, Arithmetic, and Logic Card. The data, arithmetic and logic card (called the data card) described by FIGS. 41–53 performs all arithmetic and logical manipulation of data for computer 100. FIG. 41 shows storage for bits 16–23 of the microcode memory and the micro-instruction register. The micro-instruction format is discussed in Appendix B.

FIGS. 42A–46 show the arithmetic and logic unit (ALU) 126, bus latch 128, data hi register 127, DHI to data bus 100 driver 130, and ALU multiplexer 129. Data from the DHI register 127 and/or the bus data latch 128 flows through the ALU 126 and multiplexer 129 on each clock cycle, then is written back to the DHI register 127. FIG. 47 shows the DLO register 131.

FIG. 48 shows the logic used to detect when the output of the ALU is exactly zero. This is very useful for conditional branching. FIG. 49 shows the generation of the data bus latch 128 control signal and the shift-in bits to the DLO register 131 and the DHI register 127. These shift-in bits are conditioned to provide capability of one-cycle-per-bit multiplication shift-and-conditional-add and non-restoring division algorithms. FIG. 50 shows the conditioning of ALU 126 input control signals to likewise provide for efficient multiplication and division functions.

FIGS. 51–53 show connector arrangements for transmitting and receiving signals from other cards in the system.

The Address Card. The address card described by FIGS. 54–75 performs the memory addressing functions, microcoded control and branching functions, and memory data manipulation functions for computer 100. FIG. 54 shows storage for bits 24–31 of the microcode memory and the micro-instruction register. The micro-instruction format is discussed in Appendix B.

FIG. 55 shows the arrangement of the RAM address latch 118. The RAM address latch is used to address program memory for all non-instruction operations, for return from subroutine operations, and passes data through for DMA transfers with host 103. FIGS. 56–58 show the address counter 117. The address counter 117 may be incremented and passed through the address latch 118 to step through memory one word at a time during DMA access or block memory operations. The address counter 117 is also incremented when performing a subroutine call operation in order to save a correct subroutine return address in return stack 109. FIG. 59 shows the next address register 119 and page register 120. The next address register is used to store the address field of an instruction that points to the memory address of the next instruction during the instruction fetch and decode operation.

FIG. 60 shows the logic used to control return stack 109 and return stack pointer 111. In particular, this logic implements the subroutine call and return control operations for the return stack means. FIG. 61 shows the instruction latch 125 and micro-program counter 129. FIG. 62 shows the interrupt status register 126. Interrupts are set by a processor condition pulling a "PR" pin of IC53–IC56 low, causing the flip-flop to activate, or by loading a one bit from data bus 101. Any one or more active interrupts causes an interrupt at the next instruction decoding operation. An interrupt mask bit from IC53 pin 5 is used to allow masking of all further interrupts during interrupt processing.

FIG. 63 shows the condition code register 127. This register is set at the end of every clock cycle, and forms the basis of the lowest bit of the next micro-instruction address fetched during the succeeding clock cycle. FIG. 64 shows a special forcing driver for the microcode-memory address that forces an opcode of 1 during interrupt recognition. FIG. 65 shows a timing chain used to control the 2 cycle instruction fetch and decoding operation.

FIGS. 66–69 show the RAM data to data bus 101 transfer logic shown by block 122 on FIG. 1. This transfer logic allows access of arbitrary bytes within the 32-bit memory organization as well as 32-bit full word access on evenly-divisible-by-four memory address locations.

FIGS. 70–75 show connector arrangements for transmitting and receiving signals from other cards in the system.

The Memory Card. The memory card described by FIGS. 76–89 is a single program memory 121 storage card for computer 100. Computer 100 may have one to sixteen of these cards in operation simultaneously to use up to 8 megabytes of memory.

FIG. 76 shows data buffering logic used to satisfy current driving requirements of the memory chips. Similarly, FIG. 77 shows address buffering logic. FIG. 78 shows the memory board selection, bank selection, and chip selection logic. Jumpers J0–J7 may be set to map the memory board to one of 16 non-overlapping 512 kilobyte locations within the first eight megabytes of the available memory space. Only one memory board is activated at a time. Once the memory board is activated, a particular bank of chips (numbered from 0–3) is enabled selecting a 32 kiloword address within the board. If byte memory access is being used, a single chip within the bank is selected for a single byte operation, otherwise all chips within the bank are enabled.

FIGS. 79–86 show the four banks of four RAM chips each.

FIGS. 87–89 show connector arrangements for transmitting and receiving signals from other cards in the system.

SYSTEM SOFTWARE

Computer 100 in this preferred embodiment uses various software packages, including a FORTH kernel, a cross-compiler, a micro-assembler, as well as microcode. The software for these packages, written using MVP-FORTH, are listed in Appendix A. Further, the microcode format is discussed in Appendix B. The User's Manual (less appendices duplicated elsewhere in this document) is included as Appendix C. Some general comments about the software are in order. The Cross-Compiler. The cross-compiler maintains a sealed vocabulary with all the words currently defined for computer 100. At the base of this dictionary are special cross-compiler words such as IF ELSE THEN : and ;. After cross-compilation has started, words are added to this sealed vocabulary and are also cross-compiled into computer 100. Whenever the keyword CROSS-COMPILER is used, any word definitions, constants, variables, etc. will be compiled to computer 100. However, any immediate operations will be taken from the cross-compiler's vocabulary, which is chained to the normal MVP-FORTH vocabulary.

By entering the FORTH word {, the cross-compiler enters the immediate execution mode for computer 100. All words are searched for in the sealed vocabulary for computer 100 and are executed by computer 100 itself. The "START.." "END" that is displayed indicates the start and the end of execution of computer 100. If the execution freezes in between the start and end, that means that computer 100 is hung up. The cross-compiler builds a special FORTH word in computer 100 to execute the desired definition, then performs a HALT instruction. Entering the FORTH word } will leave the computer 100 mode of execution and return to the cross-compiler. No colon definitions or other creation of dictionary entries should be performed while between { and }.

The FORTH word CPU32 will automatically transfer control of the system to computer 100 via its Forth language cold start command. The host MVP-FORTH will then execute an idle loop waiting for computer 100 to request services. The word BYE will return control back the host's MVP FORTH.

The current cross-compiler can not keep track of the dictionary pointer DP, etc., in computer 100 if it is out of sync with the cross-compiler's copy. This means that no cross- C compiling or micro-assembly may be done after the FORTH of computer 100 has altered the dictionary in any way. This could be fixed at a later date by updating the cross-compiler's variables from computer 100 after every BYE command of computer 100.

Cross-compiled code should be kept to a minimum, since it is tricky to write. After a bare minimum kernel is up and running, computer 100 should do all further FORTH compilation. The Micro-assembler. The micro-assembler is a tool to save the programmer from having to set all the bits for microcode by hand. It allows the use of mnemonics for setting the micro-operation fields in a micro-instruction, and, for the most part, automatically handles the micro-instruction addressing scheme.

The micro-assembler is written to be co-resident with the cross-compiler. It uses the same routines for computer 100 and sealed host vocabulary dictionary handling, etc. Currently all microcode must be defined before the board starts altering its dictionary, but this could be changed as discussed previously.

In the terminology used here, a micro-instruction is a 32-bit instruction in microcode, while a micro-operation is formed by one or more microcode fields within a single micro-instruction.

Appendix B gives a quick reference to all the hardware-defined micro-instruction fields supported by the micro-assembler. The usage and operation of each field of the micro-instruction format is covered in detail in Part Two of the User's Manual included as Appendix C. Since the microcode layout is very horizontal, there is a direct relationship between bit settings and control line inputs to various ips on computer 100. As with most horizontally mic oded machines, as many micro-

operations as desired may take place at the same time, although some operations don't do anything useful when used together. Microcode Definitions Format. The micro-assembler has a few keywords to make life easier for the micro-programmer. The word OP-CODE: starts a microcode definition. The input parameter is the page number from 0-OFF hex that the op-code resides in. For example, the word ± is op-code 7. This means that whenever computer **100** interprets a hex **038**xxxxx (where the x's represent don't care bit values), the word ± will be executed in microcode. The character string after OP-CODE: is the name of the op-code that will be added to the cross-compiler and computer **100** dictionaries. It is the programmer's responsibility to ensure that two op-codes are not assigned to the same microcode memory page. The variable CURRENT-OPCODE contains the page currently assigned by OP-CODE:. It may be changed to facilitate multi-page definitions.

The word :: signifies the start of the definition of a micro-instruction. The number before :: must be from 0 to 7, and signifies the offset from 0 to 7 within the current micro-program memory page for that micro-instruction. Micro-instructions may be defined in any order desired. When directly setting the micro-instruction register (MIR) for interactive execution, the word >> may be used without a preceding number instead of the sequence 0 ::.

The word ;; signifies the end of a micro-instruction and stores the micro-instruction into the appropriate location in micro-program memory.

The word ;;END signifies the end of a definition of a FORTH microcoded primitive.

If the FORTH vocabulary is in use, the programmer may single-step microcoded programs. Use the >> word to start a micro-instruction. Instead of using ;;, use ;SET to copy the micro-instruction to the MIR. This allows reading resources of computer **100** to the host **103** with the X@ word or storing resource values with the X! word. Using ;DO instead of ;; will load the instruction into the MIR and cycle the clock once. This is an excellent way of single-stepping microcode. The User's Manual in Appendix C and the Diagnostics of computer **100** given in Appendix A part III provide examples of how to use these features. End/Decode. END and DECODE are the two micro-operations that perform the FORTH NEXT function and perform subroutine calls, subroutine returns, and unconditional branches in parallel with other operations. DECODE is always in the next to last micro-instruction of a microcoded instruction. It causes the interrupt register **126** to be clocked near the falling clock edge, and loads highest 9 bits of the instruction into the instruction latch **125** at the following rising clock edge. Thereafter, instruction fetching and decoding proceeds according to the actions described in Appendix C part II. END is a micro-operation that marks the last instruction in a program and forces a jump to offset 0 of the next instruction's microcoded memory page. Microcode Next Address Generation. The micro-assembler automatically generates an appropriate microcode jump to the next sequential offset within a page. This means that if a 3 is used before the :: word, then the micro-assembler will assume that the next micro-instruction is at offset 4 unless a JMP= micro-instruction is used to tell it otherwise.

The JMP= micro-operation allows forcing non-sequential execution or conditional branching simultaneously with other micro-operations. A JMP=000, JMP=001, ... , JMP=111 command forces an unconditional microcode jump to the offset within the same page specified by the binary operand after JMP=. For example, JMP=101 would force a jump to offset 5 for the next micro-cycle.

A conditional jump allows jumping to one of the two locations depending on the value of one of the 8 condition codes. The unconditional jump described in the preceding paragraph is just a special conditional jump in which the condition picked is a constant that is always set to 0 or 1. The sign bit conditional jump is used below as an example.

A conditional jump sets the lowest bit of the next micro-instruction address to the value of the condition that was valid at the end of the previous microcycle. The syntax is JMP=00S, where "S" can be replaced by any of the conditions: Z, L, C, S, 0, 1. The first two bits are always numeric, indicating the top two binary bits of the jump destination address within the micro-program memory page. The example JMP=10S would jump to offset 4 within the micro-program memory page if the sign bit were 0, and location 5 if it were 1.

Appendix C is the user manual for computer **100**, and describes other information of interest in the operation of the preferred embodiment of the invention.

It will thus be appreciated that the described preferred embodiment achieves the desired features and advantages of the invention. While the invention has been particularly shown and described with reference to the foregoing preferred embodiment, it will be understood by those skilled in the art that other changes in form and detail may be made therein without departing from the spirit and scope of the invention, as defined in the claims.

APPENDIX A

PART I

MICRO-ASSEMBLER, CROSS-COMPILER,

AND MICROCODE PROGRAM LISTING

```
SCREEN #1
 0 INDEX --- CPU/32 X COMPILER & MICRO-ASSEMBLER   PHIL KOOPMAN JR.
 1 LIBRARY VERSION    FILE: XCOMP.4TH          LAST UPDATE: 6/1/87
 2
 3 BETA TEST VERSION  (C) COPYRIGHT 1987
 4 BY PHIL KOOPMAN JR.
 5
 6 LOAD      SOURCE
 7 SCREEN    SCREENS     CONTENTS
 8 =======   ========    ========
 9 2 LOAD    2 - 102     MICRO-CODE ASSEMBLER & CROSS-COMPILER
10    ...    105 - 203   KERNEL MICROCODE DEFINITIONS
11    ...    205 - 254   EXTENDED KERNEL MICROCODE DEFINITIONS
12
13
14
15

SCREEN #2
 0 \ LOAD SCREEN FOR MICRO-ASSEMBLER   & CROSS-COMPILER
 1 HEX  MATH  CR CR ." Loading cross-compiler & micro-assembler" CR
 2  ." (C) Copyright 1987 by Phil Koopman Jr." CR
 3 VARIABLE OPTIMIZE?   \  True value means combine ops with jumps
 4 1 OPTIMIZE? !
 5 100000. DCONSTANT  MEM-SIZE
 6   8000. DCONSTANT SAVE-SIZE
 7 DECIMAL
 8 CR MEM-SIZE  1024. D/   5 D.R ." K bytes memory on system"  CR
 9   4 102  THRU  ( Load cross-compiler )
10 CR CR ." Loading microcode source to CPU/32" CR
11  105 203  THRU  ( Load basic microcode )
12  205 254  THRU  ( Load extended microcode )
13 CR CR ." Saving memory image" CR SAVE-ALL CR CR
14 ." Do a SAVE-FORTH to a DOS file name XCOMP.COM if desired" CR
15 CROSS-COMPILER          FORTH DEFINITIONS

SCREEN #3
 0 \  NO LOADING DONE FROM CPU/32 FROM THIS FILE
 1
 2
 3 CR CR ." NO LOADING DONE FROM CPU/32 FROM THIS FILE" CR CR
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

```
SCREEN #4
 0 \ CASE STATEMENT -- CASE  IFCASE
 1 DECIMAL
 2 : CASE     ( -> )   ( COMPILE )
 3            ( FLAG -> FLAG )    ( EXECUTE )        ;
 4    COMPILE >R   COMPILE R@   0 31  ;
 5    IMMEDIATE
 6
 7 : IFCASE    ( ..ADDRS.. COUNT 31 -> ... BRANADDR COUNT 32 )
 8          ( FLAG -> )    ( EXECUTE )
 9    31 ?PAIRS 32   COMPILE OBRANCH  HERE 0 ,
10    ROT 1+ ROT  ;
11    IMMEDIATE
12
13
14
15

SCREEN #5
 0 \    CASE ---  NEXTCASE  ELSECASE
 1 DECIMAL
 2 : NEXTCASE  ( ... BRANADDR COUNT 32 -> .. ELSEADDR COUNT 31 )
 3       ( -> FLAG ) ( EXECUTE )
 4    32 ?PAIRS   COMPILE BRANCH  0 ,
 5    SWAP  HERE OVER -  SWAP !
 6    HERE 2- SWAP 31  COMPILE R@  ;
 7    IMMEDIATE
 8
 9 : ELSECASE    ( ... COUNT 32 -> ... COUNT 33 )
10    32 ?PAIRS COMPILE BRANCH 0 ,
11    SWAP  HERE OVER - SWAP !
12    HERE 2- SWAP 33 ;
13    IMMEDIATE
14
15

SCREEN #6
 0 \    CASE --- ENDCASE    BETWEEN
 1 DECIMAL
 2 : ENDCASE  ( -> )
 3    DUP 32 =  IF DROP SWAP HERE OVER -  SWAP !
 4    1- 33 THEN   33 ?PAIRS  BEGIN
 5    DUP 0> WHILE SWAP HERE OVER - SWAP !
 6    1- REPEAT  DROP COMPILE R> COMPILE DROP  ;
 7    IMMEDIATE
 8
 9 : BETWEEN    ( N1 N2 N3 -> N2<=N1<=N3? )
10    >R   OVER   > NOT   SWAP
11    R>   > NOT    AND  ;
12
13
14
15

SCREEN #7
 0 \  DEFINE READ0 .. READ7   WRITE0 .. WRITE7
 1 HEX
 2 : READ0   300 P@  ;          : WRITE0   300 P! ;
 3 : READ1   301 P@  ;          : WRITE1   301 P! ;
 4 : READ2   302 P@  ;          : WRITE2   302 P! ;
 5 : READ3   303 P@  ;          : WRITE3   303 P! ;
```

```
 6 : READ4    304 P@  ;          : WRITE4    304 P! ;
 7 : READ5    305 P@  ;          : WRITE5    305 P! ;
 8 : READ6    306 P@  ;          : WRITE6    306 P! ;
 9 : READ7    307 P@  ;          : WRITE7    307 P! ;
10
11 DECIMAL
12
13
14
15

SCREEN #8
 0 \ FORTH BOARD STOP & GO --  STOP  GO   STATUS?  CYCLE
 1 HEX
 2 : STOP    ( -> )
 3     0 WRITE4 ;
 4 : GO    ( -> )
 5     0 WRITE3 ;
 6
 7 : STATUS  ( -> STATUS )     \ Status register contents
 8     READ2 ;   \    OFF AND  ;
 9 : STATUS?   STATUS U.  ;
10
11 : CYCLE   ( -> )       \ Cycle one clock tick
12     0 WRITE2  ;
13 DECIMAL
14
15

SCREEN #9
 0 \ FORTH BOARD    RESET SET-DMA  PCREQ
 1 HEX
 2 : PCREQ   ( N -> )
 3     WRITE7 ;
 4
 5 : RESET-SEQ   ( -> )
 6     0 WRITE6  ;
 7
 8 : SET-DMA ( -> )
 9     0 WRITE5 ;
10
11 DECIMAL
12
13
14
15

SCREEN #10
 0 \ VARIABLES FOR MICROASSEMBLER -- 1
 1 DECIMAL
 2 \ Following variables are all flags that indicate if fields
 3 \       have been used in current micro-instruction assembly
 4 VARIABLE >SOURCE                VARIABLE >DEST
 5 VARIABLE >DP                    VARIABLE >RP
 6 VARIABLE >ALU-MUX               VARIABLE >ALU
 7 VARIABLE >CIN                   VARIABLE >DLO
 8 VARIABLE >CONDITION             VARIABLE >JMP
 9 VARIABLE >INC-MPC               VARIABLE >DECODE
10 VARIABLE >INC-ADC
11
12
```

```
13
14
15

SCREEN #11
 0 \ VARIABLES FOR MICROASSEMBLER   -- 2
 1 DECIMAL
 2 VARIABLE CURRENT-OPCODE  \ Micromemory page number
 3 VARIABLE CURRENT-OFFSET  \ Micromemory offset value
 4 DVARIABLE MICRO-WORD     \ Current micro-assembler building word
 5
 6
 7
 8
 9
10
11
12
13
14
15

SCREEN #12
 0 \ INITIALIZE MICROASSEMBLER FLAGS
 1 DECIMAL
 2 : RESET  ( -> )
 3    0 >SOURCE     !       0 >DEST      !
 4    0 >DP         !       0 >RP        !
 5    0 >ALU-MUX    !       0 >ALU       !
 6    0 >CIN        !       0 >DLO        !
 7    0 >CONDITION  !       0 >JMP       !
 8    0 >INC-MPC    !       0 >DECODE    !
 9    0 >INC-ADC    !
10    0. MICRO-WORD D!   ;
11
12
13
14
15

SCREEN #13
 0 \  SET UP DEFAULTS FOR MICROCODE WORD
 1 HEX   MATH
 2 : FINISH     ( -> )   \ Set required bits for default fields
 3    >DP             @ NOT IF  00000200. MICRO-WORD D+!  THEN
 4        ( Default ALU operation is pass A side function )
 5    >ALU            @ NOT IF  00100000. MICRO-WORD D+!  THEN
 6 \  >ALU  @  NOT IF
 7 \     MICRO-WORD D@  SWAP DROP  20 AND
 8 \     IF   ( CYIN=1 )  00000000.
 9 \     ELSE ( CYIN=0 )  000F0000.    THEN
10 \  MICRO-WORD D+!  THEN
11    >DECODE         @ NOT IF  40000000. MICRO-WORD D+!  THEN ;
12
13 DECIMAL
14
15

SCREEN #14
 0 \ SOURCE DEFINITIONS    - 1
 1 DECIMAL   MATH
```

```
 2 : SRC  ( DVALUE -> )     ( PFA -> )
 3    CREATE  D,
 4    DOES>      >SOURCE @   ABORT" MULTIPLE SOURCE= STATEMENTS"
 5            1 >SOURCE !    D@  MICRO-WORD  D+!  ;
 6
 7    1. SRC SOURCE=DP              2. SRC SOURCE=DS
 8    3. SRC SOURCE=DLO             4. SRC SOURCE=DHI
 9    5. SRC MULTIPLY               6. SRC DIVIDE-SELECT
10    7. SRC SOURCE=FLAGS           8. SRC SOURCE=RP
11    9. SRC SOURCE=RS             10. SRC SOURCE=ADDRESS-COUNTER
12   11. SRC SOURCE=RAM            12. SRC SOURCE=RAM-BYTE
13   13. SRC SOURCE=MPC            14. SRC SOURCE=MRAM
14   15. SRC SOURCE=FPU             0. SRC SOURCE=HOST  ( default )
15 : SOURCE=LATCH ;  ( for readability )

SCREEN #15
 0 \ DEST DEFINITIONS -- 1
 1 DECIMAL  MATH
 2 : DST  ( DVALUE -> )     ( PFA -> )
 3    CREATE  4 DLSLN  D,
 4    DOES>      >DEST @   ABORT" MULTIPLE DEST= STATEMENTS"
 5            1 >DEST !    D@  MICRO-WORD D+!  ;
 6
 7    1. DST DEST=DP                2. DST DEST=DS
 8  ( 3. is unassigned )            4. DST DEST=PAGE
 9    5. DST DEST=ADDRESS-LATCH     6. DST DEST=STATUS
10    7. DST DEST=FLAGS             8. DST DEST=RP
11    9. DST DEST=RS               10. DST DEST=ADDRESS-COUNTER
12   11. DST DEST=RAM              12. DST DEST=RAM-BYTE
13   13. DST DEST=DECODE           14. DST DEST=MRAM
14   15. DST DEST=FPU       : DEST=LATCH ;  ( for readability )
15 : DEST=DHI ;   ( DHI is always clocked -- use for readability )

SCREEN #16
 0 \ DP CONTROL
 1 DECIMAL  MATH
 2 : DPQ  ( DVALUE -> )     ( PFA -> )
 3    CREATE  8 DLSLN  D,
 4    DOES>      >DP @  ABORT" MULTIPLE DP STATEMENTS"
 5            1 >DP !   D@    MICRO-WORD D+! ;
 6
 7   0. DPQ DEC[DP]
 8   1. DPQ INC[DP]
 9
10
11
12
13
14
15

SCREEN #17
 0 \ RP CONTROL
 1 DECIMAL  MATH
 2 : RP  ( DVALUE -> )     ( PFA -> )
 3    CREATE  10 DLSLN  D,
 4    DOES>      >RP @  ABORT" MULTIPLE RP STATEMENTS"
 5            1 >RP !   D@    MICRO-WORD D+! ;
 6
 7   3. RP DEC[RP]
 8   2. RP INC[RP]
```

```
 9
10
11
12
13
14
15

SCREEN #18
 0 \ ALU MUX SELECTION DEFINITIONS
 1 DECIMAL
 2 : ALUMUX  ( DVALUE -> )     ( PFA -> )
 3    CREATE 12 DLSLN  D,
 4    DOES>      >ALU-MUX @   ABORT" MULTIPLE ALU MUX STATEMENTS"
 5             1 >ALU-MUX !    D@  MICRO-WORD D+!  ;
 6
 7   1. ALUMUX SR[ALU]
 8   2. ALUMUX SL[ALU]
 9   3. ALUMUX ROLL[ALU]
10
11
12
13
14
15

SCREEN #19
 0 \ ALU FUNCTION CONTROL  -- LOGICAL
 1 HEX
 2 : ALUL ( DVALUE -> )     ( PFA -> )
 3 \ Automatically compensates for FUNC & CIN inverters on board
 4 \ Function codes used below reflect '181 docn for active hi data
 5    CREATE  00F. DXOR    10 DLSLN  D,
 6    DOES>      >ALU @  ABORT" MULTIPLE ALU STATEMENTS"
 7             1 >ALU !   D@    MICRO-WORD D+! ;
 8
 9
10
11 DECIMAL
12
13
14
15

SCREEN #20
 0 \ ALU DEFINITIONS  -- 1
 1 HEX
 2 \ Logical definitions
 3  10. ALUL ALU=notA
 4  11. ALUL ALU=AnorB          19. ALUL ALU=AxnorB
 5                              1A. ALUL ALU=B
 6  13. ALUL ALU=0              1B. ALUL ALU=AandB
 7  14. ALUL ALU=AnandB         1C. ALUL ALU=-1
 8  15. ALUL ALU=notB
 9  16. ALUL ALU=AxorB          1E. ALUL ALU=AorB
10                              1F. ALUL ALU=A
11 DECIMAL
12
13
14
15
```

```
SCREEN #21
 0 \ ALU FUNCTION CONTROL  -- ARITHMETIC
 1 HEX
 2 : ALUM  ( DVALUE -> )     ( PFA -> )
 3 \ Automatically compensates for FUNC & CIN inverters on board
 4 \ Function codes used below reflect '181 docn for active hi data
 5    CREATE  02F. DXOR    10 DLSLN  D,
 6    DOES>      >ALU @  ABORT" MULTIPLE ALU STATEMENTS"
 7               >CIN @  ABORT" MULTIPLE CARRY-IN STATEMENTS"
 8               1 >CIN !   1 >ALU !   D@    MICRO-WORD D+! ;
 9
10
11
12 DECIMAL
13
14
15

SCREEN #22
 0 \ ALU DEFINITIONS - 2
 1 HEX
 2 \ Arithmetic definitions
 3  20. ALUM ALU=A+0     \ Passes A through, valid CY out
 4  26. ALUM ALU=A-B-1
 5  29. ALUM ALU=A+B
 6  2C. ALUM ALU=A+A
 7  2F. ALUM ALU=A-1
 8
 9  00. ALUM ALU=A+1
10  02. ALUM ALU=AornotB+1
11  06. ALUM ALU=A-B
12  09. ALUM ALU=A+B+1
13  0C. ALUM ALU=A+A+1
14 DECIMAL
15

SCREEN #23
 0 \ CARRY IN CONTROL
 1 DECIMAL
 2 : CIN ( DVALUE -> )     ( PFA -> )
 3    CREATE   21 DLSLN  D,
 4    DOES>      >CIN @  ABORT" MULTIPLE CARRY-IN STATEMENTS"
 5             1 >CIN !   D@    MICRO-WORD D+! ;
 6
 7    0. CIN CIN=0
 8    1. CIN CIN=1
 9
10
11
12
13
14
15

SCREEN #24
 0 \ DLO CONTROL DEFINITIONS
 1 DECIMAL
 2 : DLO ( DVALUE -> )     ( PFA -> )
 3    CREATE  22 DLSLN  D,
 4    DOES>      >DLO @   ABORT" MULTIPLE DLO STATEMENTS"
 5             1 >DLO !    D@  MICRO-WORD D+!  ;
```

```
 6
 7    1. DLO SR[DLO]
 8    2. DLO SL[DLO]
 9    3. DLO DEST=DLO
10
11
12
13
14
15

SCREEN #25
 0 \ DIVIDE OPERATION MACRO
 1 HEX
 2 : DIVIDE ( -> )
 3     DIVIDE-SELECT
 4     >CIN      @  ABORT" MULTIPLE CARRY-IN STATEMENTS"
 5     >ALU      @  ABORT" MULTIPLE ALU MUX STATEMENTS"
 6     1 >ALU ! 1 >CIN !
 7        002F0000.     MICRO-WORD D+! ;
 8
 9 DECIMAL
10
11
12
13
14
15

SCREEN #26
 0 \ NEXT ADDRESS CONSTANT CONTROL
 1 DECIMAL
 2 : JMP  ( DVALUE -> )     ( PFA -> )
 3     CREATE  24 DLSLN  D,
 4     DOES>     >JMP @    ABORT" MULTIPLE JMP STATEMENTS"
 5             1 >JMP !    D@     MICRO-WORD D+! ;
 6 HEX
 7 \ Constant address jumps
 8     00. JMP JMP=000              07. JMP JMP=001
 9     08. JMP JMP=010              0F. JMP JMP=011
10     10. JMP JMP=100              17. JMP JMP=101
11     18. JMP JMP=110              1F. JMP JMP=111
12
13 DECIMAL
14
15

SCREEN #27
 0 \ JMP STATEMENTS -- 2
 1 HEX
 2 \ Branch on not carry-out
 3     01. JMP JMP=00C              11. JMP JMP=10C
 4     09. JMP JMP=01C              19. JMP JMP=11C
 5
 6 \ Branch on ALU output not equal to 0
 7     02. JMP JMP=00Z              12. JMP JMP=10Z
 8     0A. JMP JMP=01Z              1A. JMP JMP=11Z
 9
10 \ Branch on sign bit = 1    (ALU output bit 31)
11     03. JMP JMP=00S              13. JMP JMP=10S
12     0B. JMP JMP=01S              1B. JMP JMP=11S
```

```
13
14 DECIMAL
15

SCREEN #28
 0 \ JMP STATEMENTS -- 3
 1 HEX
 2 \ Branch on lowest bit of DLO (shift-out bit)
 3    04. JMP JMP=00L            14. JMP JMP=10L
 4    0C. JMP JMP=01L            1C. JMP JMP=11L
 5 DECIMAL
 6 EXIT
 7 \ Branch on ?????????????????????????????? CONDITION # 5
 8    05. JMP JMP=00?            15. JMP JMP=00?
 9    0D. JMP JMP=01?            1D. JMP JMP=01?
10
11 \ Branch on ???????????????????????????????? CONDITION # 6
12    06. JMP JMP=00??           16. JMP JMP=00??
13    0E. JMP JMP=01??           1E. JMP JMP=01??
14
15 DECIMAL

SCREEN #29
 0 \       MPC CONTROL
 1 DECIMAL     MATH
 2 : MPC  ( DVALUE -> )    ( PFA -> )
 3    CREATE    29 DLSLN  D,
 4     DOES>    >INC-MPC @    ABORT" MULTIPLE MPC STATEMENTS"
 5           1 >INC-MPC !     D@  MICRO-WORD D+! ;
 6
 7   1. MPC  INC[MPC]
 8
 9
10
11
12
13
14
15

SCREEN #30
 0 \  ADDRESS COUNTER  CONTROL
 1 DECIMAL     MATH
 2 : INADC ( DVALUE -> )    ( PFA -> )
 3    CREATE    31 DLSLN  D,
 4     DOES>    >INC-ADC @    ABORT" MULTIPLE INC[ADC] STATEMENTS"
 5           1 >INC-ADC !     D@  MICRO-WORD D+! ;
 6
 7   1. INADC INC[ADC]
 8
 9
10
11
12
13
14
15

SCREEN #31
 0 \  DECODE MACRO    &  END    RO
 1 HEX     MATH
 2 : DECODE ( -> )
```

```
 3    >DECODE @   ABORT" MULTIPLE DECODE STATEMENTS"
 4    1 >DECODE !    00000000.  MICRO-WORD D+!  ;
 5
 6 : END   ( -> )
 7    JMP=000 ;
 8
 9
10 DECIMAL
11
12
13
14
15
```

```
SCREEN #32
 0 \ FORTH BOARD I/O PRIMITIVES --  X!  X@
 1 HEX
 2 CODE X!  ( D -> )   \ WRITE WORD TO BOARD
 3    DX , # 306 MOV   AL , DX  OUT ( Reset sequencer )
 4    DX , # 300 MOV     BX POP    AX POP
 5                  AL , DX OUT      AL , AH MOV   AL , DX OUT
 6    AL , BL MOV   AL , DX OUT      AL , BH MOV   AL , DX OUT
 7    NEXT  JMP    END-CODE
 8
 9 CODE X@  ( D -> )    \ READ WORD FROM BOARD
10    DX , # 306 MOV   AL , DX OUT  ( Reset sequencer )
11    DX , # 300 MOV                 AL , DX IN    BL , AL MOV
12    AL , DX IN   BH , AL MOV       AL , DX IN    CL , AL MOV
13    AL , DX IN   CH , AL MOV        BX PUSH   CX PUSH
14    NEXT JMP     END-CODE
15 DECIMAL
```

```
SCREEN #33
 0 \ FORTH BOARD I/O PRIMITIVES --  MIR!  MIR@
 1 HEX
 2 CODE MIR!  ( D -> )   \ WRITE WORD TO BOARD
 3    DX , # 306 MOV   AL , DX  OUT ( Reset sequencer )
 4    DX , # 301 MOV     BX POP    AX POP
 5                  AL , DX OUT      AL , AH MOV   AL , DX OUT
 6    AL , BL MOV   AL , DX OUT      AL , BH MOV   AL , DX OUT
 7    NEXT  JMP    END-CODE
 8
 9 CODE MIR@  ( D -> )    \ READ WORD FROM BOARD
10    DX , # 306 MOV   AL , DX OUT  ( Reset sequencer )
11    DX , # 301 MOV                 AL , DX IN    BL , AL MOV
12    AL , DX IN   BH , AL MOV       AL , DX IN    CL , AL MOV
13    AL , DX IN   CH , AL MOV        BX PUSH   CX PUSH
14    NEXT JMP     END-CODE
15 DECIMAL
```

```
SCREEN #34
 0 \  SET UP MICROCODE WORD --   ;SET  ;DO
 1 DECIMAL
 2 : ;SET   ( -> )    \ Perform MIR!, but don't cycle clock
 3    FINISH  MICRO-WORD D@  MIR!  ;
 4
 5 : ;DO   ( -> )     \ Load and execute single instruction
 6    ;SET  CYCLE  ;
 7
 8 : ::  RESET   'RRENT-OFFSET ! ;
 9
```

```
10 : >>  RESET  -2 CURRENT-OFFSET ! ;
11
12
13
14
15

SCREEN #35
 0 \ AID TO SPEED EXECUTION   -- MIR-SETUP
 1 DECIMAL
 2 : MIR-SETUP  ( -> )   \ Use after [ ] around microcode word
 3    MICRO-WORD D@
 4    [COMPILE] DLITERAL
 5    COMPILE MIR!  ;
 6    IMMEDIATE
 7
 8
 9
10
11
12
13
14
15

SCREEN #36
 0 \ AUTOMATIC NEXT ADDRESS GENERATION
 1 DECIMAL
 2 : <AUTO-ADDR>   ( -1..7 -> )
 3    CASE -1 =  IFCASE   JMP=000   NEXTCASE
 4          0 =  IFCASE   JMP=001   NEXTCASE
 5          1 =  IFCASE   JMP=010   NEXTCASE
 6          2 =  IFCASE   JMP=011   NEXTCASE
 7          3 =  IFCASE   JMP=100   NEXTCASE
 8          4 =  IFCASE   JMP=101   NEXTCASE
 9          5 =  IFCASE   JMP=110   NEXTCASE
10          6 =  IFCASE   JMP=111   NEXTCASE
11         -2 =  IFCASE   ABORT" MUST USE :: WITH ;;" NEXTCASE
12          7 =  IFCASE   ABORT" NO JMP= ON PAGE CROSSING"
13    ELSECASE   1 ABORT" ERROR IN AUTO-ADDR"   ENDCASE    ;
14
15

SCREEN #37
 0 \ MICRO-RAM !  AND @
 1 DECIMAL     MATH
 2 : MRAM!   ( DMICRO-WORD  ADDR -> )
 3    [ >> DEST=DECODE ;SET  ] MIR-SETUP    DUP  0  20 DLSLN   X!
 4    [ >> DECODE ;SET    ] MIR-SETUP    CYCLE
 5    [ >>        ;SET    ] MIR-SETUP    CYCLE   CYCLE
 6    [ >> DEST=MRAM ;SET MICRO-WORD D@ ] DLITERAL  MICRO-WORD D!
 7    0 >JMP !  7 AND 1-  <AUTO-ADDR>   MICRO-WORD D@ MIR!  X! ;
 8 : MRAM@   ( ADDR -> DMICRO-WORD  )
 9    [ >> DEST=DECODE ;SET  ] MIR-SETUP    DUP  0  20 DLSLN   X!
10    [ >> DECODE ;SET    ] MIR-SETUP    CYCLE
11    [ >>        ;SET    ] MIR-SETUP    CYCLE   CYCLE
12    [ >> SOURCE=MRAM ;SET MICRO-WORD D@ ] DLITERAL  MICRO-WORD D!
13    0 >JMP !  7 AND 1-  <AUTO-ADDR>   MICRO-WORD D@ MIR!  X@ ;
14
15
```

```
SCREEN #38
 0 \ PROGRAM RAM !  AND @
 1 DECIMAL      MATH
 2 : RAM!    ( DWORD  DADDR -> )
 3     [ >> DEST=ADDRESS-COUNTER ;SET ] MIR-SETUP  X!
 4     [ >>              DEST=RAM  ;SET    ] MIR-SETUP  X!    ;
 5
 6 : RAM@    ( DADDR -> DWORD  )
 7     [ >> DEST=ADDRESS-COUNTER ;SET ] MIR-SETUP  X!
 8     [ >> SOURCE=RAM  ;SET         ] MIR-SETUP  X@ ;
 9
10
11
12
13
14
15

SCREEN #39
 0 \ PROGRAM RAM C!  AND C@
 1 DECIMAL      MATH
 2 : RAMC!    ( B  DADDR -> )
 3     [ >> DEST=ADDRESS-LATCH ;SET ] MIR-SETUP  X!
 4     [ >>              DEST=RAM-BYTE  ;SET    ] MIR-SETUP 0 X!    ;
 5
 6 : RAMC@    ( DADDR -> B  )
 7     [ >> DEST=ADDRESS-LATCH ;SET ] MIR-SETUP  X!
 8     [ >> SOURCE=RAM-BYTE  ;SET         ] MIR-SETUP  X@  DROP ;
 9
10
11
12
13
14
15

SCREEN #40
 0 \  STORE MICROCODE WORD --   ;;   INSTRUCTION  END
 1 DECIMAL
 2 : ;;  ( -> )
 3   >JMP @ NOT   \ Default next address is current word + 1
 4   IF    CURRENT-OFFSET @   <AUTO-ADDR>  THEN
 5   FINISH   MICRO-WORD D@
 6   CURRENT-OPCODE @  8 *   CURRENT-OFFSET @  +  MRAM!  ;
 7 : INSTRUCTION  ( N -> )
 8      STOP  CURRENT-OPCODE !   -1 CURRENT-OFFSET !  ;
 9
10 : RESET-BOARD  ( -> )   \ Sets page=0 &
11     \ Does 2 dummy cycles to flush out any possible decodes
12     STOP RESET-SEQ     [ >> ;SET ] MIR-SETUP  CYCLE CYCLE
13     [ >> DEST=DECODE ;SET ] MIR-SETUP 0. X!
14     [ >> DEST=PAGE ;SET ] MIR-SETUP 0. X!
15     [ >> DEST=FLAGS ;SET ] MIR-SETUP -1. X! ( Mask interrupts ) ;

SCREEN #41
 0 EXIT  \ INIT DMA REQUEST -- DMA CHANNEL 3
 1 HEX  MATH
 2 ( Note:  write means write from PC to the FORTH system )
 3 : SETUP-DMA  ( WRITE-FLAG HI-4-SEG ADDRESS COUNT -> )
 4     RESET-SEQ     >R         07  0A  !   ( disable ch 3 )
 5     SET-DMA ( set DMA mode )
```

```
 6    0 0C P!  ( clear byte ptr flip-flop )  \ BIOS places NOP here
 7    ROT IF ( write ) 4B   ELSE  ( read ) 47  THEN  0B P!
 8          DUP    0FF AND  06 P!  ( low address )
 9       BYTESWAP  0FF AND  06 P!  ( high address )
10         82 P!  ( set up top 4 bits of address )
11    R>    DUP    0FF AND  07 P!  ( high count )
12       BYTESWAP  0FF AND  07 P!  ( low count )
13    03   0A P!  ( enable channel 3 ) ;
14 DECIMAL
15


SCREEN #42
 0 \ INIT DMA REQUEST -- DMA CHANNEL 3  -- ASSEMBLER VERSION --1
 1 HEX
 2 ( Note:  write means write from PC to the FORTH system )
 3 CODE  SETUP-DMA  ( WRITE-FLAG HI-4-SEG ADDRESS COUNT -> )
 4    CLI
 5    AL , # 07 MOV   DX , # 0A MOV   DX , AL OUT  \ Disable ch 3
 6    DX , # 306 MOV   DX , AL OUT   \  Reset CPU/32 sequencer
 7    DX , # 305 MOV   DX , AL OUT   \  Set CPU/32 DMA mode
 8    AL , AL XOR
 9    DX , # 0C MOV   DX , AL OUT   \  Clear DMA contrllr byte
10    CX POP  ( Count )  DI POP  ( low addr )  BX POP ( Hi addr )
11    AX POP  ( non-0 is write flag )   AX , AX OR
12    <>0? IF ( write ) AL , # 4B MOV  ELSE  ( read ) AL , # 47 MOV
13       THEN   DX , # 0B MOV  DX , AL OUT
14
15 DECIMAL

SCREEN #43
 0 \ INIT DMA REQUEST -- DMA CHANNEL 3  -- ASSEMBLER VERSION --2
 1 HEX
 2    DX , # 6 MOV    \ Set low address
 3       AX , DI MOV   DX , AL OUT      <>0? IF ELSE THEN
 4       AL , AH MOV   DX , AL OUT
 5    <>0?  IF  ELSE  THEN    AL , BL MOV  \  Set DMA page register
 6    DX , # 82 MOV   DX , AL OUT       <>0? IF ELSE THEN
 7    DX , # 7 MOV    \ Set count
 8       AX , CX MOV  DX , AL OUT      <>0? IF ELSE THEN
 9       AL , AH MOV   DX , AL OUT
10    STI                              <>0? IF ELSE THEN
11    AL , # 03 MOV   DX , # 0A MOV   DX , AL OUT  \ Enable ch 3
12    NEXT JMP
13 END-CODE
14 DECIMAL
15

SCREEN #44
 0 \ FORTH BOARD I/O PRIMITIVES --  W->BOARD
 1 HEX
 2 CODE DMA-WAIT      ( -> )
 3    BEGIN    AL , 8 IN   AL , # 8 AND  <>0? UNTIL  NEXT JMP
 4 END-CODE
 5 HEX       MATH
 6 : W->BOARD    ( PCADDR DBOARD-ADDR WORDCOUNT -> )
 7    >R  RESET-SEQ
 8    [ >> DEST=ADDRESS-COUNTER ;SET ]  MIR-SETUP  X!
 9    [ >> DEST=RAM  INC[ADC] ;SET ]  MIR-SETUP
10    >R   1  THIS-SEG  10 U*  R> 0 D+    OVER R@  0 ADC
11       ABORT" DMA CROSSES PAGE REG BOUNDARY"  DROP
```

```
12    SWAP  R>  2* 2*   SETUP-DMA   DMA-WAIT   ;
13 \  BEGIN  08 P@  8 AND
14 \     ( ?TERMINAL  ABORT" ..DMA-1.." )  UNTIL   ;
15 DECIMAL
```

```
SCREEN #45
 0 EXIT 103 LOAD \  W->BOARD  ** HIGH  LEVEL **
 1 DECIMAL  MATH  \ Transfer words from PC to board
 2 : W->BOARD   ( PCADDR DBOARD-ADDR WORDCOUNT -> )
 3    0 DO
 4       ROT   DUP XD@   D>R   4 +  ROT ROT       ;
 5       DR>  DOVER   RAM!   4. D+
 6    LOOP   DDROP DROP ;
 7
 8
 9
10
11
12
13
14
15
```

```
SCREEN #46
 0 \ FORTH BOARD I/O PRIMITIVES --  BOARD->W
 1 HEX      MATH
 2 : BOARD->W   ( DBOARD-ADDR PCADDR WORDCOUNT -> )
 3    >R   >R    RESET-SEQ
 4    [ >> DEST=ADDRESS-COUNTER ;SET ]   MIR-SETUP  X!
 5    [ >> SOURCE=RAM INC[ADC] ;SET ]  MIR-SETUP
 6    0  THIS-SEG  10 U*  R> 0 D+    OVER R@  0 ADC
 7       ABORT" DMA CROSSES PAGE REG BOUNDARY"  DROP
 8    SWAP  R>  2* 2*   1-    SETUP-DMA      DMA-WAIT ;
 9 \  BEGIN  08 P@  8 AND
10 \     ( ?TERMINAL  ABORT" ..DMA-2.." )  UNTIL ;
11 DECIMAL
12
13
14
15
```

```
SCREEN #47
 0 EXIT \ FORTH BOARD I/O PRIMITIVES --  BOARD->W  ** HIGH LEVEL *
 1 DECIMAL  MATH  \ Transfer words from board to PC
 2 : BOARD->W   ( DBOARD-ADDR PCADDR WORDCOUNT -> )
 3    0 DO
 4       ROT ROT   DDUP RAM@   D>R   4. D+  ROT
 5       DR>  3 PICK  XD!   4  +
 6    LOOP   DDROP DROP ;
 7
 8
 9
10
11
12
13
14
15
```

```
SCREEN #48
 0 EXIT  \  LIBFORTH FILE NAME FOR MICROCODE SAVES
 1 DECIMAL
 2 CREATE SAVE-FILE-NAME   30 ALLOT
 3 : SET-SAVE-FILE
 4    SAVE-FILE-NAME 30 0 FILL
 5    BL WORD  SAVE-FILE-NAME OVER C@ 1+  CMOVE ;
 6
 7 SET-SAVE-FILE CPU32.DAT
 8
 9 : OPEN-SAVE-FILE  ( -> FILE# OFFSET#  )
10    [COMPILE] MATH    CURRENT-FILE @  OFFSET @
11    FILE0  ENSURE-CLOSED
12    SAVE-FILE-NAME   FILE-NAME  OVER C@ 1+ CMOVE  OPEN ;
13
14 : CLOSE-SAVE-FILE  ( FILE# OFFSET# -> )
15    CLOSE   OFFSET !   CURRENT-FILE !  ;

SCREEN #49
 0 \  SAVE MICROCODE IMAGE ON SCREENS 310-325  -- SAVE-MICROCODE
 1 DECIMAL
 2 : SAVE-MICRO-SCREEN  ( START-uADDR SCREEN# -> )
 3    DUP  BUFFER  SWAP OFFSET @ + OVER 2- !  OVER  256 +   ROT
 4    DO  I   MRAM@
 5            3 PICK D!    4 +   LOOP
 6    DROP   UPDATE  SAVE-BUFFERS ;
 7
 8 : SAVE-MICROCODE   ( -> )
 9    ( OPEN-SAVE-FILE  )
10    RESET-BOARD   0
11    326 310 DO   2 EMIT         DUP  I SAVE-MICRO-SCREEN
12                              256 +    LOOP
13    DROP    ( CLOSE-SAVE-FILE ) ;
14
15

SCREEN #50
 0 EXIT \  LOAD MICROCODE IMAGE
 1 DECIMAL
 2 : LOAD-MICRO-SCREEN  ( START-uADDR SCREEN# -> )
 3    BLOCK     OVER  256 +   ROT
 4    DO  DUP D@  I  MRAM!   4 +   LOOP
 5    DROP    ;
 6
 7 : LOAD-MICROCODE   ( -> )
 8    ( OPEN-SAVE-FILE  )
 9    RESET-BOARD     0
10    326 310 DO   2 EMIT       DUP  I LOAD-MICRO-SCREEN
11                              256 +    LOOP
12    DROP    ( CLOSE-SAVE-FILE ) ;
13
14
15

SCREEN #51
 0 \ MICRO-RAM !  AND @
 1 HEX MATH \ Store 8 microwords.  ADDR must be offset 0 into page
 2 : MRAM!PAGE   ( ..LO.. 8xDMICRO-WORD ..HI..  ADDR -> )
 3    [ >> DEST=DECODE ;SET  ] MIR-SETUP    0  14 DLSLN   X!
 4    [ >> DECODE ;SET   ] MIR-SETUP    CYCLE
 5    [ >>        ;SET   ] MIR-SETUP    CYCLE   CYCLE
```

```
 6     [ >> DEST=MRAM ;SET MICRO-WORD D@ ] DLITERAL     D>R
 7     DR@ 1F00 OR  MIR!  X!       \ Offset 7
 8     DR@ 1800 OR  MIR!  X!       \ Offset 6
 9     DR@ 1700 OR  MIR!  X!       \ Offset 5
10     DR@ 1000 OR  MIR!  X!       \ Offset 4
11     DR@ 0F00 OR  MIR!  X!       \ Offset 3
12     DR@ 0800 OR  MIR!  X!       \ Offset 2
13     DR@ 0700 OR  MIR!  X!       \ Offset 1
14   ( Offset 0 )   DR>   MIR!  X!            ;
15 DECIMAL

SCREEN #52
 0 \  LOAD MICROCODE IMAGE ON SCREENS 310-325  -- LOAD-MICROCODE
 1 DECIMAL
 2 VARIABLE MICRO-ADDR
 3 : LOAD-MICRO-SCREEN  ( START-uADDR SCREEN# -> )
 4     BLOCK   MICRO-ADDR !     DUP  256 +  SWAP
 5     DO      8 0 DO   MICRO-ADDR @ D@  4 MICRO-ADDR +!  LOOP
 6             I MRAM!PAGE
 7     8 +LOOP   ;
 8
 9 : LOAD-MICROCODE   ( -> )
10     ( OPEN-SAVE-FILE  )
11     RESET-BOARD      0
12     326 310 DO   2 EMIT      DUP  I LOAD-MICRO-SCREEN
13                               256 +    LOOP
14     DROP     ( CLOSE-SAVE-FILE ) ;
15

SCREEN #53
 0 \  SAVE PGM RAM IMAGE ON SCREENS 327 - ... -- SAVE-BOARD-FORTH
 1 DECIMAL  MATH
 2 : SAVE-BOARD-SCREEN  ( DBOARD-ADDR SCREEN# -> )
 3     DUP  BUFFER  SWAP OFFSET @ + OVER 2- !
 4     256   BOARD->W  UPDATE SAVE-BUFFERS ;
 5
 6 : SAVE-BOARD-FORTH   ( -> )
 7     ( OPEN-SAVE-FILE  )
 8     RESET-BOARD  0.
 9     SAVE-SIZE 1024.  D/    DROP  327 +   327
10     DO  1 EMIT  DDUP  I SAVE-BOARD-SCREEN
11                              1024. D+   LOOP
12     DDROP     ( CLOSE-SAVE-FILE ) ;
13
14
15

SCREEN #54
 0 \  LOAD PGM RAM IMAGE ON SCREENS 327 - ... -- LOAD-BOARD-FORTH
 1 DECIMAL  MATH
 2 : LOAD-BOARD-SCREEN  ( DBOARD-ADDR SCREEN# -> )
 3     BLOCK ROT ROT  256    W->BOARD  ;
 4
 5 : LOAD-BOARD-FORTH   ( -> )
 6     ( OPEN-SAVE-FILE  )
 7     RESET-BOARD  0.
 8     SAVE-SIZE 1024.  D/    DROP  327 +   327
 9     DO  1 EMIT   DDUP  I LOAD-BOARD-SCREEN
10                               1024. D+   LOOP
11     DDROP     ( CLOSE-SAVE-FILE ) ;
```

```
12
13
14
15

SCREEN #55
 0 \ LOAD-ALL & SAVE-ALL
 1 DECIMAL
 2 : LOAD-ALL  LOAD-MICROCODE LOAD-BOARD-FORTH  ;
 3
 4 : SAVE-ALL  SAVE-MICROCODE SAVE-BOARD-FORTH  ;
 5
 6
 7
 8
 9
10
11
12
13
14
15

SCREEN #56
 0 \  DATA STACK TO BOARD
 1 HEX
 2 : DS->BOARD  ( ...DSTACKIN... -> )   \ Assume all 32-bit 0's
 3    STOP
 4    DEPTH 2/  1+   NEGATE  OFFF AND   >R
 5    DEPTH 0=  IF  0 0  THEN
 6    [ >> DEST=DP ;SET ] MIR-SETUP   R@  0   X!
 7    OFFF  R@  DO
 8    [ >> DEST=DS          ;SET ] MIR-SETUP     X!
 9    [ >>          INC[DP] ;SET ] MIR-SETUP    CYCLE   LOOP
10    [ >> DEST=DP  ;SET ] MIR-SETUP    R>  0  X!
11    [ >> SOURCE=DS  ALU=B  DEST=DHI ;SET ] MIR-SETUP  CYCLE
12    [ >> INC[DP] ;SET ] MIR-SETUP  CYCLE   ;
13
14 DECIMAL
15


SCREEN #57
 0 \  DATA STACK FROM BOARD
 1 HEX
 2 : BOARD->DS  ( -> ...DSTACKOUT... )
 3    STOP
 4    SP!    [ >> SOURCE=DP  ;SET ] MIR-SETUP   X@, DROP OFFF AND
 5    DUP  000  =  IF    DROP
 6    ELSE  1-
 7       DUP OFFE = NOT
 8       IF  [ >>  DEST=DP  ;SET ] MIR-SETUP    OFFE. X!
 9          OFFE  SWAP  DO
10             [ >> SOURCE=DS  ;SET  ] MIR-SETUP  X@  ?STACK
11             [ >>            DEC[DP]  ;SET  ] MIR-SETUP CYCLE LOOP
12       ELSE DROP THEN
13       [ >> SOURCE=DHI ;SET  ]    MIR-SETUP    X@
14    THEN  ;
15   DECIMAL
```

```
SCREEN #58
 0 \ BOARD VOCABULARY
 1 DECIMAL   FORTH  DEFINITIONS
 2 : B-RES    [ 'INTERPRET @ ] LITERAL   'INTERPRET !  ;
 3
 4 VARIABLE 'BINTER    'INTERPRET @  'BINTER !
 5
 6 VOCABULARY BOARD-VOC  IMMEDIATE
 7 : {     ( -> )        \  Enter board vocabulary/execution mode
 8    [COMPILE] BOARD-VOC  DEFINITIONS
 9    'BINTER @  'INTERPRET !
10    ( Get rid of return stack junk )
11    R> R> DDROP  'INTERPRET @ EXECUTE ;
12
13 FORTH DEFINITIONS
14
15

SCREEN #59
 0 \ NULL FOR BOARD VOCABULARY
 1 HEX   BOARD-VOC  DEFINITIONS
 2 : X   FORTH    BLK @              \ Redefine  null
 3    IF  STATE @  ?STREAM  THEN       R> DROP   ;
 4
 5 FORTH DEFINITIONS
 6
 7
 8 DECIMAL
 9
10
11
12
13
14
15

SCREEN #60
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15

SCREEN #61
 0 \ SERVICE ARRAY
 1 DECIMAL  FORTH DEFINITIONS
 2 CREATE SERVICE-ARRAY  512 ALLOT
 3
 4 : SERVICE-ENTRY  ( CFA INDEX -> )
 5    DUP 255 >   OVER  0<  OR
```

```
 6        ABORT" INVALID SERVICE-ENTRY INDEX"
 7     2*  SERVICE-ARRAY + ! ;
 8
 9
10
11
12
13
14
15

SCREEN #62
 0 \ SERVICE ARRAY  INITIALIZATION
 1 DECIMAL
 2 : BAD-SERVICE-CALL  ( -> )
 3   ' 1 ABORT" CALL TO BAD SERVICE ROUTINE" ;
 4
 5 : XXX        \ SCRATCH
 6    256 0 DO
 7        ' BAD-SERVICE-CALL CFA  I  SERVICE-ENTRY LOOP ;
 8
 9 XXX   FORGET XXX
10
11 : BAD-OPCODE   ( Board:  -> )
12   CR CR ." ILLEGAL OPCODE -- RETURN TO PC FORTH SYSTEM" ABORT ;
13
14 ' BAD-OPCODE CFA  254 SERVICE-ENTRY
15

SCREEN #63
 0 \ SERVICE CALL PRIMITIVES
 1 HEX
 2 : RESTART  ( -> )     \ Restart FORTH board for concurrent usage
 3    [ >> JMP=100  ;SET  ]       MIR-SETUP GO ;
 4
 5 : SERV-PAGE   ( Board:  -> )
 6    RESTART  PAGE ;
 7 ' SERV-PAGE CFA  1 SERVICE-ENTRY
 8
 9 : SERV-KEY     ( Board:  0 -> CHAR )
10    [ >> ALU=B  DEST=DHI ;SET  ] MIR-SETUP
11    KEY 0  X!      RESTART  ;
12 ' SERV-KEY CFA  2 SERVICE-ENTRY
13
14 DECIMAL
15

SCREEN #64
 0 \ SERVICE CALL PRIMITIVES   - 2
 1 DECIMAL
 2 : SV-2ND  ( -> D2ND.STACK.ITEM )
 3    [ >> SOURCE=DS ;SET ]  MIR-SETUP   X@  ;
 4
 5 : SERV-READ   ( Board: AD BLK# -> AD BLK# )
 6    X@  DROP DUP  >R
 7    BLOCK  SV-2ND   256 W->BOARD     RESTART
 8 \  Perform read of next block to have it available in RAM
 9    R> 1+ BLOCK DROP  ;
10 ' SER  READ CFA  3 SERVICE-ENTRY
11 : SE   WRITE  ( Board: AD BLK# -> AD BLK# )
```

```
12    X@  DROP
13    SV-2ND  ROT  OFFSET @ +  BUFFER  256 BOARD->W
14    RESTART  UPDATE  SAVE-BUFFERS  ;
15 ' SERV-WRITE CFA  4 SERVICE-ENTRY
```

```
SCREEN #65
 0 \ SERVICE CALL PRIMITIVES   - 3
 1 HEX
 2 : SERV-TERM   ( Board:  0 -> FLAG )
 3    [ >> ALU=B DEST=DHI ;SET  ] MIR-SETUP
 4    ?TERMINAL
 5    IF  -1.  ELSE 0.  THEN  X!    RESTART ;
 6 ' SERV-TERM CFA  5 SERVICE-ENTRY
 7
 8 DECIMAL
 9
10
11
12
13
14
15
```

```
SCREEN #66
 0 \ SERVICE CALL PRIMITIVES    - 4
 1 HEX
 2 : SERV-CR     ( Board:  EPRINT -> EPRINT )
 3    EPRINT @      X@  OR  EPRINT !
 4    RESTART  CR     EPRINT ! ;
 5 ' SERV-CR   CFA  7 SERVICE-ENTRY
 6
 7 DECIMAL
 8
 9
10
11
12
13
14
15
```

```
SCREEN #67
 0 \ SERVICE CALL PRIMITIVES    - 4
 1 HEX
 2 : SERV-EMIT   ( Board:  CHAR -> CHAR )
 3    X@ RESTART    DROP  0 EPRINT !  EMIT ;
 4 ' SERV-EMIT CFA  8 SERVICE-ENTRY
 5
 6 : SERV-EMIT-PRINT   ( Board:  CHAR -> CHAR )
 7    X@ RESTART    DROP  1 EPRINT !  EMIT ;
 8 ' SERV-EMIT-PRINT CFA  9 SERVICE-ENTRY
 9
10 DECIMAL
11
12
13
14
15
```

```
SCREEN #68
 0 \ NOP SERVICE CALL WORD
 1 DECIMAL
 2 : NOP ;
 3 ' NOP CFA    0 SERVICE-ENTRY      \ False hit on status register
 4 ' NOP CFA  255 SERVICE-ENTRY      \ Halt command
 5
 6
 7
 8
 9
10
11
12
13
14
15

SCREEN #69
 0 EXIT  \ SERVICE CALL WAIT LOOP WORD
 1 HEX
 2 \  Uses the DECODE instruction in SYSCALL OFFSET:2 to restart
 3 : BSERVICE   ( -> )
 4    BEGIN
 5       BEGIN
 6          ?TERMINAL 3 =  ABORT" BREAK..."
 7       STATUS  UNTIL
 8    STOP    STATUS
 9    DUP >R  OFF AND  2*  SERVICE-ARRAY  + @  EXECUTE
10    R>          OFF =  UNTIL 0 EPRINT ! ;
11 DECIMAL
12
13
14
15

SCREEN #70
 0 \ SERVICE CALL WAIT LOOP WORD  -- HIGH SPEED
 1 HEX
 2 \  Uses the DECODE instruction in SYSCALL OFFSET:2 to restart
 3 CODE DO-SERVICE   ( -> STATUS   EXECUTE-ADDRESS )
 4    DX , # 302 MOV
 5    BEGIN   DX , AL IN  AL , AL OR  <>0? UNTIL
 6    DX , # 304 MOV   DX , AL OUT    DX , # 302 MOV
 7    DX , AL IN    AH , AH XOR     AX PUSH    BX , AX MOV
 8    BX , BX ADD   AX , SERVICE-ARRAY [BX] MOV    AX PUSH
 9 NEXT JMP  END-CODE
10 : BSERVICE   ( -> )
11    BEGIN    DO-SERVICE
12    EXECUTE
13    OFF =  UNTIL  0 EPRINT ! ;
14 DECIMAL
15

SCREEN #71
 0 \ EXECUTE A PROGRAM ON THE BOARD --  BEXECUTE
 1 HEX   MATH
 2 \  Uses the NOP instruction in OPCODE:0 OFFSET:5 to run
 3 : BEXECUTE   ( ..STACKIN.  DADDR -> ..STACKOUT.. )
 4       D>R   ?STACK
```

```
 5    [ >> DEST=STATUS ;SET ] MIR-SETUP  0. X!  ( Reset status )
 6    [ >> DEST=FLAGS  ;SET ] MIR-SETUP -1. X!  ( Mask intrpts )
 7    DS->BOARD
 8    [ >> DEST=PAGE  ;SET ] MIR-SETUP  DR> DDUP   X!
 9    [ >> DEST=ADDRESS-LATCH    ;SET ] MIR-SETUP  DDUP X!
10    [ >> DEST=DECODE  ;SET ] MIR-SETUP
11                DDUP 7FFFFC. DAND  X!
12    [ >> DECODE  ;SET ] MIR-SETUP CYCLE
13    [ 5 :: JMP=100 ;SET ] MIR-SETUP CYCLE CYCLE
14     ." START.."   GO    BSERVICE     ." END    "
15     BOARD->DS  0 EPRINT !   ; DECIMAL

SCREEN #72
 0 \ OP-EXECUTE        BOARD
 1 HEX MATH
 2 : OP-EXECUTE  ( D -> )   \ Single step execute a primitive
 3    ( If addr info present, make it a subroutine call )
 4    FFBFFFFF.  DAND     8.   4. RAM!       ( nop )
 5   0C.   8. RAM!       ( nop )
 6    RESET-BOARD    DDUP  07FFFFC. DAND  D0=
 7    IF  10. DOR  ELSE  2. DOR  THEN  0C. RAM!
 8    FF800010. ( HALT = OP-CODE 511 ) 10.  RAM!
 9    4. BEXECUTE      ;
10 : OPCODE-EXEC  ( N -> )
11    0  17 DLSLN  OP-EXECUTE ;
12 \  Jump start the board using the COLD execution vector
13 : BOARD   ( -> )
14    RESET-BOARD 14. BEXECUTE  ;
15 : CPU32   BOARD ;           DECIMAL

SCREEN #73
 0 \ REDEFINE INTERPRET FOR  BOARD VOCABULARY USAGE
 1 DECIMAL
 2 VARIABLE BOARD-FENCE    HERE BOARD-FENCE !
 3 : BOARD-INTERPRET
 4    BEGIN  -FIND
 5       IF
 6          DROP  DUP     BOARD-FENCE @   U<
 7          IF    CFA  EXECUTE
 8          ELSE  D@  OP-EXECUTE
 9          THEN
10       ELSE  HERE  NUMBER        DPL @ 1+
11                                 NOT  IF  DROP   THEN
12       THEN
13    ?STACK   AGAIN    ;
14
15 ' BOARD-INTERPRET  CFA  'BINTER !

SCREEN #74
 0 \ CROSS COMPILER VOCABULARY SETUP
 1 DECIMAL
 2 : C-RES
 3    [ 'INTERPRET @ ] LITERAL   'INTERPRET ! ; \ Restore vector
 4
 5 VARIABLE 'CINTER    'INTERPRET @  'CINTER !
 6
 7 VOCABULARY CROSS-VOC  IMMEDIATE
 8
 9 : CROSS-COMPILER  ( -> )
10    [COMPILE]  CROSS-VOC    'CINTER @  'INTERPRET !  INTERPRET ;
11                                    IMMEDIATE
```

```
12
13 : CC  [COMPILE]  CROSS-COMPILER ;
14
15

SCREEN #75
 0 \ CROSS COMPILER VOCABULARY SETUP - 2
 1 DECIMAL
 2 CROSS-VOC  DEFINITIONS
 3 : FORTH-VOC   [COMPILE]  FORTH  ;  IMMEDIATE
 4
 5 : FORTH     C-RES    [COMPILE] FORTH  INTERPRET ; IMMEDIATE
 6
 7 : EDIT      C-RES  EDIT    INTERPRET  ;
 8 : EDITOR    C-RES  [COMPILE]  EDITOR INTERPRET ;      IMMEDIATE
 9 FORTH DEFINITIONS
10 : ;;END  [COMPILE]  CROSS-COMPILER ;
11 CROSS-COMPILER DEFINITIONS
12
13
14
15

SCREEN #76
 0 \  CONSTANTS FOR MICRO-CODE REFERENCES BY CROSS-ASSEMBLER
 1 DECIMAL  CROSS-COMPILER  DEFINITIONS      MATH
 2 : OP-VALUE    00  23 DLSLN  DCONSTANT ;
 3 0       OP-VALUE [NOP]          137  OP-VALUE [1]
 4 34      OP-VALUE [DO]
 5 511     OP-VALUE [HALT]          93  OP-VALUE [SYSCALL]
 6 67      OP-VALUE [DOCON]
 7 : OP-CALL-VALUE    00  23 DLSLN  2. D+ DCONSTANT ;
 8 15  OP-CALL-VALUE [OBRANCH]
 9 25  OP-CALL-VALUE [+LOOP]
10 167 OP-CALL-VALUE [LOOP]
11 68  OP-CALL-VALUE [DOVAR]
12 78  OP-CALL-VALUE [LIT]
13 : OP-EXIT-VALUE    00  23 DLSLN  1. D+ DCONSTANT ;
14 00 OP-EXIT-VALUE [EXIT]
15

SCREEN #77
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

```
SCREEN #78
 0 \ OPTIMIZING COMPILER VARIABLES, ETC.
 1 DECIMAL
 2 VARIABLE OPT-STATUS
 3    \ 1 = previous instruction is a defaulted JMP
 4    \ 4 = previous instruction is a CALL
 5    \ 8 = previous instruction can not be changed
 6
 7 : DEFAULT-JUMP   1 OPT-STATUS ! ;
 8 : IS-A-CALL      4 OPT-STATUS ! ;
 9 : DON'T-DISTURB  8 OPT-STATUS ! ;
10
11
12
13
14
15

SCREEN #79
 0 \ CROSS-COMPILER DICTIONARY HANDLING   --1
 1 DECIMAL  CROSS-COMPILER  DEFINITIONS        MATH
 2 32 CONSTANT USER-AREA
 3 VARIABLE BDP   200 BDP !      \ 10 .. 200 is USER area
 4 : BHERE  ( -> N )
 5    BDP @  ;
 6
 7 : DATA,  ( D -> )    \ Send data word to the dictionary
 8    BHERE  FORTH  0 RAM!  CROSS-COMPILER  4 BDP +! ;
 9
10 : DATAC,  ( B -> )   \ Send data byte to the dictionary
11    BHERE  FORTH  0 RAMC!  CROSS-COMPILER  1 BDP +! ;
12
13
14
15

SCREEN #80
 0 \ CROSS-COMPILER DICTIONARY HANDLING   --2
 1 HEX      CROSS-COMPILER  DEFINITIONS        MATH
 2 : OPCODE,  ( D-MICRO-CODE-INSTRUCTION -> )
 3    FORTH  FFBFFFFF. DAND   CROSS-COMPILER \ Mask "special" bit
 4    BHERE  4 +    0  DOR   DATA,   DEFAULT-JUMP ;
 5
 6 : CALL,    ( D-Subroutine-address -> )
 7    OPTIMIZE? @
 8    IF  OPT-STATUS @ 1 =
 9       IF  -4 BDP +!  BHERE 0  FORTH RAM@ CROSS-COMPILER
10           FF800000. DAND  DOR  THEN
11    THEN
12    IS-A-CALL    2 0 DOR  DATA, ;
13 DECIMAL
14
15

SCREEN #81
 0 \ CROSS-COMPILER DICTIONARY HANDLING   --3
 1 HEX      CROSS-COMPILER  DEFINITIONS        MATH
 2 : EXIT,    ( -> )
 3    1 0
 4    OPTIMIZE? @
```

```
 5     IF OPT-STATUS @ 1 =     \ If 1 status, combine the EXIT
 6        IF  -4 BDP +!  BHERE 0  FORTH RAM@ CROSS-COMPILER
 7             FF800000. DAND  DOR   THEN
 8        OPT-STATUS @ 4 =     \ If 4 status, do tail recurs. elim.
 9        IF  DDROP  -4 BDP +!  BHERE 0  FORTH RAM@ CROSS-COMPILER
10             FFFFFFFC. DAND  THEN
11     THEN
12     DATA, DON'T-DISTURB ;
13
14 DECIMAL
15

SCREEN #82
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15

SCREEN #83
 0 \ CROSS-COMPILER LITERAL HANDLING
 1 DECIMAL  CROSS-COMPILER  DEFINITIONS
 2 : BLITERAL  ( D -> ../D )
 3    STATE @
 4    IF  [LIT]  BHERE  8 + 0  D+  DATA,
 5                DATA,  DON'T-DISTURB   THEN ;
 6
 7 : DBLITERAL  ( Q -> ../Q )
 8    STATE @
 9    IF  DSWAP   BLITERAL  BLITERAL  THEN  ;
10
11 HEX
12 : WORD-ALIGN  ( -> )
13    4  BDP @  3 AND  -   3 AND
14    ?DUP  IF  0 DO  BL DATAC, LOOP  THEN ;
15 DECIMAL

SCREEN #84
 0 \ CREATE  FOR CROSS-COMPILER
 1 HEX   CROSS-COMPILER  DEFINITIONS
 2 VARIABLE  BLATEST              0 BLATEST !
 3 : BCREATE   ( -> )    \ Create a header on both IBM and Board
 4    [COMPILE] BOARD-VOC  DEFINITIONS FORTH RESET-BOARD CREATE
 5    CROSS-COMPILER   ( At run time BOARD-VOC is still current )
 6    WORD-ALIGN   BLATEST  @  0 DATA,   BHERE  BLATEST  !
 7    LATEST   C@  01F AND  8000  DATA,
 8    LATEST   DUP  C@  01F AND   0
 9    DO   1+   DUP C@  7F AND  DATAC,   LOOP    DROP
10    WORD-ALIGN   [COMPILE] CROSS-VOC DEFINITIONS DON'T-DISTURB ;
11
```

```
12 : BSMUDGE   ( -> )
13    BLATEST @   DUP  FORTH  0 RAM@   2000  XOR
14    ROT   0 RAM!     CROSS-COMPILER   ;
15 DECIMAL

SCREEN #85
 0 \ OPCODE:  --- DEFINE A MICROCODE DEFINITION
 1 HEX  CROSS-COMPILER  DEFINITIONS
 2 : OPCODE:  ( N -> )  ( LOADING: Usage  17 OPCODE: <name> )
 3    ( Executing <name> gives formatted micro-code reference )
 4    DUP  INSTRUCTION
 5    BCREATE       0 17 DLSLN  DDUP D,  OPCODE,   DON'T-DISTURB
 6    BLATEST @
 7    FORTH  DUP  0 RAM@  1000  OR  ROT  0 RAM!
 8    CROSS-COMPILER    EXIT,
 9    [COMPILE]   FORTH    ;
10
11 DECIMAL
12
13
14
15

SCREEN #86
 0 \  ABORT"  ."  FOR CROSS-COMPILER
 1 HEX    CROSS-COMPILER  DEFINITIONS
 2 VARIABLE ABORT"-ADDR     VARIABLE  ."-ADDR
 3 BOARD-VOC  DEFINITIONS
 4 : ABORT"   ( FLAG -> )   CROSS-COMPILER   ( Evenly /4 # bytes )
 5    ABORT"-ADDR @   0   DATA,         22 WORD
 6    COUNT  DUP 3 + FFFC AND  0 DATA,  OVER + SWAP
 7    DO  I C@ DATAC,  1 /LOOP  WORD-ALIGN   ( Fills w/blanks )
 8    DON'T-DISTURB ;   IMMEDIATE
 9
10 : ."   ( -> )                  CROSS-COMPILER
11    ."-ADDR @    0   DATA,         22 WORD
12    COUNT   DUP 0 DATA,     OVER +  SWAP
13    DO  I C@ DATAC,   1 /LOOP   WORD-ALIGN
14    DON'T-DISTURB ;   IMMEDIATE
15 CROSS-COMPILER   DEFINITIONS     DECIMAL

SCREEN #87
 0 \ IF..THEN  FOR CROSS-COMPILER
 1 HEX   BOARD-VOC  DEFINITIONS
 2 : IF      ( -> PATCH-ADDR 222 )
 3    CROSS-COMPILER  BDP @   [OBRANCH] DATA,  222
 4    DON'T-DISTURB ;   IMMEDIATE
 5
 6 : THEN   ( PATCH-ADDR 222 -> )
 7    CROSS-COMPILER   222 ?PAIRS   FORTH     DUP 0 RAM@
 8    CROSS-COMPILER  BDP @ 0  D+   ROT  FORTH  0 RAM!
 9    CROSS-COMPILER  DON'T-DISTURB ;
10 IMMEDIATE
11
12 DECIMAL
13
14
15
```

```
SCREEN #88
 0 \ ELSE  FOR CROSS-COMPILER
 1 HEX   BOARD-VOC  DEFINITIONS
 2 : ELSE   ( PATCH-ADDR1 222 -> PATCH-ADDR2 222 )
 3    CROSS-COMPILER  222 ?PAIRS    0 0 DATA,   ( Jump NOP opcode )
 4    OPTIMIZE? @
 5    IF  OPT-STATUS @ 1 =
 6       IF  ( Redirect jump to after the THEN )
 7           -4 BDP +!     BDP @ 4 - DUP   FORTH  0 RAM@
 8                  FF800000. DAND ROT 0 RAM!  CROSS-COMPILER
 9       THEN
10    THEN
11    BDP @   4 -
12    SWAP   222   BOARD-VOC  [COMPILE] THEN    222   ;
13 IMMEDIATE
14 DECIMAL
15

SCREEN #89
 0 \ CROSS COMPILER   BEGIN  AGAIN
 1 HEX   BOARD-VOC DEFINITIONS      MATH
 2 : BEGIN     ( -> JMP-ADDR  333 )
 3    CROSS-COMPILER   ?COMP  BDP @   333  DON'T-DISTURB  ;
 4 IMMEDIATE
 5
 6 : AGAIN     ( JMP-ADDR 333 -> )
 7    CROSS-COMPILER   333 ?PAIRS   0
 8    OPTIMIZE? @
 9    IF  OPT-STATUS @ 1 =
10       IF  ( Redirect jump in previous word )
11          -4 BDP +!  BHERE 0  FORTH RAM@
12                        FF800000. DAND  DOR.     CROSS-COMPILER
13    THEN     THEN
14    DATA,   DON'T-DISTURB ;
15 IMMEDIATE             DECIMAL

SCREEN #90
 0 \ CROSS COMPILER    UNTIL
 1 HEX   BOARD-VOC DEFINITIONS
 2 : UNTIL     ( JMP-ADDR  333 -> )
 3    CROSS-COMPILER   333 ?PAIRS   0  [OBRANCH] D+  DATA,
 4    DON'T-DISTURB ;
 5 IMMEDIATE
 6
 7 CROSS-COMPILER  DEFINITIONS  DECIMAL
 8
 9
10
11
12
13
14
15

SCREEN #91
 0 \ CROSS COMPILER   WHILE  REPEAT
 1 HEX   BOARD-VOC  DEFINITIONS
 2 : WHILE     ( JMP-ADDR 333 ->  JMP-ADDR 333  PATCH-ADDR 224 )
 3    [COMPILE]  IF   2+  ;
 4 IMMEDIATE
```

```
 5
 6 : REPEAT     ( JMP-ADDR 333  PATCH-ADDR 224 -> )
 7     >R >R    [COMPILE] AGAIN
 8     R> R>    2-  [COMPILE] THEN  ;
 9 IMMEDIATE
10
11 CROSS-COMPILER  DEFINITIONS  DECIMAL
12
13
14
15

SCREEN #92
 0 \ CROSS-COMPILER ... DO  LOOP  +LOOP
 1 HEX   BOARD-VOC  DEFINITIONS
 2 : DO      ( -> JMP-ADDR 4444 )
 3     CROSS-COMPILER  [DO] OPCODE, DON'T-DISTURB  BDP @   4444 ;
 4 IMMEDIATE
 5
 6 : LOOP   ( JMP-ADDR  4444 -> )
 7     CROSS-COMPILER  4444 ?PAIRS  0 [LOOP] D+ DATA,
 8     DON'T-DISTURB  ;    IMMEDIATE
 9
10 : +LOOP    ( JMP-ADDR  4444 -> )
11     CROSS-COMPILER  4444 ?PAIRS  0 [+LOOP] D+ DATA,
12     DON'T-DISTURB  ;    IMMEDIATE
13
14 CROSS-COMPILER  DEFINITIONS  DECIMAL
15

SCREEN #93
 0 \ CONSTANT VARIABLE USER   FOR CROSS-COMPILER
 1 DECIMAL CROSS-COMPILER  DEFINITIONS
 2 VARIABLE  DOUSE-ADDR    \ MUST be set by cross-compiled code
 3 : USER         ( VALUE -> )
 4     BCREATE   BHERE 2 OR  0 D,     DOUSE-ADDR @  0  CALL,
 5      4 *  0 DATA,  [COMPILE] CROSS-COMPILER DEFINITIONS
 6     DON'T-DISTURB ;
 7 : VARIABLE
 8     BCREATE   BHERE 2 OR  0  D,
 9     [DOVAR]  OPCODE,  0 0 DATA,     DON'T-DISTURB
10     [COMPILE] CROSS-COMPILER DEFINITIONS ;
11
12 : CONSTANT     ( DVALUE -> )
13     BCREATE   BHERE 2 OR  0 D,
14     [DOCON]  OPCODE,       DATA,   DON'T-DISTURB
15     [COMPILE] CROSS-COMPILER DEFINITIONS  ;

SCREEN #94
 0 \ CROSS-COMPILER      EXIT
 1 HEX   BOARD-VOC  DEFINITIONS
 2 : EXIT    ( -> )
 3     CROSS-COMPILER  EXIT, ;
 4 IMMEDIATE
 5
 6 CROSS-COMPILER  DEFINITIONS  DECIMAL
 7
 8
 9
10
```

```
11
12
13
14
15

SCREEN #95
 0 \ BOARD VOCABULARY   DEFINITIONS & UTILITIES
 1 DECIMAL   BOARD-VOC  DEFINITIONS
 2 : }   ( -> )       \ Leave board vocabulary/execution mode
 3    B-RES  [COMPILE] CROSS-VOC   DEFINITIONS
 4     FORTH  R> R> DDROP
 5     [COMPILE]  CROSS-COMPILER  ;
 6
 7 CROSS-VOC DEFINITIONS
 8
 9 : BLIST      [COMPILE] BOARD-VOC   VLIST  [COMPILE]  CROSS-VOC ;
10
11 : B'         [COMPILE] BOARD-VOC  [COMPILE] '
12              [COMPILE] CROSS-VOC  ;   IMMEDIATE
13
14
15

SCREEN #96
 0 \   (  ; DEFINITION FOR CROSS-COMPILER
 1 HEX   MATH
 2 BOARD-VOC  DEFINITIONS
 3 : (      \ Required to allow stack comments
 4    -1 >IN +!   29 WORD   C@ 1+  HERE +
 5    C@  29 =  NOT  ?STREAM   ;     IMMEDIATE
 6
 7 : ;
 8       CROSS-VOC  EXIT,
 9       [COMPILE] CROSS-VOC  DEFINITIONS
10       BSMUDGE   SMUDGE   [COMPILE]  [  ;     IMMEDIATE
11
12 CROSS-COMPILER   DEFINITIONS     DECIMAL
13
14
15

SCREEN #97
 0 \ SEAL BOARD VOCABULARY
 1 HEX   BOARD-VOC
 2 ' X  ( X is the null used in the BOARD vocabulary )
 3 FORTH DEFINITIONS
 4 80C0  OVER   NFA  !      ( Make null header out of X )
 5
 6               LFA    0 SWAP    !
 7 HERE  BOARD-FENCE !
 8
 9 DECIMAL
10
11
12
13
14
15
```

```
SCREEN #98
 0 \ COLON DEFINITIONS FOR CROSS-COMPILER
 1 HEX
 2 CROSS-COMPILER   DEFINITIONS
 3 : SPECIAL  ( -> )   \ Used for DON'T-disturb tag by CPU/32
 4    HERE 4 - D@   40 OR  HERE 4 - D!  \ Set "special" bit
 5    BLATEST  @     FORTH
 6    DUP  0 RAM@  0800 OR  ROT  0 RAM! FORTH ;   CROSS-COMPILER
 7
 8 : POISON   ( -> )   \ Used for non-interactive tag by CPU/32
 9 \  A poison word crashes system when entered in immediate mode
10 \   e.g.  LIT, >R, ...      This makes outer interpreter better
11    BLATEST  @     FORTH
12    DUP  0 RAM@  0400 OR  ROT  0 RAM! FORTH ;  CROSS-COMPILER
13
14 CROSS-COMPILER  DECIMAL
15

SCREEN #99
 0 \ COLON DEFINITIONS FOR CROSS-COMPILER
 1 HEX
 2 CROSS-COMPILER   DEFINITIONS
 3 : IMMEDIATE  ( -> )
 4    BLATEST  @     FORTH
 5    DUP  0 RAM@  4000 OR  ROT  0 RAM!  FORTH ;   CROSS-COMPILER
 6
 7 : :     ( Usage:  : <name> )
 8    BCREATE    BHERE 2 OR   0 D,    [COMPILE]  BOARD-VOC
 9      SMUDGE  BSMUDGE   ] FORTH  ;
10 CROSS-COMPILER  DECIMAL
11
12
13
14
15

SCREEN #100
 0 \ REDEFINE INTERPRET FOR CROSS-COMPILER
 1 HEX      FORTH DEFINITIONS
 2 VARIABLE  CROSS-FENCE     HERE  CROSS-FENCE !
 3 : CROSS-INTERPRET
 4    BEGIN  -FIND
 5       IF  STATE @ <
 6          OVER  CROSS-FENCE @  U>
 7          IF    IF   D@   CROSS-COMPILER   OVER  2 AND
 8                     IF  CALL,  ELSE  DUP >R  OPCODE,
 9 ( Test "special" bit ) R> 40 AND IF DON'T-DISTURB THEN  THEN
10 FORTH          ELSE  D@   CROSS-COMPILER  OP-EXECUTE FORTH THEN
11          ELSE  ABORT" ILLEGAL COMPILAND"  CFA EXECUTE  THEN
12       ELSE   HERE   NUMBER   CROSS-COMPILER BLITERAL FORTH
13          DPL @ 1+  0=  STATE @ 0= AND IF  DROP THEN
14    THEN  ?STACK   AGAIN    ;
15 ' CROSS-INTERPRET  CFA  'CINTER !  DECIMAL  FORTH DEFINITIONS

SCREEN #101
 0
 1
 2
 3
 4
```

```
 5
 6
 7
 8
 9
10
11
12
13
14
15

SCREEN #102
 0 \  DUMP-RAM  AND INIT-STACKS .  TESTING/UTILITY WORDS
 1 FORTH DEFINITIONS
 2 DECIMAL
 3 : DUMP-RAM    ( START END -> )                MATH
 4   HEX CR     4 + SWAP DO                          \
 5        I . ." ="  I 0 RAM@  8 DU.R CR    4 /LOOP ;
 6
 7 : INIT-STACKS    ( -> )
 8    0 :: DEST=RS  DEC[RP]  ;SET
 9    4100 0 DO  0 0 X! LOOP
10    0 :: DEST=DS  DEC[DP]  ;SET
11    4100 0 DO  0 0 X! LOOP      ;
12
13
14
15

SCREEN #103
 0 \ 32-BIT FETCH AND STORE IN PROPER BOARD BYTE ORDER
 1 HEX
 2 CODE XD!   ( D ADDR -> )     \ Stores in lo..hi byte order
 3    BX POP   DX POP   AX POP
 4      [BX] , AL MOV      1 [BX] , AH MOV
 5    2 [BX] , DL MOV      3 [BX] , DH MOV
 6    NEXT JMP  END-CODE
 7
 8 CODE XD@   ( ADDR -> D )     \ Fetches in lo..hi byte order
 9    BX POP
10    AL , 0 [BX] MOV      AH , 1 [BX] MOV
11    DL , 2 [BX] MOV      DH , 3 [BX] MOV
12    AX PUSH  DX PUSH
13    NEXT JMP  END-CODE
14 DECIMAL
15

SCREEN #104
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
```

```
12
13
14
15


SCREEN #105
 0 EXIT \  32 BIT MICROCODE SOURCE - NOTES   & LOAD SCREEN
 1 ASSUME: 1. Top element in DS is in DHI reg.
 2
 3
 4 NOTE: THE OP-CODES OF NOP AND HALT
 5       MUST NOT BE CHANGED!!!!
 6       NOP = 0
 7       HALT = 1
 8
 9 1) Return stack too slow to get thru ALU on 1 cycle
10       Can copy to DS and to DLO, LATCH, ETC.
11 2) RAM too slow to go thru ALU during read
12 3) RS/RP Can't be used on 1st and last cycle (Except SPECIAL)
13 4) Don't use RS thru address-latch & do RAM on next cycle
14 5) Don't use Z test after ALU MATH function
15      (also means no ALU math function with END)

SCREEN #106
 0 \  MICROCODE --- NOP(2)                          (n) = # clock cycles
 1 DECIMAL  CROSS-COMPILER
 2 0 OPCODE: NOP     ( -> )     \  MUST be opcode 0 !!!!!!!!!
 3    0 ::  DECODE ;;
 4    1 ::  END ;;
 5
 6 \  Special NOP for initialization use in restarting board
 7    3 :: ;;
 8    4 :: ;;
 9    5 :: ;;
10    6 ::  DECODE ;;
11    7 ::  END  ;;
12 ;;END
13 SPECIAL
14
15


SCREEN #107
 0 \  MICROCODE --- HALT(4+X)
 1 DECIMAL  CROSS-COMPILER
 2 511 OPCODE: HALT     ( -> )       \ -1 is flag to PC host
 3    0 ::  SOURCE=DHI DEST=DLO      ALU=-1   DEST=DHI ;;
 4    1 ::  SOURCE=DLO   DEST=LATCH  ;;
 5    2 ::  SOURCE=DHI DEST=STATUS
 6          SOURCE=LATCH ALU=B  DEST=DHI ;;
 7    3 ::  JMP=011  ;;     \ Infinite loop
 8
 9 ;;END
10
11
12
13
14
15
```

```
 5 : COLD  ( -> )
 6    EMPTY-BUFFERS
 7    PAGE  CR CR
 8    19 SPACES ." The WISC CPU/16"  CR CR
 9    19 SPACES ." with MVPFORTH/16  " CR CR
10    1A SPACES ." 14 March 1987"  CR CR
11    0 EPRINT !
12    FIRST  USE !   FIRST  PREV !
13    DECIMAL  ABORT ;
14
15 DECIMAL


SCR #224
 0 \  MVP-FORTH SOURCE --  WHERE
 1 HEX   CROSS-COMPILER
 2 : WHERE  ( -> )
 3    BLK @
 4    IF  BLK @  DUP  SCR !   CR  CR  ." SCREEN #"
 5       DUP .  >IN @  3FF MIN  C/L  /MOD  DUP
 6       ." LINE #"  .  C/L  *  ROT  BLOCK  +.
 7       CR CR  C/L  -TRAILING  TYPE
 8       >IN  @   3FF >  1 AND  +
 9    ELSE  >IN @
10    THEN   CR  HERE 1+  @   DUP >R  -  HERE 1+ R@  +
11    1+  @  BL =
12    IF 1-  THEN  SPACES  R>  0
13    DO  5E  EMIT  LOOP  ;
14 B' WHERE @  ( <WHERE-ADDR> ! )   \ Vector for <ABORT">
15 DECIMAL


SCR #225
 0 \  MVP-FORTH SOURCE --  <LOAD>  LOAD   THRU
 1 HEX   CROSS-COMPILER
 2 : <LOAD>   ( N -> )
 3    ?DUP NOT   ABORT" UNLOADABLE"
 4    BLK @  >R  >IN  @  >R
 5    0 >IN !   BLK !  INTERPRET
 6    R>  >IN  !  R>  BLK !  ;
 7
 8 B' <LOAD> @  ( 'LOAD ! )
 9 : LOAD   ( N -> )
10    'LOAD @  EXECUTE ;
11
12 : THRU  ( N1 N2 -> )
13    1+ SWAP   DO I U.  I LOAD
14       ?TERMINAL  ABORT" BREAK..."  LOOP  ;
15 DECIMAL


SCR #226
 0 \  MVP-FORTH SOURCE --  CREATE
 1 HEX   CROSS-COMPILER
 2 : CREATE   ( -> )
 3    BL WORD   DUP   1+  @
 4    0= ABORT" ATTEMPTED TO REDEFINE NULL"
 5    DUP  CONTEXT @  @  <FIND>
 6    IF  DDROP  WARNING @
 7       IF  DUP  COUNT  TYPE  SPACE  ." ISN'T UNIQUE" CR
 8       THEN
 9    THEN
10    LATEST ,  ( Store LFA )   HERE  CURRENT  @  !
```

```
 6 ;;END
 7 SPECIAL
 8
 9
10
11
12
13
14
15

SCREEN #112
 0 \  MICROCODE --- !(4)       Store within same page
 1 DECIMAL  CROSS-COMPILER
 2 7 OPCODE: !     ( N ADDR -> )
 3     0 ::  ;;
 4     1 ::  SOURCE=DHI  DEST=ADDRESS-LATCH ;;
 5     2 ::  SOURCE=DS   DEST=RAM          INC[DP]  DECODE  ;;
 6     3 ::  SOURCE=DS  ALU=B  DEST=DHI    INC[DP]    END  ;;
 7
 8 ;;END
 9
10
11
12
13
14
15

SCREEN #113
 0 \  MICROCODE --- +(2)
 1 DECIMAL  CROSS-COMPILER
 2 8  OPCODE: +     ( N1 N2 -> NSUM )
 3     0 ::   SOURCE=DS  ALU=A+B   DEST=DHI     INC[DP]   DECODE ;;
 4     1 ::                                                 END  ;;
 5
 6 ;;END
 7
 8
 9
10
11
12
13
14
15

SCREEN #114
 0 \  MICROCODE --- +!(6)
 1 DECIMAL  CROSS-COMPILER
 2 9 OPCODE: +!    ( N  ADDR -> )
 3  \ Fetch operand
 4     0 ::  SOURCE=DS  INC[DP]  ;;  \ Load N into bus latch
 5     1 ::  SOURCE=DHI  DEST=ADDRESS-LATCH   ALU=B  DEST=DHI ;;
 6     2 ::  SOURCE=RAM   DEST=DLO ;;
 7     3 ::  SOURCE=DLO  ALU=A+B   DEST=DHI ;;
 8     4 ::  SOURCE=DHI  DEST=RAM                    DECODE  ;;
 9     5 ::  SOURCE=DS  ALU=B  DEST=DHI    INC[DP]    END  ;;
10 ;;END
11
12
```

```
 6     2 ::  SOURCE=DS  DEST=ADDRESS-COUNTER  INC[DP] ;;
 7     3 ::  ;;
 8     4 ::  SOURCE=RAM  DEST=DECODE      DECODE ;;
 9     5 ::  SOURCE=DS  ALU=B  DEST=DHI  INC[DP]   END ;;
10
11 ;;END
12 POISON  SPECIAL
13
14
15

SCREEN #216
 0 \  MICROCODE --- WFILL(8+2*N)
 1 DECIMAL  CROSS-COMPILER
 2 129 OPCODE: WFILL  ( ADDR WORD-COUNT WORD -> )
 3   \  Move count to DHI and value to DS
 4     0 ::  SOURCE=DS   DEST=LATCH ;;
 5     1 ::  SOURCE=LATCH  ALU=B  DEST=DHI
 6                 SOURCE=DHI DEST=DS  INC[DP] ;;
 7   \ Stash ADC value in DLO, set ADC to ADDR for fill
 8     2 ::  SOURCE=ADDRESS-COUNTER  DEST=DLO ;; \ Save ADC
 9     3 ::  SOURCE=DS  DEST=ADDRESS-COUNTER   ALU=A-1  DEST=DHI
10                   DEC[DP]  INC[MPC] ;;
11     4 ::                          JMP=OOS ;;
12
13
14
15

SCREEN #217
 0 \  MICROCODE --- WFILL  --2
 1 DECIMAL  CROSS-COMPILER
 2 130 CURRENT-OPCODE !
 3 ( Fill another word )
 4     0 ::  SOURCE=DS  DEST=RAM          ALU=A-1  DEST=DHI
 5               INC[ADC]     JMP=111 ;;
 6     7 ::  SOURCE=ADDRESS-COUNTER  DEST=ADDRESS-LATCH JMP=OOS ;;
 7
 8 ( Done )
 9     1 ::  INC[DP] SOURCE=DLO DEST=ADDRESS-COUNTER ;;
10     2 ::  INC[DP]   DECODE ;;
11     3 ::  SOURCE=DS  ALU=B  DEST=DHI  INC[DP]  END ;;
12
13 ;;END
14
15

SCREEN #218
 0 \  MICROCODE --- 1(3)
 1 DECIMAL  CROSS-COMPILER        \  ( Used by xcompiler!)
 2 137 OPCODE: 1  ( -> 1 )
 3     0 ::  DEC[DP]  SOURCE=DHI  DEST=DLO
 4                   ALU=0 DEST=DHI        ;;
 5     1 ::  SOURCE=DLO DEST=DS
 6                   ALU=A+1  DEST=DHI   DECODE  ;;
 7     2 ::  END ;;
 8 ;;END
 9
10
11
```

```
 2 13 OPCODE: 0<   ( N -> FLAG )
 3    0 ::  JMP=01S                          DECODE  ;;
 4
 5 ( >=0 )    2 ::    ALU=0   DEST=DHI   END  ;;
 6 ( <0 )     3 ::    ALU=-1  DEST=DHI   END  ;;
 7 ;;END
 8
 9
10
11
12
13
14
15

SCREEN #119
 0 \  MICROCODE --- 0=(2)
 1 DECIMAL  CROSS-COMPILER
 2 14 OPCODE: 0=   ( N -> FLAG )
 3    0 ::  JMP=01Z                          DECODE  ;;
 4
 5 (  =0 )   2 ::  ALU=-1  DEST=DHI   END  ;;
 6 ( <>0 )   3 ::  ALU=0   DEST=DHI   END  ;;
 7 ;;END
 8
 9
10
11
12
13
14
15

SCREEN #120
 0 \  MICROCODE --- OBRANCH(3/4)    Favors taking the branch
 1 DECIMAL  CROSS-COMPILER
 2 15 OPCODE: OBRANCH    ( FLAG -> )
 3 \  0 flag => branch     non-0 flag => go to next instruction
 4    \ Must ALWAYS be used with a CALL attribute!}!!!
 5    0 ::  SOURCE=ADDRESS-COUNTER  DEST=DLO  JMP=01Z ;;
 6 \ FLAG=0     Allow compiled branch to take affect
 7    2 ::  INC[RP] JMP=001  DECODE ;;
 8    1 ::  SOURCE=DS  ALU=B  DEST=DHI  INC[DP]  END ;;
 9 \ FLAG<>0    Override compiled branch
10    3 ::  SOURCE=DLO DEST=ADDRESS-COUNTER ;; ( Also sets latch )
11              \ Wait for RAM access
12    4 ::  SOURCE=RAM  DEST=DECODE   INC[RP]   DECODE ;;
13    5 ::  SOURCE=DS  ALU=B  DEST=DHI  INC[DP]  END ;;
14 ;;END   POISON SPECIAL
15

SCREEN #121
 0 \  MICROCODE --- 1+(2)
 1 DECIMAL  CROSS-COMPILER
 2 16 OPCODE: 1+    ( N -> N+1 )
 3    0 ::  ALU=A+1  DEST=DHI          DECODE  ;;
 4    1 ::                              END  ;;
 5 ;;END
 6
 7
```

```
 8
 9
10
11
12
13
14
15

SCREEN #122
 0 \  MICROCODE --- 1-(2)
 1 DECIMAL  CROSS-COMPILER
 2 17 OPCODE: 1-    ( N -> N-1 )
 3     0 ::  ALU=A-1    DEST=DHI          DECODE  ;;
 4     1 ::                               END  ;;
 5 ;;END
 6
 7
 8
 9
10
11
12
13
14
15

SCREEN #123
 0 \  MICROCODE --- 2*(2)
 1 DECIMAL  CROSS-COMPILER
 2 18 OPCODE: 2*    ( N -> N+N )
 3     0 ::  ALU=A+A    DEST=DHI         DECODE  ;;
 4     1 ::                               END  ;;
 5 ;;END
 6
 7
 8
 9
10
11
12
13
14
15

SCREEN #124
 0 \  MICROCODE --- 2/(3)
 1 DECIMAL  CROSS-COMPILER
 2 19 OPCODE: 2/    ( N -> N/2 )
 3     0 ::  SOURCE=DHI  DEST=DLO  ALU=A+1  DEST=DHI  JMP=01S  ;;
 4
 5 ( positive ) 2 ::  SOURCE=DLO ALU=B SR[ALU] CIN=0 DEST=DHI
 6                                          JMP=111  DECODE ;;
 7     7 ::  END ;;
 8
 9 ( negative ) 3 ::  ALU=A CIN=1  SR[ALU]   DEST=DHI
10                  JMP=10Z       DECODE ;;
11
12 ( =-1    )  4 ::  ALU=0  DEST=DHI     END ;;
13 ( <> -1 )   5 ::                      END  ;;
14 ;;END
15
```

```
SCREEN #125
 0 \  MICROCODE --- <(5)    #### BENEFITS FROM DEST=DHI uBIT ####
 1 DECIMAL  CROSS-COMPILER
 2 20 OPCODE:  <  ( B A -> FLAG )
 3 \ Test for different signs
 4    0 ::  SOURCE=DS  ALU=AxorB  ;;
 5    1 ::  SOURCE=DS  ALU=AxorB  JMP=01S ;;
 6 ( Same sign inputs )
 7    2 ::  SOURCE=DS  ALU=A-B-1 JMP=101 ;;
 8    5 ::  INC[DP]       JMP=11S    DECODE  ;;
 9          6 ::  ALU=-1  DEST=DHI    END  ;;
10          7 ::  ALU=0   DEST=DHI    END  ;;
11
12 ( Different sign inputs -- < if B is negative )
13    3 ::   SOURCE=DS  ALU=notB     JMP=101  ;;
14 ;;END
15


SCREEN #126
 0 \  MICROCODE --- <$=STEP>(13)
 1 DECIMAL  CROSS-COMPILER    \ Single character string match
 2 23 OPCODE: <$=STEP>
 3      ( FLG AD1 AD2 CNT -> FLG' AD1' AD2' CNT' )
 4 \  Increments AD1 AD2, Decrements CNT
 5 \   IF <>,  CNT <= -1  FLG <= 0
 6    0 ::  SOURCE=DHI  DEST=DLO  ;;
 7    1 ::  SOURCE=DS   ALU=B  DEST=DHI DEST=ADDRESS-LATCH
 8            DEC[RP] ;;
 9    2 ::  SOURCE=DLO  DEST=RS            ALU=A+1  DEST=DHI ;;
10    3 ::  SOURCE=DHI  DEST=DS   INC[DP]  ALU=0    DEST=DHI ;;
11    4 ::  SOURCE=RAM-BYTE  DEST=DLO      ALU=A+1 DEST=DHI ;;
12    5 ::  SOURCE=DS  ALU=A+B    DEST=DHI DEST=ADDRESS-LATCH
13                                          INC[MPC] ;;
14    6 ::  SOURCE=DHI  DEST=DS   JMP=000 ;;
15


SCREEN #127
 0 \  MICROCODE --- <$=STEP> --2
 1 DECIMAL  CROSS-COMPILER    \ Single character string match
 2 24 CURRENT-OPCODE !
 3    0 ::  SOURCE=DLO  ALU=B  DEST=DHI  ;;
 4    1 ::  SOURCE=RAM-BYTE  DEST=DLO  ;;
 5    2 ::  SOURCE=DLO  ALU=AxorB DEST=DHI ;;
 6    3 ::  ALU=0  DEST=DHI  INC[DP]
 7             SOURCE=RS  DEST=DLO  INC[RP]    JMP=10Z ;;
 8 ( Bytes are <>  Place 0 in flag, -1 in count )
 9    5 ::  SOURCE=DHI  DEST=DS    DEC[DP]    DECODE ;;
10    6 ::  ALU=notA    DEST=DHI   DEC[DP]    END    ;;
11
12 ( Bytes are =   Leave flag non-0, decrement count )
13    4 ::  SOURCE=DLO ALU=A-B  DEST=DHI
14          DEC[DP]                          JMP=110 DECODE  ;;
15 ;;END


SCREEN #128
 0 \  MICROCODE --- <+LOOP>(9/11)     --1
 1 DECIMAL  CROSS-COMPILER
 2 \ Always use with CALL attribute!!!!!
 3 \ IMPLICIT LOOP RANGE SPAN OF 80000000 Hex
 4 25 OPCODE: <+LOOP>   ( N -> )     ( RS: LIMIT COUNT -> ... )
 5  \ Capture non-looping address in address latch
```

```
 6     0 ::   INC[RP] ;;
 7     1 ::   SOURCE=RS  DEST=DLO          INC[MPC] ;;
 8     2 ::   SOURCE=DLO ALU=A+B  DEST=DHI
 9                                    JMP=01S ;;    \ Add N to counter
10
11
12
13
14
15

SCREEN #129
 0 \  MICROCODE --- <+LOOP>  -- 2
 1 DECIMAL  CROSS-COMPILER
 2 26 CURRENT-OPCODE !
 3 ( + N )   2 ::  SOURCE=DHI  DEST=RS INC[RP] JMP=000 ;;
 4     0 ::   SOURCE=RS    DEST=DLO           DEC[RP]  ;; \ Test count
 5     1 ::   SOURCE=DLO   ALU=A-B  INC[MPC]  JMP=111 ;; \ Test count
 6     7 ::   SOURCE=RS    DEC[RP] JMP=00S ;;
 7
 8 ( - N )   3 ::  SOURCE=DHI  DEST=RS INC[RP] ;;
 9     4 ::   SOURCE=RS    DEST=DLO           DEC[RP] ;; \ Test count
10     5 ::   SOURCE=DLO   ALU=A-B  INC[MPC]          ;; \ Test count
11     6 ::   SOURCE=RS    DEC[RP] JMP=01S ;;
12
13
14
15

SCREEN #130
 0 \  MICROCODE --- <+LOOP>  -- 3
 1 DECIMAL  CROSS-COMPILER
 2 27 CURRENT-OPCODE !
 3 ( +N DONE ) 0 :: SOURCE=RS DEST=ADDRESS-COUNTER
 4                    INC[RP] JMP=100 ;;
 5 ( -N DONE ) 3 :: SOURCE=RS DEST=ADDRESS-COUNTER INC[RP] ;;
 6     4 ::   INC[RP] ;;
 7     5 :: SOURCE=RAM  DEST=DECODE  DECODE INC[RP]  JMP=111 ;;
 8
 9 ( +N LOOP  ) 1 :: INC[RP] DECODE  JMP=111  ;;
10 ( -N LOOP  ) 2 :: INC[RP] DECODE  JMP=111  ;;
11
12     7 ::  SOURCE=DS  ALU=B  DEST=DHI  INC[DP] END ;;
13 ;;END
14 POISON SPECIAL
15

SCREEN #131
 0 \  MICROCODE --- <<ABORT">>(11/3)
 1 DECIMAL  CROSS-COMPILER
 2 29 OPCODE: <<ABORT">>   ( FLAG -> )
 3 \  If flag <> 0, then NOP   if flag = 0, then jump over message
 4     0 :: JMP=01Z  ALU=A+1  ;;
 5
 6 ( FLAG=0 )  2 ::  SOURCE=RS  DEST=DLO  ALU=A+1  JMP=100 ;;
 7      4 ::  SOURCE=DLO  ALU=A+B       DEST=ADDRESS-LATCH ;;
 8      5 ::  SOURCE=RAM  DEST=DLO ALU=A+1     INC[RP] INC[MPC]  ;;
 9      6 ::  SOURCE=DLO  ALU=A+B     DEST=DHI          JMP=000  ;;
10
11 ( FLAG<>0 )
```

```
12      3 ::                JMP=001                    DECODE ;;
13      1 :: SOURCE=DS ALU=B DEST=DHI INC[DP]          END    ;;
14
15


SCREEN #132
 0 \  MICROCODE --- <<ABORT">> --2
 1 DECIMAL  CROSS-COMPILER
 2 30 CURRENT-OPCODE !
 3 \  If flag <> 0, then NOP   if flag = 0, then jump over message
 4      0 ::  ALU=A+1  DEST=DHI                    ;;
 5      1 ::  SOURCE=DHI  DEST=ADDRESS-COUNTER ;;    \ RS+count+7
 6      2 ::  SOURCE=RAM  DEST=DECODE   DECODE  ;;
 7      3 ::  SOURCE=DS ALU=B DEST=DHI INC[DP]   END    ;;
 8 ;;END  POISON SPECIAL
 9
10
11
12
13
14
15


SCREEN #133
 0 \  MICROCODE --- <DO>(4)
 1 DECIMAL  CROSS-COMPILER
 2 \ Must always be used with a JUMP since it starts a loop
 3 34 OPCODE: <DO>  ( LIMIT  START -> )
 4 \  Return:       ( -> LIMIT  START )
 5    0 ::   DEC[RP]  ;;
 6    1 ::   SOURCE=DS  DEST=RS  DEC[RP]  INC[DP]  ;;
 7    2 ::   SOURCE=DHI DEST=RS          DECODE ;;
 8    3 ::   SOURCE=DS  ALU=B    DEST=DHI INC[DP]  END ;;
 9 ;;END
10 POISON SPECIAL
11
12
13
14
15


SCREEN #134
 0 \  MICROCODE --- <ENCLA>  (11)
 1 DECIMAL  CROSS-COMPILER \ SCAN TO FIRST NON-DELIMITER
 2 35 OPCODE: <ENCLA>   ( ADDR -> ADDR+1/ADDR  DONE-FLAG )
 3 \                RETURN STACK  ( DELIM -> DELIM )
 4    0 ::  DEC[DP] ;;
 5    1 ::  SOURCE=DHI  DEST=DS   ;;
 6    2 ::  SOURCE=DHI  DEST=ADDRESS-LATCH ;;
 7    3 ::  SOURCE=RS   DEST=DLO         ;;
 8    4 ::  SOURCE=DLO  ALU=B  DEST=DHI ;;
 9 \ Compare with delimiter
10    5 ::  SOURCE=RAM-BYTE  DEST=DLO ;;
11    6 ::  SOURCE=DLO       ALU=AxorB  DEST=DHI  INC[MPC] ;;
12    7 ::  SOURCE=DS   ALU=B  DEST=DHI            JMP=00Z ;;
13
14
15
```

```
 5     1 ::   ALU=A-1   DEST=DHI   ;;
 6     2 ::   SOURCE=DHI   DEST=DP   ;;
 7     3 ::   SOURCE=DS    ALU=B     DEST=DHI      DECODE ;;
 8     4 ::   SOURCE=DLO              DEST=DP       END  ;;
 9
10 ;;END
11
12
13
14
15

SCREEN #139
 0 \  MICROCODE --- <ROLL>(3*N+6)
 1 DECIMAL  CROSS-COMPILER
 2 42 OPCODE: <ROLL>    ( ..STACK.. COUNT-1 -> ..NEW-STACK.. )
 3     0 ::   SOURCE=DS  DEST=DLO    INC[DP]   ;;
 4     1 ::   ALU=A-1    DEST=DHI    DEC[RP]       INC[MPC] ;;
 5     2 ::   SOURCE=DP  DEST=RS     DEC[RP]   JMP=00Z   ;;
 6
 7
 8
 9
10
11
12
13
14
15

SCREEN #140
 0 \  MICROCODE --- <ROLL>        --2
 1 DECIMAL  CROSS-COMPILER
 2 43 CURRENT-OPCODE !    ( ..STACK.. COUNT-1 -> ..NEW-STACK.. )
 3 ( LOOP UNTIL COUNT REACHES 0 )
 4     1 ::   SOURCE=DS  DEST=RS        ALU=A-1   DEST=DHI ;;
 5     2 ::   SOURCE=DLO    DEST=DS     ;;
 6     3 ::   SOURCE=RS  DEST=DLO  INC[DP]    JMP=00Z ;;
 7
 8 ( Done )    0 ::   INC[RP]  SOURCE=DLO  ALU=B  DEST=DHI JMP=101 ;;
 9     5 ::   SOURCE=RS  DEST=DP    INC[RP]  DECODE ;;
10     6 ::    END   ;;
11 ;;END
12
13
14
15

SCREEN #141
 0 \  MICROCODE --- =(3)
 1 DECIMAL  CROSS-COMPILER
 2 44 OPCODE:   =   ( A B -> FLAG )
 3     0 ::   SOURCE=DS  ALU=AxorB   ;;    \ Test for =
 4     1 ::   INC[DP]      JMP=01Z                DECODE   ;;
 5
 6 ( <> )    3 ::   ALU=0    DEST=DHI              END   ;;
 7 ( =  )    2 ::   ALU=-1   DEST=DHI              END   ;;
 8
 9 ;;END
10
11
```

```
SCREEN #135
 0 \  MICROCODE --- <ENCLA>  --2
 1 DECIMAL  CROSS-COMPILER \ SCAN TO FIRST NON-DELIMITER
 2 36 CURRENT-OPCODE !
 3 ( = DELIMITER )
 4    0 ::  ALU=A+1  DEST=DHI   JMP=111   DECODE ;;
 5    7 ::  SOURCE=DHI  DEST=DS  ALU=0   DEST=DHI          END ;;
 6
 7 ( <> DELIMITER )
 8    1 ::  ALU=A    DEST=DHI              DECODE ;;
 9    2 ::  SOURCE=DHI  DEST=DS  ALU=-1  DEST=DHI          END ;;
10
11 ;;END
12
13
14
15

SCREEN #136
 0 \  MICROCODE --- <ENCLB>  (10/11)
 1 DECIMAL  CROSS-COMPILER \ SCAN TO FIRST DELIMITER
 2 37 OPCODE: <ENCLB>   ( ADDR -> ADDR+1/ADDR  DONE-FLAG )
 3 \                RETURN STACK  ( DELIM -> DELIM )
 4    0 ::  DECIDP] ;;
 5    1 ::  SOURCE=DHI  DEST=DS ;;
 6    2 ::  SOURCE=DHI  DEST=ADDRESS-LATCH ;;
 7    3 ::  SOURCE=RS   DEST=DLO ;;
 8    4 ::  SOURCE=DLO  ALU=B  DEST=DHI ;;
 9 \ Compare with delimiter
10    5 ::  SOURCE=RAM-BYTE  DEST=DLO   ;;
11    6 ::  SOURCE=DLO       ALU=AxorB  DEST=DHI  INCIMPC] ;;
12 \ Compare with 0
13    7 ::  SOURCE=DLO       ALU=B      DEST=DHI   JMP=00Z ;;
14
15

SCREEN #137
 0 \  MICROCODE --- <ENCLB>  --2
 1 DECIMAL  CROSS-COMPILER \ SCAN TO FIRST NON-DELIMITER
 2 38 CURRENT-OPCODE !
 3 ( = DELIMITER )
 4    0 ::                      JMP=111   DECODE ;;
 5    7 ::                       ALU=-1  DEST=DHI          END ;;
 6
 7 ( <> DELIMITER )
 8    1 ::  SOURCE=DS   ALU=B  DEST=DHI   JMP=01Z ;;
 9    ( = 0 )
10    2 ::                      JMP=111   DECODE ;;
11    ( <> DELIMITER <>0 -- Go past this character )
12    3 ::  ALU=A+1  DEST=DHI              DECODE ;;
13    4 ::  SOURCE=DHI  DEST=DS  ALU=0   DEST=DHI          END ;;
14 ;;END
15

SCREEN #138
 0 \  MICROCODE --- <PICK>(5) \ Does all of PICK exc. error check
 1 DECIMAL  CROSS-COMPILER
 2 41 OPCODE: <PICK>  ( ..STACK.. A -> ..STACK.. B )
 3    0 ::  SOURCE=DP  DEST=DLO                  \ Save DP value
 4          ALU=A+B  DEST=DHI ;;
```

```
12
13
14
15

SCREEN #142
 0 \  MICROCODE --- >(5)     #### BENEFITS FROM DEST=DHI uBIT ####
 1 DECIMAL  CROSS-COMPILER
 2 45 OPCODE:  >  ( B A -> FLAG )
 3 \ Test for different signs
 4    0 ::  SOURCE=DS  ALU=AxorB  DEST=DHI  ;;
 5    1 ::  SOURCE=DS  ALU=AxorB  DEST=DHI  JMP=01S ;;
 6 ( Same sign inputs )
 7    2 ::  SOURCE=DS  ALU=A-B   JMP=101 ;;
 8    5 ::  INC[DP]       JMP=11S    DECODE  ;;
 9          6 ::  ALU=0   DEST=DHI    END  ;;
10          7 ::  ALU=-1  DEST=DHI    END  ;;
11
12 ( Different sign inputs -- > if B is non-negative )
13    3 ::   SOURCE=DS  ALU=notB   DEST=DHI   JMP=101  ;;
14 ;;END
15

SCREEN #143
 0 \  MICROCODE --- >R(3)
 1 DECIMAL  CROSS-COMPILER
 2 46 OPCODE: >R    ( N -> )   ( Return: -> N )
 3    0 ::  DEC[RP] ;;
 4    1 ::  SOURCE=DHI  DEST=RS  DECODE ;;
 5    2 ::  SOURCE=DS   ALU=B  DEST=DHI   INC[DP]  END ;;
 6
 7 ;;END
 8 POISON SPECIAL
 9
10
11
12
13
14
15

SCREEN #144
 0 \  MICROCODE --- ?DUP(2)
 1 DECIMAL  CROSS-COMPILER
 2 47 OPCODE:  ?DUP   ( A -> 0 / A A )
 3    0 ::   DEC[DP]   JMP=01Z                          DECODE  ;;
 4
 5 ( =0  )   2 ::  INC[DP]                              END ;;
 6 ( <>0 )   3 ::  SOURCE=DHI            DEST=DS       END ;;
 7
 8 ;;END
 9
10
11
12
13
14
15

SCREEN #145
 0 \  MICROCODE --- @(4)
 1 DECIMAL  CROSS-COMPILER
```

```
 2 48 OPCODE: @    ( ADDR -> N )
 3    0 ::  ;;
 4    1 ::  SOURCE=DHI    DEST=ADDRESS-LATCH ;;
 5    2 ::  SOURCE=RAM  DEST=DLO        DECODE ;;
 6    3 ::  SOURCE=DLO  ALU=B  DEST=DHI          END    ;;
 7
 8 ;;END
 9
10
11
12
13
14
15

SCREEN #146
 0 \  MICROCODE --- ABS(2)
 1 DECIMAL  CROSS-COMPILER
 2 49 OPCODE:  ABS    ( N1 -> N2 )
 3    0 ::  SOURCE=DHI DEST=DLO    ALU=A-1  DEST=DHI
 4                                        JMP=01S  DECODE ;;
 5
 6 ( >=0 )    2 ::  SOURCE=DLO  ALU=B     DEST=DHI      END  ;;
 7 (  <0 )    3 ::  SOURCE=DLO  ALU=notA DEST=DHI      END  ;;
 8
 9 ;;END
10
11
12
13
14
15

SCREEN #147
 0 \  MICROCODE --- ADC(4)
 1 DECIMAL  CROSS-COMPILER
 2 50 OPCODE: ADC    ( N1 N2 CYIN -> NSUM CYOUT )
 3    0 ::  SOURCE=DS  ALU=B  DEST=DHI  INC[DP]  JMP=10Z  ;;
 4
 5 ( CYIN<>0 )  5 ::  SOURCE=DS ALU=A+B+1  DEST=DHI JMP=011 ;;
 6 ( CYIN=0 )   4 ::  SOURCE=DS ALU=A+B    DEST=DHI JMP=011 ;;
 7
 8    3 :: SOURCE=DHI  DEST=DS  JMP=11C    DECODE ;;
 9
10 ( COUT<>0 )  6 ::  ALU=-1  DEST=DHI   END ;;
11 ( COUT=0  )  7 ::  ALU=0   DEST=DHI   END ;;
12
13 ;;END
14
15

SCREEN #148
 0 \  MICROCODE --- AND(2)
 1 DECIMAL  CROSS-COMPILER
 2 51 OPCODE: AND   ( N1 N2 -> N3 )
 3    0 ::  SOURCE=DS  ALU=AandB  DEST=DHI INC[DP]  DECODE ;;
 4    1 ::                  END  ;;
 5
 6 ;;END
 7
 8
```

```
 9
10
11
12
13
14
15

SCREEN #149
 0 \  MICROCODE --- ASR(2)
 1 DECIMAL  CROSS-COMPILER
 2 52 OPCODE: ASR   ( N1 -> N2 )
 3    0 ::    JMP=01S    DECODE ;;
 4
 5 ( POSITIVE )    2 ::  ALU=A CIN=0    SR[ALU]  DEST=DHI   END ;;
 6 ( NEGATIVE )    3 ::  ALU=A CIN=1    SR[ALU]  DEST=DHI   END ;;
 7
 8 ;;END
 9
10
11
12
13
14
15

SCREEN #150
 0 \  MICROCODE --- BYTEROLL(2)      (ROLL 8 BITS RIGHT)
 1 DECIMAL  CROSS-COMPILER
 2 53 OPCODE: BYTEROLL     ( N1 -> N2 )
 3    0 ::  ALU=A    ROLL[ALU]   DEST=DHI      DECODE ;;
 4    1 ::                                    END ;;
 5
 6 ;;END
 7
 8
 9
10
11
12
13
14
15

SCREEN #151
 0 \  MICROCODE --- C!(4)       Store within same page
 1 DECIMAL  CROSS-COMPILER
 2 54 OPCODE: C!    ( N ADDR -> )
 3    0 ::  ;;
 4    1 ::  SOURCE=DHI  DEST=ADDRESS-LATCH ;;
 5    2 ::  SOURCE=DS   DEST=RAM-BYTE     INC[DP]  DECODE  ;;
 6    3 ::  SOURCE=DS  ALU=B  DEST=DHI    INC[DP]     END  ;;
 7
 8 ;;END
 9
10
11
12
13
14
15
```

```
SCREEN #152
 0 \  MICROCODE --- C@(4)
 1 DECIMAL  CROSS-COMPILER
 2 55 OPCODE: C@     ( ADDR -> N )
 3     0 ::  ;;
 4     1 ::  SOURCE=DHI    DEST=ADDRESS-LATCH ;;
 5     2 ::  SOURCE=RAM-BYTE    DEST=DLO           DECODE ;;
 6     3 ::  SOURCE=DLO  ALU=B  DEST=DHI           END    ;;
 7
 8 ;;END
 9
10
11
12
13
14
15

SCREEN #153
 0 \  MICROCODE --- D!(7)
 1 DECIMAL  CROSS-COMPILER
 2 \ Lo half in ADDR,  Hi half in ADDR+4
 3 56 OPCODE: D!     ( D ADDR -> )
 4     0 ::  SOURCE=DHI  DEST=DLO  ALU=A+1  DEST=DHI  ;;
 5     1 ::  SOURCE=DLO  DEST=ADDRESS-LATCH  ALU=A+1  DEST=DHI
 6                       INC[DP] ;;
 7     2 ::  SOURCE=DS   DEST=RAM   ALU=A+1  DEST=DHI
 8                       DEC[DP] ;;
 9     3 ::  SOURCE=DS  DEST=DLO  INC[DP]  ALU=A+1  DEST=DHI ;;
10     4 ::  SOURCE=DHI  DEST=ADDRESS-LATCH   INC[DP] ;;
11     5 ::  SOURCE=DLO  DEST=RAM   DECODE ;;
12     6 ::  SOURCE=DS  ALU=B  DEST=DHI   INC[DP]  END ;;
13
14 ;;END
15

SCREEN #154
 0 \  MICROCODE --- D@(7)
 1 DECIMAL  CROSS-COMPILER
 2 \ Lo half in ADDR,  Hi half in ADDR+4
 3 58 OPCODE: D@     ( ADDR -> D )
 4     0 ::  SOURCE=DHI  DEST=DLO  ALU=A+1  DEST=DHI  DEC[DP] ;;
 5     1 ::  SOURCE=DLO  DEST=ADDRESS-LATCH  ALU=A+1  DEST=DHI  ;;
 6     2 ::  SOURCE=RAM  DEST=DLO   ALU=A+1  DEST=DHI   ;;
 7     3 ::  SOURCE=DLO DEST=DS     ALU=A+1  DEST=DHI ;;
 8     4 ::  SOURCE=DHI  DEST=ADDRESS-LATCH  ;;
 9     5 ::  SOURCE=RAM  DEST=DLO   DECODE ;;
10     6 ::  SOURCE=DLO ALU=B  DEST=DHI   END ;;
11
12 ;;END
13
14
15

SCREEN #155
 0 \  MICROCODE --- DDROP(2)
 1 DECIMAL  CROSS-COMPILER
 2 59 OPCODE: DDROP   ( D -> )
 3     0 ::  INC[DP]                                DECODE  ;;
 4     1 ::  SOURCE=DS  ALU=B    DEST=DHI INC[DP]     END  ;;
 5 ;;END
```

```
 6
 7
 8
 9
10
11
12
13
14
15

SCREEN #156
 0 \  MICROCODE --- DDUP(3)
 1 DECIMAL  CROSS-COMPILER
 2 60 OPCODE: DDUP   ( D1 -> D1 D1 )
 3    0 ::  SOURCE=DS  DEST=DLO          DEC[DP]  ;;
 4    1 ::  SOURCE=DHI           DEST=DS DEC[DP]   DECODE  ;;
 5    2 ::  SOURCE=DLO     DEST=DS                    END  ;;
 6 ;;END
 7
 8
 9
10
11
12
13
14
15

SCREEN #157
 0 \  MICROCODE --- DLSL(3)
 1 DECIMAL  CROSS-COMPILER
 2 62 OPCODE: DLSL  ( D1 -> D2 )
 3    0 ::  SOURCE=DS  DEST=DLO ;;
 4    1 ::  ALU=A CIN=0   SL[DLO] SL[ALU]  DEST=DHI  DECODE ;;
 5    2 ::  SOURCE=DLO  DEST=DS   END ;;
 6
 7 ;;END
 8
 9
10
11
12
13
14
15

SCREEN #158
 0 \  MICROCODE --- DLSR(3)
 1 DECIMAL  CROSS-COMPILER
 2 63 OPCODE: DLSR  ( D1 -> D2 )
 3    0 ::  SOURCE=DS  DEST=DLO ;;
 4    1 ::  ALU=A CIN=0   SR[DLO] SR[ALU]  DEST=DHI  DECODE ;;
 5    2 ::  SOURCE=DLO  DEST=DS   END ;;
 6
 7 ;;END
 8
 9
10
11
```

```
12
13
14
15

SCREEN #159
 0 \  MICROCODE --- DNEGATE(5)
 1 DECIMAL  CROSS-COMPILER
 2 66 OPCODE: DNEGATE  ( D1 -> -D1 )
 3    0 ::  SOURCE=DHI  DEST=DLO   ALU=0  DEST=DHI ;;
 4    1 ::  SOURCE=DS   ALU=A-B   DEST=DHI ;;
 5    2 ::  SOURCE=DHI  DEST=DS
 6               ALU=0  DEST=DHI  JMP=10C ;;
 7
 8 ( CY=1 ) 4 ::  SOURCE=DLO  ALU=AornotB+1  DEST=DHI
 9                                              JMP=110  DECODE ;;
10 ( CY=0 ) 5 ::  SOURCE=DLO  ALU=notB       DEST=DHI    DECODE ;;
11
12    6 ::  END ;;
13 ;;END
14
15

SCREEN #160
 0 \  MICROCODE --- DOCON(5)
 1 \ DOCON Must never be used with either CALL or EXIT attributes
 2 \    Address field for jump points to data word
 3 DECIMAL  CROSS-COMPILER
 4 67 OPCODE: DOCON   ( -> VALUE )
 5    0 ::  SOURCE=RAM  DEST=DLO            ;;
 6    1 ::  SOURCE=RS  DEST=ADDRESS-COUNTER  DEC[DP] INC[RP] ;;
 7    2 ::  SOURCE=DHI  DEST=DS         ;;
 8    3 ::  SOURCE=RAM  DEST=DECODE     DECODE ;;
 9    4 ::  SOURCE=DLO  ALU=B  DEST=DHI    END ;;
10
11 ;;END
12 POISON SPECIAL
13
14
15

SCREEN #161
 0 \  MICROCODE --- DOVAR(5)
 1 \ DOVAR Must be used with a CALL attribute
 2 \    Address field for jump points to data word
 3 DECIMAL  CROSS-COMPILER
 4 68 OPCODE: DOVAR   ( -> VALUE )
 5    0 ::  SOURCE=ADDRESS-COUNTER  DEST=DLO  DEC[DP] INC[RP] ;;
 6    1 ::  SOURCE=RS  DEST=ADDRESS-COUNTER   INC[RP] ;;
 7    2 ::  SOURCE=DHI  DEST=DS  ;;
 8    3 ::  SOURCE=RAM  DEST=DECODE    DECODE ;;
 9    4 ::  SOURCE=DLO ALU=B  DEST=DHI  END ;;
10 ;;END
11 POISON SPECIAL
12
13
14
15
```

```
SCREEN #162
 0 \  MICROCODE --- DROP(2)
 1 DECIMAL  CROSS-COMPILER
 2 69 OPCODE: DROP     ( N -> )
 3    0 ::  SOURCE=DS  ALU=B   DEST=DHI INC[DP]       DECODE  ;;
 4    1 ::                                            END  ;;
 5 ;;END
 6
 7
 8
 9
10
11
12
13
14
15

SCREEN #163
 0 \  MICROCODE --- DSWAP(7)
 1 DECIMAL  CROSS-COMPILER
 2 72 OPCODE: DSWAP ( D1 D2 -> D2 D1 )
 3    0 ::  SOURCE=DS  DEST=DLO  INC[DP] ;;
 4    1 ::  SOURCE=DS          DEC[RP] ;;  \ D1(hi) to ALU latch
 5    2 ::  SOURCE=DHI  DEST=DS    ALU=B  DEST=DHI  INC[DP] ;;
 6    3 ::  SOURCE=DS  DEST=RS ;;
 7    4 ::  SOURCE=DLO  DEST=DS   DEC[DP] ;;
 8    5 ::  SOURCE=RS  DEST=DLO  INC[RP]  DEC[DP]  DECODE  ;;
 9    6 ::  SOURCE=DLO  DEST=DS                     END    ;;
10
11 ;;END
12
13
14
15

SCREEN #164
 0 \  MICROCODE --- DUP(2)
 1 DECIMAL  CROSS-COMPILER
 2 73 OPCODE: DUP   ( N -> N N )
 3    0 ::   DEC[DP]                          DECODE  ;;
 4    1 ::   SOURCE=DHI          DEST=DS   END ;;
 5 ;;END
 6
 7
 8
 9
10
11
12
13
14
15

SCREEN #165
 0 \  MICROCODE --- I(2)
 1 DECIMAL  CROSS-COMPILER
 2 74 OPCODE: I    ( -> N )
 3    0 ::  SOURCE=RS  DEC[DP]   DECODE ;;
 4    1 ::  SOURCE=DHI  DEST=DS   ALU=B  DEST=DHI  END ;;
 5
```

```
 6 ;;END
 7 SPECIAL  \  Special forces RS available in 1st clock cycle
 8
 9
10
11
12
13
14
15

SCREEN #166
 0 \  MICROCODE --- R@(3)  -- SAME AS I
 1 DECIMAL  CROSS-COMPILER
 2 74 OPCODE: R@  ( -> N )
 3 ;;END
 4 SPECIAL  \  Special forces RS available in 1st clock cycle
 5
 6
 7
 8
 9
10
11
12
13
14
15

SCREEN #167
 0 \  MICROCODE --- I'(3)
 1 DECIMAL  CROSS-COMPILER
 2 75 OPCODE: I'   ( -> N )
 3    0 ::  DEC[DP] INC[RP] ;;
 4    1 ::  SOURCE=RS    DEC[RP]                    DECODE ;;
 5    2 ::  SOURCE=DHI  DEST=DS   ALU=B  DEST=DHI END  ;;
 6 ;;END
 7 SPECIAL  \  Special forces RS available in 1st clock cycle
 8
 9
10
11
12
13
14
15

SCREEN #168
 0 \  MICROCODE --- J(5)
 1 DECIMAL  CROSS-COMPILER
 2 76 OPCODE: J   ( -> N )
 3    0 ::  DEC[DP] INC[RP] ;;
 4    1 ::  SOURCE=DHI  DEST=DS   INC[RP] ;;
 5    2 ::  SOURCE=RS  DEST=DLO   DEC[RP]  ;;
 6    3 ::  DEC[RP]                          DECODE ;;
 7    4 ::  SOURCE=DLO  ALU=B  DEST=DHI      END  ;;
 8 ;;END
 9 SPECIAL  \  Special forces RS available in 1st clock cycle
10
11
12
```

```
13
14
15

SCREEN #169
 0 \  MICROCODE --- LEAVE(4)
 1 DECIMAL  CROSS-COMPILER
 2 77 OPCODE: LEAVE    ( -> )    ( Sets COUNT=LIMIT on return stack )
 3     0 ::  INC[RP] ;;
 4     1 ::  SOURCE=RS   DEST=DLO  DEC[RP] ;;
 5     2 ::  SOURCE=DLO DEST=RS      DECODE ;;
 6     3 ::  END ;;
 7 ;;END
 8 \  Special forces RS available in 1st clock cycle
 9 POISON SPECIAL
10
11
12
13
14
15

SCREEN #170
 0 \  MICROCODE --- LIT(5)
 1 DECIMAL  CROSS-COMPILER
 2 \ This MUST be used with the CALL attribute pointing next instr
 3 78 OPCODE: LIT    ( -> N )
 4     0 ::  SOURCE=ADDRESS-COUNTER  DEST=DLO    DEC[DP] ;;
 5     1 ::  SOURCE=DLO  DEST=ADDRESS-LATCH      INC[RP] ;;
 6     2 ::  SOURCE=RAM  DEST=DLO  ;;
 7     3 ::  SOURCE=DLO  DEST=LATCH  DECODE ;;
 8     4 ::  SOURCE=DHI  DEST=DS
 9                  SOURCE=LATCH    ALU=B  DEST=DHI       END ;;
10 ;;END
11 POISON SPECIAL
12
13
14
15

SCREEN #171
 0 \  MICROCODE --- LSLN(2*N+3)
 1 DECIMAL  CROSS-COMPILER
 2 \ COUNT TREATED AS AN UNSIGNED INTEGER >= 0
 3 79 OPCODE: LSLN    ( N1 COUNT -> N2 )
 4     0 ::  SOURCE=DS  DEST=DLO  INC[DP]
 5                        ALU=A-1 DEST=DHI  JMP=01Z';;
 6
 7 ( LOOP WHILE <> 0 )
 8     3 ::  CIN=0  SL[DLO]   ;;
 9     4 ::  ALU=A-1  DEST=DHI  JMP=01Z ;;
10
11 ( DONE  )
12     2 ::  SOURCE=DLO  ALU=B  DEST=DHI    JMP=111 DECODE ;;
13     7 ::                                        END ;;
14 ;;END
15

SCREEN #172
 0 \  MICROCODE --- LSR(2)
 1 DECIMAL  CROSS-COMPILER
```

```
 2 80 OPCODE: LSR  ( N1 -> N2 )
 3    0 ::  ALU=A CIN=0   SR[ALU]  DEST=DHI          DECODE ;;
 4    1 ::                                          END ;;
 5 ;;END
 6
 7
 8
 9
10
11
12
13
14
15

SCREEN #173
 0 \  MICROCODE --- LSRN(3*N+4)
 1 DECIMAL  CROSS-COMPILER
 2 \ COUNT TREATED AS AN UNSIGNED INTEGER >= 0
 3 81 OPCODE: LSRN   ( N1 COUNT -> N2 )
 4    0 ::  SOURCE=DS  DEST=DLO   ;;
 5    1 ::  SOURCE=DHI DEST=DS   ALU=0  DEST=DHI   JMP=01Z ;;
 6
 7 ( LOOP WHILE <> 0 )
 8    3 ::  SOURCE=DS ALU=A-B       DEST=DHI  ;;
 9    4 ::  ALU=notA  DEST=DHI ;;
10    5 ::  SOURCE=DHI DEST=DS  CIN=0  SR[DLO]
11                        ALU=0  DEST=DHI              JMP=01Z ;;
12 ( DONE  )
13    2 ::  SOURCE=DLO  ALU=B  DEST=DHI    JMP=111 DECODE ;;
14    7 ::                      INC[DP]               END ;;
15 ;;END

SCREEN #174
 0 \  MICROCODE --- NEGATE(2)
 1 DECIMAL  CROSS-COMPILER
 2 82 OPCODE: NEGATE   ( N -> -N )
 3    0 ::  ALU=A-1    DEST=DHI     DECODE  ;;
 4    1 ::  ALU=notA   DEST=DHI        END  ;;
 5 ;;END
 6
 7
 8
 9
10
11
12
13
14
15

SCREEN #175
 0 \  MICROCODE --- NOT(2)
 1 DECIMAL  CROSS-COMPILER
 2 83 OPCODE: NOT   ( N -> 0=FLAG )
 3    0 ::  JMP=01Z                      DECODE  ;;
 4
 5 ( <>0 )  3 ::   ALU=0   DEST=DHI   END  ;;
 6 (  =0 )  2 ::   ALU=-1  DEST=DHI   END  ;;
 7 ;;END
```

```
 8
 9
10
11
12
13
14
15

SCREEN #176
 0 \  MICROCODE --- OR(2)
 1 DECIMAL  CROSS-COMPILER
 2 84 OPCODE: OR   ( N1 N2 -> N3 )
 3    0 ::  SOURCE=DS  ALU=AorB  DEST=DHI  INC[DP]  DECODE ;;
 4    1 ::                                END  ;;
 5
 6 ;;END
 7
 8
 9
10
11
12
13
14
15

SCREEN #177
 0 \  MICROCODE --- OVER(2)
 1 DECIMAL  CROSS-COMPILER
 2 85 OPCODE: OVER   ( N1 N2 -> N1 N2 N1 )
 3     0 ::  SOURCE=DS  DEC[DP]     DECODE ;;  \ ALU latch <- N1
 4     1 ::  SOURCE=DHI  DEST=DS   ALU=B  DEST=DHI;  END ;;
 5
 6 ;;END
 7
 8
 9
10
11
12
13
14
15

SCREEN #178
 0 \  MICROCODE --- R>(2)
 1 DECIMAL  CROSS-COMPILER
 2 86 OPCODE: R>    ( -> N )   ( return: N -> )
 3    0 ::  SOURCE=RS  DEST=LATCH   DEC[DP]  INC[RP]  DECODE ;;
 4    1 ::  SOURCE=DHI  DEST=DS
 5             SOURCE=LATCH  ALU=B  DEST=DHI          END ;;
 6
 7 ;;END
 8 POISON   SPECIAL
 9
10
11
12
13
14
15
```

```
SCREEN #179
 0 \  MICROCODE --- RLC(3)
 1 DECIMAL  CROSS-COMPILER
 2 88 OPCODE: RLC    ( N1 CYIN -> N2 CYOUT )
 3    0 ::  SOURCE=DS  DEST=DLO  ALU=B  DEST=DHI  JMP=01Z  ;;
 4
 5 ( CYIN=0 )
 6    2 ::  SL[DLO]   CIN=0   JMP=10S  DECODE ;;
 7 ( CYIN<>0 )
 8    3 ::  SL[DLO]   CIN=1   JMP=10S  DECODE ;;
 9
10 ( COUT=0  )  4 :: SOURCE=DLO DEST=DS  ALU=0   DEST=DHI  END ;;
11 ( COUT<>0 )  5 :: SOURCE=DLO DEST=DS  ALU=-1  DEST=DHI  END ;;
12
13 ;;END
14
15

SCREEN #180
 0 \  MICROCODE --- ROT(4)
 1 DECIMAL  CROSS-COMPILER
 2 89 OPCODE: ROT    ( N1 N2 N3 -> N2 N3 N1 )
 3    0 ::  SOURCE=DS  ;;
 4    1 ::  SOURCE=DHI  DEST=DS   ALU=B  DEST=DHI INC[DP] ;;
 5    2 ::  SOURCE=DS                                    DECODE ;;
 6    3 ::  SOURCE=DHI  DEST=DS   ALU=B  DEST=DHI DEC[DP] END  ;;
 7 ;;END
 8
 9
10
11
12
13
14
15

SCREEN #181
 0 \  MICROCODE --- RRC(4)
 1 DECIMAL  CROSS-COMPILER
 2 90 OPCODE: RRC    ( N1 CYIN -> N2 CYOUT )
 3    0 ::  SOURCE=DS   DEST=DLO  JMP=01Z  ;; \ test if CYIN=0
 4
 5 ( CYIN=0 )
 6    2 ::  SOURCE=DS  ALU=B SR[ALU]   CIN=0   JMP=001  ;;
 7 ( CYIN<>0 )
 8    3 ::  SOURCE=DS  ALU=B SR[ALU]   CIN=1   JMP=001  ;;
 9
10    1 ::  SOURCE=DHI  DEST=DS   JMP=10L    DECODE ;;
11
12 ( COUT=0  )  4 :: ALU=0   DEST=DHI  END ;;
13 ( COUT<>0 )  5 :: ALU=-1  DEST=DHI  END ;;
14 ;;END
15

SCREEN #182
 0 \  MICROCODE --- S->D(3)
 1 DECIMAL  CROSS-COMPILER
 2 91 OPCODE:  S->D    ( N1 -> D2 )
 3    0 ::  DEC[DP]  JMP=01S   DECODE  ;;
 4
 5 ( >=0 )  2 ::  SOURCE=DHI  DEST=DS ALU=0  DEST=DHI   END  ;;
 6 ( <0  )  3 ::  SOURCE=DHI  DEST=DS ALU=-1 DEST=DHI   END  ;;
```

```
 7
 8 ;;END
 9
10
11
12
13
14
15

SCREEN #183
 0 \  MICROCODE --- SWAP(2)
 1 DECIMAL  CROSS-COMPILER
 2 92 OPCODE: SWAP     ( N1 N2 -> N2 N1 )
 3     0 ::  SOURCE=DS                DECODE ;;  \ ALU latch <- N1
 4     1 ::  SOURCE=DHI  DEST=DS    ALU=B  DEST=DHI    END ;;
 5
 6 ;;END
 7
 8
 9
10
11
12
13
14
15

SCREEN #184
 0 \  MICROCODE --- SYSCALL(7+host delay)
 1 DECIMAL  CROSS-COMPILER
 2 93 OPCODE: SYSCALL   ( DATA1  N -> DATA2 )
 3    0 ::  SOURCE=DS  DEST=LATCH  INC[DP] ;;
 4    1 ::  SOURCE=LATCH  ALU=B DEST=DHI
 5                    SOURCE=DHI  DEST=STATUS ;;
 6    2 ::  SOURCE=DHI  JMP=010  ;;     \ Infinite loop
 7
 8 \  Restart at address 4
 9    4 ::  ;;
10    5 ::  SOURCE=DHI  DEST=DLO   ALU=0   DEST=DHI ;;
11    6 ::  SOURCE=DHI  DEST=STATUS                DECODE ;;
12    7 ::  SOURCE=DLO  ALU=B   DEST=DHI             END  ;;
13
14 ;;END
15

SCREEN #185
 0 \  MICROCODE --- TOGGLE(6)
 1 DECIMAL  CROSS-COMPILER   \ Works on 8-bit value
 2 94 OPCODE: TOGGLE   ( ADDR B -> )
 3    0 ::  SOURCE=DS  DEST=DLO ;;
 4  \ Fetch operand
 5    1 ::  SOURCE=DLO  DEST=ADDRESS-LATCH   INC[DP] ;;
 6    2 ::  SOURCE=RAM-BYTE   DEST=DLO ;;
 7    3 ::  SOURCE=DLO  ALU=AxorB   DEST=DHI ;;
 8    4 ::  SOURCE=DHI  DEST=RAM-BYTE              DECODE  ;;
 9    5 ::  SOURCE=DS  ALU=B  DEST=DHI   INC[DP]    END  ;;
10
11 ;;END
12
```

```
13
14
15

SCREEN #186
 0 \  MICROCODE --- U*(36)
 1 DECIMAL  CROSS-COMPILER
 2 96 OPCODE: U*    ( U1 U2 -> UD3 )
 3 \  ALU LATCH = U1   DLO = U2     DHI/DLO = PRODUCT
 4     0 ::  SOURCE=DHI  DEST=DLO     ;;
 5     1 ::  SR[DLO]  ;;
 6     2 ::  SOURCE=DS  ALU=0  DEST=DHI SR[DLO]  JMP=10L ;;
 7
 8 \ NOTE:  MULTIPLY sets holds the ALU latch with U1
 9 ( BIT 0 )
10    4 ::  MULTIPLY  ALU=A+0 SR[ALU] SR[DLO] JMP=11L INC[MPC] ;;
11    5 ::  MULTIPLY  ALU=A+B SR[ALU] SR[DLO] JMP=11L INC[MPC] ;;
12 ( BIT 1 )
13    6 ::  MULTIPLY  ALU=A+0 SR[ALU] SR[DLO] JMP=00L ;;
14    7 ::  MULTIPLY  ALU=A+B SR[ALU] SR[DLO] JMP=00L ;;
15

SCREEN #187
 0 \  MICROCODE --- U*  -- 2
 1 DECIMAL  CROSS-COMPILER
 2 97 CURRENT-OPCODE !
 3 ( BIT 2 )
 4     0 ::  MULTIPLY  ALU=A+0 SR[ALU] SR[DLO] JMP=01L ;;
 5     1 ::  MULTIPLY  ALU=A+B SR[ALU] SR[DLO] JMP=01L ;;
 6 ( BIT 3 )
 7     2 ::  MULTIPLY  ALU=A+0 SR[ALU] SR[DLO] JMP=10L ;;
 8     3 ::  MULTIPLY  ALU=A+B SR[ALU] SR[DLO] JMP=10L ;;
 9 ( BIT 4 )
10     4 ::  MULTIPLY  ALU=A+0 SR[ALU] SR[DLO] JMP=11L INC[MPC] ;;
11     5 ::  MULTIPLY  ALU=A+B SR[ALU] SR[DLO] JMP=11L INC[MPC] ;;
12 ( BIT 5 )
13     6 ::  MULTIPLY  ALU=A+0 SR[ALU] SR[DLO] JMP=00L ;;
14     7 ::  MULTIPLY  ALU=A+B SR[ALU] SR[DLO] JMP=00L ;;
15

SCREEN #188
 0 \  MICROCODE --- U*  -- 3
 1 DECIMAL  CROSS-COMPILER
 2 98 CURRENT-OPCODE !
 3 ( BIT 6 )
 4     0 ::  MULTIPLY  ALU=A+0 SR[ALU] SR[DLO] JMP=01L ;;
 5     1 ::  MULTIPLY  ALU=A+B SR[ALU] SR[DLO] JMP=01L ;;
 6 ( BIT 7 )
 7     2 ::  MULTIPLY  ALU=A+0 SR[ALU] SR[DLO] JMP=10L ;;
 8     3 ::  MULTIPLY  ALU=A+B SR[ALU] SR[DLO] JMP=10L ;;
 9 ( BIT 8 )
10     4 ::  MULTIPLY  ALU=A+0 SR[ALU] SR[DLO] JMP=11L INC[MPC] ;;
11     5 ::  MULTIPLY  ALU=A+B SR[ALU] SR[DLO] JMP=11L INC[MPC] ;;
12 ( BIT 9 )
13     6 ::  MULTIPLY  ALU=A+0 SR[ALU] SR[DLO] JMP=00L ;;
14     7 ::  MULTIPLY  ALU=A+B SR[ALU] SR[DLO] JMP=00L ;;
15
```

```
SCREEN #189
 0 \  MICROCODE --- U*  -- 4
 1 DECIMAL  CROSS-COMPILER
 2 99 CURRENT-OPCODE !
 3 ( BIT 10 )
 4    0 ::  MULTIPLY  ALU=A+0 SR[ALU] SR[DLO] JMP=01L ;;
 5    1 ::  MULTIPLY  ALU=A+B SR[ALU] SR[DLO] JMP=01L ;;
 6 ( BIT 11 )
 7    2 ::  MULTIPLY  ALU=A+0 SR[ALU] SR[DLO] JMP=10L ;;
 8    3 ::  MULTIPLY  ALU=A+B SR[ALU] SR[DLO] JMP=10L ;;
 9 ( BIT 12 )
10    4 ::  MULTIPLY  ALU=A+0 SR[ALU] SR[DLO] JMP=11L INC[MPC] ;;
11    5 ::  MULTIPLY  ALU=A+B SR[ALU] SR[DLO] JMP=11L INC[MPC] ;;
12 ( BIT 13 )
13    6 ::  MULTIPLY  ALU=A+0 SR[ALU] SR[DLO] JMP=00L ;;
14    7 ::  MULTIPLY  ALU=A+B SR[ALU] SR[DLO] JMP=00L ;;
15

SCREEN #190
 0 \  MICROCODE --- U*  -- 5
 1 DECIMAL  CROSS-COMPILER
 2 100 CURRENT-OPCODE !
 3 ( BIT 14 )
 4    0 ::  MULTIPLY  ALU=A+0 SR[ALU] SR[DLO] JMP=01L ;;
 5    1 ::  MULTIPLY  ALU=A+B SR[ALU] SR[DLO] JMP=01L ;;
 6 ( BIT 15 )
 7    2 ::  MULTIPLY  ALU=A+0 SR[ALU] SR[DLO] JMP=10L ;;
 8    3 ::  MULTIPLY  ALU=A+B SR[ALU] SR[DLO] JMP=10L ;;
 9 ( BIT 16 )
10    4 ::  MULTIPLY  ALU=A+0 SR[ALU] SR[DLO] JMP=11L INC[MPC] ;;
11    5 ::  MULTIPLY  ALU=A+B SR[ALU] SR[DLO] JMP=11L INC[MPC] ;;
12 ( BIT 17 )
13    6 ::  MULTIPLY  ALU=A+0 SR[ALU] SR[DLO] JMP=00L ;;
14    7 ::  MULTIPLY  ALU=A+B SR[ALU] SR[DLO] JMP=00L ;;
15

SCREEN #191
 0 \  MICROCODE --- U*  -- 6
 1 DECIMAL  CROSS-COMPILER
 2 101 CURRENT-OPCODE !
 3 ( BIT 18 )
 4    0 ::  MULTIPLY  ALU=A+0 SR[ALU] SR[DLO] JMP=01L ;;
 5    1 ::  MULTIPLY  ALU=A+B SR[ALU] SR[DLO] JMP=01L ;;
 6 ( BIT 19 )
 7    2 ::  MULTIPLY  ALU=A+0 SR[ALU] SR[DLO] JMP=10L ;;
 8    3 ::  MULTIPLY  ALU=A+B SR[ALU] SR[DLO] JMP=10L ;;
 9 ( BIT 20 )
10    4 ::  MULTIPLY  ALU=A+0 SR[ALU] SR[DLO] JMP=11L INC[MPC] ;;
11    5 ::  MULTIPLY  ALU=A+B SR[ALU] SR[DLO] JMP=11L INC[MPC] ;;
12 ( BIT 21 )
13    6 ::  MULTIPLY  ALU=A+0 SR[ALU] SR[DLO] JMP=00L ;;
14    7 ::  MULTIPLY  ALU=A+B SR[ALU] SR[DLO] JMP=00L ;;
15

SCREEN #192
 0 \  MICROCODE --- U*  -- 7
 1 DECIMAL  CROSS-COMPILER
 2 102 CURRENT-OPCODE !
 3 ( BIT 22 )
```

```
 4     0 ::  MULTIPLY  ALU=A+0 SR[ALU] SR[DLO] JMP=01L ;;
 5     1 ::  MULTIPLY  ALU=A+B SR[ALU] SR[DLO] JMP=01L ;;
 6 ( BIT 23 )
 7     2 ::  MULTIPLY  ALU=A+0 SR[ALU] SR[DLO] JMP=10L ;;
 8     3 ::  MULTIPLY  ALU=A+B SR[ALU] SR[DLO] JMP=10L ;;
 9 ( BIT 24 )
10     4 ::  MULTIPLY  ALU=A+0 SR[ALU] SR[DLO] JMP=11L INC[MPC] ;;
11     5 ::  MULTIPLY  ALU=A+B SR[ALU] SR[DLO] JMP=11L INC[MPC] ;;
12 ( BIT 25 )
13     6 ::  MULTIPLY  ALU=A+0 SR[ALU] SR[DLO] JMP=00L ;;
14     7 ::  MULTIPLY  ALU=A+B SR[ALU] SR[DLO] JMP=00L ;;
15


SCREEN #193
 0 \  MICROCODE --- U*  -- 8
 1 DECIMAL  CROSS-COMPILER
 2 103 CURRENT-OPCODE !
 3 ( BIT 26 )
 4     0 ::  MULTIPLY  ALU=A+0 SR[ALU] SR[DLO] JMP=01L ;;
 5     1 ::  MULTIPLY  ALU=A+B SR[ALU] SR[DLO] JMP=01L ;;
 6 ( BIT 27 )
 7     2 ::  MULTIPLY  ALU=A+0 SR[ALU] SR[DLO] JMP=10L ;;
 8     3 ::  MULTIPLY  ALU=A+B SR[ALU] SR[DLO] JMP=10L ;;
 9 ( BIT 28 )
10     4 ::  MULTIPLY  ALU=A+0 SR[ALU] SR[DLO] JMP=11L INC[MPC] ;;
11     5 ::  MULTIPLY  ALU=A+B SR[ALU] SR[DLO] JMP=11L INC[MPC] ;;
12 ( BIT 29 )
13     6 ::  MULTIPLY  ALU=A+0 SR[ALU] SR[DLO] JMP=00L ;;
14     7 ::  MULTIPLY  ALU=A+B SR[ALU] SR[DLO] JMP=00L ;;
15


SCREEN #194
 0 \  MICROCODE --- U*  -- 9
 1 DECIMAL  CROSS-COMPILER
 2 104 CURRENT-OPCODE !
 3 ( BIT 30 )
 4     0 ::  MULTIPLY  ALU=A+0 SR[ALU] SR[DLO] JMP=01L ;;
 5     1 ::  MULTIPLY  ALU=A+B SR[ALU] SR[DLO] JMP=01L ;;
 6 ( BIT 31 )
 7     2 ::  MULTIPLY  ALU=A+0 SR[ALU] SR[DLO] JMP=100 DECODE ;;
 8     3 ::  MULTIPLY  ALU=A+B SR[ALU] SR[DLO] JMP=100 DECODE ;;
 9
10     4 ::  SOURCE=DLO  DEST=DS          END ;;
11
12 ;;END
13
14
15


SCREEN #195
 0 \  MICROCODE --- U/MOD(39/40)
 1 DECIMAL  CROSS-COMPILER
 2 \  Max divisor is 7FFFFFFF  Max dividend is 7FF...FF
 3 105 OPCODE: U/MOD    ( UD1DVDND U2DIVISOR -> UREM UQUOT )
 4 \ ALU LATCH=DIVISOR
```

```
 5 \ DHI=DIVIDEND(HI)  DLO=DIVIDEND(LO)
 6 \ DHI=REMAINDER     DLO=QUOTIENT
 7
 8 \ Shuffle operands around
 9    0 ::  SOURCE=DS   DEST=LATCH ;;
10    1 ::  SOURCE=DHI  DEST=DS    ALU=B  DEST=DHI INC[DP] ;;
11    2 ::  SOURCE=DS   DEST=DLO   DEC[DP]   ;;
12 \ Initial subtraction
13    3 ::  SOURCE=DS   ALU=A-B  DEST=DHI SL[ALU]  SL[DLO] ;;
14 ( *** CONTINUED ON NEXT SCREEN *** )
15

SCREEN #196
 0 \  MICROCODE --- U/MOD  - 2
 1 DECIMAL  CROSS-COMPILER
 2 ( **** Continuation of opcode page # 105 **** )
 3 ( BIT 0 )   4 :: DIVIDE SL[DLO] SL[ALU] ;;
 4 ( BIT 1 )   5 :: DIVIDE SL[DLO] SL[ALU]   INC[MPC] ;;
 5 ( BIT 2 )   6 :: DIVIDE SL[DLO] SL[ALU]   JMP=000 ;;
 6
 7
 8
 9
10
11
12
13
14
15

SCREEN #197
 0 \  MICROCODE --- U/MOD  - 3
 1 DECIMAL  CROSS-COMPILER
 2 106 CURRENT-OPCODE !
 3 ( BIT 3 )   0 :: DIVIDE SL[DLO] SL[ALU] ;;
 4 ( BIT 4 )   1 :: DIVIDE SL[DLO] SL[ALU] ;;
 5 ( BIT 5 )   2 :: DIVIDE SL[DLO] SL[ALU] ;;
 6 ( BIT 6 )   3 :: DIVIDE SL[DLO] SL[ALU] ;;
 7 ( BIT 7 )   4 :: DIVIDE SL[DLO] SL[ALU] ;;
 8 ( BIT 8 )   5 :: DIVIDE SL[DLO] SL[ALU] ;;
 9 ( BIT 9 )   6 :: DIVIDE SL[DLO] SL[ALU]   INC[MPC] ;;
10 ( BIT 10 )   7 :: DIVIDE SL[DLO] SL[ALU]   JMP=000 ;;
11
12
13
14
15

SCREEN #198
 0 \  MICROCODE --- U/MOD  - 4
 1 DECIMAL  CROSS-COMPILER
 2 107 CURRENT-OPCODE !
 3 ( BIT 11 )   0 :: DIVIDE SL[DLO] SL[ALU] ;;
 4 ( BIT 12 )   1 :: DIVIDE SL[DLO] SL[ALU] ;;
 5 ( BIT 13 )   2 :: DIVIDE SL[DLO] SL[ALU] ;;
 6 ( BIT 14 )   3 :: DIVIDE SL[DLO] SL[ALU] ;;
 7 ( BIT 15 )   4 :: DIVIDE SL[DLO] SL[ALU] ;;
 8 ( BIT 16 )   5 :: DIVIDE SL[DLO] SL[ALU] ;;
 9 ( BIT 17 )   6 :: DIVIDE SL[DLO] SL[ALU]   INC[MPC] ;;
10 ( BIT 18 )   7 :: DIVIDE SL[DLO] SL[ALU]   JMP=000 ;;
```

```
11
12
13
14
15

SCREEN #199
 0 \  MICROCODE --- U/MOD  - 5
 1 DECIMAL  CROSS-COMPILER
 2 108 CURRENT-OPCODE !
 3 ( BIT 19 )   0 :: DIVIDE SL[DLO] SL[ALU] ;;
 4 ( BIT 20 )   1 :: DIVIDE SL[DLO] SL[ALU] ;;
 5 ( BIT 21 )   2 :: DIVIDE SL[DLO] SL[ALU] ;;
 6 ( BIT 22 )   3 :: DIVIDE SL[DLO] SL[ALU] ;;
 7 ( BIT 23 )   4 :: DIVIDE SL[DLO] SL[ALU] ;;
 8 ( BIT 24 )   5 :: DIVIDE SL[DLO] SL[ALU] ;;
 9 ( BIT 25 )   6 :: DIVIDE SL[DLO] SL[ALU]   INC[MPC] ;;
10 ( BIT 26 )   7 :: DIVIDE SL[DLO] SL[ALU]   JMP=000 ;;
11
12
13
14
15


SCREEN #200
 0 \  MICROCODE --- U/MOD  - 6
 1 DECIMAL  CROSS-COMPILER
 2 109 CURRENT-OPCODE !
 3 ( BIT 27 )   0 :: DIVIDE SL[DLO] SL[ALU] ;;
 4 ( BIT 28 )   1 :: DIVIDE SL[DLO] SL[ALU] ;;
 5 ( BIT 29 )   2 :: DIVIDE SL[DLO] SL[ALU] ;;
 6 ( BIT 30 )   3 :: DIVIDE SL[DLO] SL[ALU] INC[MPC] ;;
 7 ( BIT 31 )   4 :: DIVIDE SL[DLO]         JMP=000  ;;
 8
 9
10
11
12
13
14
15


SCREEN #201
 0 \  MICROCODE --- U/MOD  - 7
 1 DECIMAL  CROSS-COMPILER
 2 110 CURRENT-OPCODE !
 3 \ Test sign of result
 4    0 ::  SOURCE=DHI        INC[DP]  JMP=01S ;;
 5
 6 \ Result >= 0  Don't add divisor back
 7    2 ::  SOURCE=DHI  DEST=DS  JMP=111  DECODE ;;
 8    7 ::  SOURCE=DLO  ALU=B  DEST=DHI       END ;;
 9
10 \ Result < 0  Add divisor to remainder
11    3 ::  SOURCE=DHI   ALU=A+B  DEST=DHI  JMP=010      ;;
12
13 ;;END
14
15
```

```
SCREEN #202
 0 \  MICROCODE --- WORDSWAP(2)      (ROLL 16 BITS RIGHT)
 1 DECIMAL  CROSS-COMPILER
 2 111 OPCODE: WORDSWAP    ( N1 -> N2 )
 3     0 ::  ALU=A  ROLL[ALU]   DEST=DHI     DECODE ;;
 4     1 ::  ALU=A  ROLL[ALU]   DEST=DHI     END ;;
 5
 6 ;;END
 7
 8
 9
10
11
12
13
14
15

SCREEN #203
 0 \  MICROCODE --- XOR(2)
 1 DECIMAL  CROSS-COMPILER
 2 112 OPCODE: XOR   ( N1 N2 -> N3 )
 3     0 ::  SOURCE=DS  ALU=AxorB  DEST=DHI INC[DP]  DECODE ;;
 4     1 ::                                           END  ;;
 5
 6 ;;END
 7
 8
 9
10
11
12
13
14
15

SCREEN #204
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15

SCREEN #205
 0 \  MICROCODE --- SWAP_DROP (2)
 1 DECIMAL  CROSS-COMPILER
 2 120 OPCODE: SWAP_DROP     ( N1 N2 -> N1 )
 3     0 ::  INC[DP]                          DECODE ;;
 4     1 ::                                   END ;;
 5
```

```
 6 ;;END
 7
 8
 9
10
11
12
13
14
15

SCREEN #206
 0 \  MICROCODE --- <CM-STEP>(10/3)
 1 DECIMAL  CROSS-COMPILER
 2 \ CMOVE step
 3 121 OPCODE: <CM-STEP> ( SRCADDR DESTADDR U -> S+1 D+1 U-1 DONE?)
 4    0 ::  ALU=A-1  DEST=DHI   DEC[DP]    JMP=01Z  ;;
 5 ( Count not = 0 -- continue )
 6    3 ::  SOURCE=DHI  DEST=DS INC[DP]     ALU=0  DEST=DHI ;;
 7    4 ::  INC[DP]  ALU=A+1 ;;
 8    5 ::  SOURCE=DS  DEST=ADDRESS-LATCH   ALU=A+B    DEST=DHI ;;
 9 \ Fetch source byte
10    6 ::  SOURCE=DHI  DEST=DS  DEC[DP]              INC[MPC] ;;
11    7 ::  SOURCE=RAM-BYTE  DEST=DLO   ALU=0 DEST=DHI  JMP=000  ;;
12
13 ( Count = 0 -- stop )
14    2 ::                          DECODE   JMP=001 ;;
15    1 ::  SOURCE=DHI  DEST=DS     ALU=-1  DEST=DHI  END ;;

SCREEN #207
 0 \  MICROCODE --- <CM-STEP> --2
 1 DECIMAL  CROSS-COMPILER
 2 122 CURRENT-OPCODE !
 3 \ Store destination byte
 4    0 ::  ALU=A+1 ;;
 5    1 ::  SOURCE=DS DEST=ADDRESS-LATCH   ALU=A+B DEST=DHI ;;
 6    2 ::  SOURCE=DHI  DEST=DS  DEC[DP]    ;;
 7    3 ::  SOURCE=DLO DEST=RAM-BYTE      ;;
 8    4 ::                                      DECODE ;;
 9    5 ::                          ALU=0  DEST=DHI  END    ;;
10 ;;END
11
12
13
14
15

SCREEN #208
 0 \  MICROCODE --- <<CM-STEP>(11/3)
 1 DECIMAL  CROSS-COMPILER
 2 \ CMOVE step
 3 123 OPCODE: <<CM-STEP> ( SRCAD DSTAD U -> S-1 D-1 U-1 DONE? )
 4    0 ::  ALU=A-1  DEST=DHI  DEC[RP]     JMP=01Z ;;
 5 ( Count not = 0 -- continue )
 6    3 ::  SOURCE=DHI  DEST=RS INC[DP]     ALU=-1  DEST=DHI ;;
 7 \ Fetch source byte
 8    4 ::  SOURCE=DS DEST=ADDRESS-LATCH    ALU=A+B DEST=DHI ;;
 9    5 ::  SOURCE=DHI  DEST=DS  DEC[DP]              INC[MPC] ;;
10    6 ::  SOURCE=RAM-BYTE  DEST=DLO  ALU=-1 DEST=DHI  JMP=000  ;;
11
```

```
12 ( Count = 0 -- stop )
13     2 ::   DEC[DP]       INC[RP]   DECODE    JMP=001 ;;
14     1 ::   SOURCE=DHI    DEST=DS     ALU=-1   DEST=DHI   END ;;
15
```

```
SCREEN #209
 0 \  MICROCODE --- <<CM-STEP> --2
 1 DECIMAL  CROSS-COMPILER
 2 124 CURRENT-OPCODE !
 3 \ Store destination byte
 4     0 ::   SOURCE=DS DEST=ADDRESS-LATCH ALU=A+B DEST=DHI ;;
 5     2 ::   SOURCE=DHI  DEST=DS  DEC[DP]     ;;
 6     1 ::   SOURCE=DLO DEST=RAM-BYTE  ;;
 7     3 ::   SOURCE=RS  DEST=DLO ALU=0  DEST=DHI INC[RP]   DECODE ;;
 8     4 ::   SOURCE=DLO DEST=DS  END ;;
 9 ;;END
10 SPECIAL
11
12
13
14
15
```

```
SCREEN #210
 0 \  MICROCODE --- ORC!(6)
 1 DECIMAL  CROSS-COMPILER
 2 125 OPCODE: ORC!    ( B  ADDR -> )
 3  \ Fetch operand
 4     0 ::   SOURCE=DS  INC[DP]  ;;  \ Load N into bus latc
 5     1 ::   SOURCE=DHI  DEST=ADDRESS-LATCH   ALU=B  DEST=DHI ;;
 6     2 ::   SOURCE=RAM-BYTE   DEST=DLO ;;
 7     3 ::   SOURCE=DLO  ALU=AorB   DEST=DHI ;;
 8     4 ::   SOURCE=DHI  DEST=RAM-BYTE              DECODE  ;;
 9     5 ::   SOURCE=DS  ALU=B  DEST=DHI   INC[DP]    END  ;;
10 ;;END
11
12
13
14
15
```

```
SCREEN #211
 0 \  MICROCODE --- 4*(2)
 1 DECIMAL  CROSS-COMPILER
 2 126 OPCODE: 4*    ( N -> N+N )
 3 \ Get a 0 into highest bit of DLO for shift operation
 4     0 ::   ALU=A+A  SR[DLO] DEST=DHI       DECODE  ;;
 5     1 ::   ALU=A    SL[ALU] DEST=DHI       END     ;;
 6 ;;END
 7
 8
 9
10
11
12
13
14
15
```

```
SCREEN #212
 0 \  MICROCODE --- <INTERRUPT>(11)  Interrupt service word
 1 DECIMAL  CROSS-COMPILER    \ MUST be op-code 1 !!!!!!
 2 1 OPCODE: <INTERRUPT>
 3     ( -> PAGE&ADDR-CNT INT-FLAGS )
 4     0 ::  ;;
 5     1 ::  SOURCE=ADDRESS-COUNTER  DEST=DLO DEC[DP] ;;
 6     2 ::  SOURCE=DHI DEST=DS   DEC[DP] ;;
 7     3 ::  SOURCE=DLO    DEST=DS   ALU=0  DEST=DHI  ;;
 8     4 ::  SOURCE=DHI  DEST=ADDRESS-COUNTER    ;;
 9 \ Place -1 in flags to mask interrupts
10     5 ::  SOURCE=FLAGS    ALU=-1  DEST=DHI          INC[MPC] ;;
11     6 ::  SOURCE=DHI  DEST=FLAGS  ALU=B DEST=DHI    JMP=000 ;;
12
13
14
15

SCREEN #213
 0 \  MICROCODE --- <INTERRUPT> --2
 1 DECIMAL  CROSS-COMPILER
 2 2 CURRENT-OPCODE !
 3 \ Execute program at address 0
 4     0 :: SOURCE=RAM  DEST=DECODE DECODE ;;
 5     1 ::   END ;;
 6
 7 ;;END
 8 POISON  SPECIAL
 9
10
11
12
13
14
15

SCREEN #214
 0 \  MICROCODE --- SET-FLAGS (6)
 1 DECIMAL  CROSS-COMPILER
 2 127 OPCODE: SET-FLAGS    ( NEW-FLAG-WORD -> OLD-FLAG-WORD )
 3     0 ::  ;;
 4     1 ::  SOURCE=FLAGS  DEST=DLO ;;
 5     2 ::  SOURCE=DHI  DEST=FLAGS    ;;
 6     3 ::  SOURCE=DLO                ALU=B  DEST=DHI ;;
 7     4 ::  DECODE ;;
 8     5 ::  END ;;
 9 ;;END
10
11
12
13
14
15

SCREEN #215
 0 \  MICROCODE --- RTI(6)  Return from interrupt
 1 DECIMAL  CROSS-COMPILER
 2 128 OPCODE: RTI
 3     ( PAGE&ADDR-CNT INT-FLAGS  -> )
 4     0 ::  SOURCE=DHI  DEST=FLAGS ;;
 5     1 ::  SOURCE=DS  DEST=PAGE   ;;
```

```
 5       134  ABS  134 = NOT ABORT" ABSA"
 6      -134  ABS  134 = NOT ABORT" ABSB"
 7    #LOOP  ." ."  ?TERMINAL ABORT" BREAK"    #LOOP
 8    CR . . . . .  CR ;
 9
10 X-ABS
11
12
13
14
15


SCR #70
 0 \ MICROCODE TESTING  CMOVE
 1 DECIMAL
 2 CREATE TEXTD  10 ALLOT    CREATE TEXTE 10 ALLOT
 3 : X-CMOVE
 4    1 2 3 4 5
 5    1111 TEXTE !  2222 TEXTE 1+ !
 6     REPS #DO   3000 #DO
 7        0 TEXTD !  0 TEXTD 1+ !  0 TEXTD 2+ !  0 TEXTD 3 + !
 8        TEXTE  TEXTD 1+  2 CMOVE
 9        TEXTD  @  ABORT" CMOVEA"   TEXTD 3 + @ ABORT" CMOVEB"
10        TEXTD 1+ @  1111 = NOT ABORT" CMOVEC"
11        TEXTD 2+ @  2222 = NOT ABORT" CMOVED"
12    #LOOP  ." ."  ?TERMINAL ABORT" BREAK"    #LOOP
13    CR . . . . .  CR ;
14 X-CMOVE
15


SCR #71
 0 \ MICROCODE TESTING  2/
 1 DECIMAL
 2 : X-2/
 3    1 2 3 4 5
 4     REPS #DO     -1 #DO
 5        5  2/  2 = NOT ABORT" 2/A"
 6       -5  2/  -2 = NOT ABORT" 2/B"
 7       -6  2/  -3 = NOT ABORT" 2/C"
 8       -1  2/          ABORT" 2/D"
 9    #LOOP  ." ."  ?TERMINAL ABORT" BREAK"    #LOOP
10    CR . . . . .  CR ;
11
12 X-2/
13
14
15


SCR #72
 0 \ MICROCODE TESTING   @  !
 1 DECIMAL
 2 VARIABLE VARXA
 3 : X-@!
 4    1 2 3 4 5
 5     REPS #DO     -1 #DO
 6        -1 VARXA !   VARXA @  -1 = NOT ABORT" !@A"
 7         0 VARXA !   VARXA @        ABORT" !@B"
 8    #LOOP  ." ."  ?TERMINAL ABORT" BREAK"    #LOOP
 9    CR . . . . .  CR ;
10
```

```
12
13
14
15

SCREEN #219
 0 \  MICROCODE --- COUNT-DOWN (3/4)     Favors taking the branch
 1 DECIMAL  CROSS-COMPILER  \ Similar to DUP_1-_SWAP_0=_OBRANCH
 2 138 OPCODE: <COUNT-DOWN> ( FLAG -> ../FLAG ) \ BUT drops at end!
 3 \  0 flag => branch     non-0 flag => go to next instruction
 4    \ Must ALWAYS be used with a CALL attribute!!!!
 5    0 ::  SOURCE=ADDRESS-COUNTER DEST=DLO  JMP=01Z ;;
 6 \ FLAG=0     Allow compiled branch to take affect
 7    3 ::  ALU=A-1  DEST=DHI    INC[RP]    DECODE ;;
 8    4 ::  END ;;
 9 \ FLAG<>0    Override compiled branch
10    2 ::  SOURCE=DLO  DEST=ADDRESS-COUNTER JMP=101 ;;
11               \ Wait for RAM access
12    5 ::  SOURCE=RAM  DEST=DECODE    INC[RP] DECODE ;;
13    6 ::  SOURCE=DS  ALU=B  DEST=DHI  INC[DP]  END ;;
14 ;;END   POISON SPECIAL
15


SCREEN #220
 0 \  MICROCODE --- @+(5)
 1 DECIMAL  CROSS-COMPILER
 2 139 OPCODE: @+    ( N1 ADDR -> N2 )
 3    0 ::  SOURCE=DS  ;;
 4    1 ::  SOURCE=DHI   DEST=ADDRESS-LATCH     ALU=B DEST=DHI ;;
 5    2 ::  SOURCE=RAM  DEST=DLO     ;;
 6    3 ::  SOURCE=DLO  ALU=A+B  DEST=DHI  INC[DP] DECODE     ;;
 7    4 ::  END ;;
 8
 9 ;;END
10
11
12
13
14
15


SCREEN #221
 0 \  MICROCODE --- 3_PICK (4)
 1 DECIMAL  CROSS-COMPILER
 2 140 OPCODE: 3_PICK  ( N1 N2 N3 N4 -> N1 N2 N3 N4  N2 )
 3    0 ::  SOURCE=DHI  DEST=DLO          INC[DP] ;;
 4    1 ::  SOURCE=DS   ALU=B  DEST=DHI   DEC[DP] ;;
 5    2 ::                DEC[DP]                    DECODE ;;
 6    3 ::  SOURCE=DLO  DEST=DS                      END    ;;
 7
 8 ;;END
 9
10
11
12
13
14
15
```

```
SCREEN #222
 0 \  MICROCODE --- 4_PICK (6)
 1 DECIMAL  CROSS-COMPILER
 2 141 OPCODE: 4_PICK  ( N1 N2 N3 N4 -> N1 N2 N3 N4  N1 )
 3    0 ::  SOURCE=DHI  DEST=DLO          INC[DP] ;;
 4    1 ::   INC[DP] ;;                    '
 5    2 ::  SOURCE=DS   ALU=B  DEST=DHI   DEC[DP] ;;
 6    3 ::                 DEC[DP]      ;;
 7    4 ::                 DEC[DP]                  DECODE ;;
 8    5 ::  SOURCE=DLO  DEST=DS                     END    ;;
 9
10 ;;END
11
12
13
14
15

SCREEN #223
 0 \  MICROCODE --- D>R(4)
 1 DECIMAL  CROSS-COMPILER
 2 142 OPCODE: D>R    ( N1 N2  -> )   ( Return: -> N2 N1 )
 3    0 ::  DEC[RP] ;;
 4    1 ::  SOURCE=DHI  DEST=RS  DEC[RP] ;;
 5    2 ::  SOURCE=DS   DEST=RS  INC[DP]  DECODE ;;
 6    3 ::  SOURCE=DS   ALU=B  DEST=DHI   INC[DP]  END ;;
 7
 8 ;;END
 9 POISON SPECIAL
10
11
12
13
14
15

SCREEN #224
 0 \  MICROCODE --- DR>(5)
 1 DECIMAL  CROSS-COMPILER
 2 143 OPCODE: DR>    ( -> N1 N2 )   ( return: N2 N1 -> )
 3    0 ::  DEC[DP] ;;
 4    1 ::  SOURCE=DHI  DEST=DS  DEC[DP] ;;
 5    2 ::  SOURCE=RS  DEST=DS            INC[RP]  ;;
 6    3 ::  SOURCE=RS  DEST=DLO           INC[RP]  DECODE ;;
 7    4 ::  SOURCE=DLO  ALU=B  DEST=DHI            END  ;;
 8
 9 ;;END
10 POISON SPECIAL
11
12
13
14
15

SCREEN #225
 0 \  MICROCODE --- DROT (13)
 1 DECIMAL  CROSS-COMPILER
 2 144 OPCODE: DROT  ( D1 D2 D3 -> D2 D3 D1 )
 3    0 ::  ;;
 4    1 ::  DEC[RP] ;;
```

```
 5    \ Transfer D3 to RS
 6     2 ::  SOURCE=DHI  DEST=RS  DEC[RP] ;;
 7     3 ::  SOURCE=DS   DEST=RS   DEC[RP]  INC[DP] ;;
 8    \ Transfer D2 to RS
 9     4 ::  SOURCE=DS   DEST=RS   DEC[RP]  INC[DP] ;;
10     5 ::  SOURCE=DS   DEST=RS            INC[DP] ;;
11    \ Transfer D1 to DHI:DLO
12     6 ::  SOURCE=DS   ALU=B DEST=DHI     INC[DP] INC[MPC] ;;
13     7 ::  SOURCE=DS   DEST=DLO                   JMP=000 ;;
14
15


SCREEN #226
 0 \  MICROCODE --- DROT --2
 1 DECIMAL  CROSS-COMPILER
 2 145 CURRENT-OPCODE !
 3    \ Transfer D2 from RS to DS
 4     0 ::  SOURCE=RS   DEST=DS   INC[RP]  DEC[DP] ;;
 5     1 ::  SOURCE=RS   DEST=DS   INC[RP]  DEC[DP] ;;
 6    \ Transfer D3 from RS to DS
 7     2 ::  SOURCE=RS   DEST=DS   INC[RP]  DEC[DP] ;;
 8     3 ::  SOURCE=RS   DEST=DS   INC[RP]  DEC[DP]  DECODE ;;
 9    \ Transfer D1(LO)  from DLO to DS
10     4 ::  SOURCE=DLO  DEST=DS   END ;;
11
12 ;;END
13
14
15


SCREEN #227
 0 \  MICROCODE --- DOVER (7)
 1 DECIMAL  CROSS-COMPILER
 2 146 OPCODE: DOVER  ( D1 D2 -> D1 D2 D1 )
 3     0 ::  INC[DP] ;;
 4 \ Save D1 in RS:DLO
 5     1 ::  DEC[RP] INC[DP]  ;;
 6     2 ::  SOURCE=DS  DEST=DLO           DEC[DP] ;;
 7     3 ::  SOURCE=DS  DEST=RS            DEC[DP] ;;
 8 \ Transfer D2hi and D1lo to DS
 9     4 ::  SOURCE=RS    DEST=LATCH  DEC[DP] INC[RP] ;;
10     5 ::  SOURCE=DHI  DEST=DS  ALU=B DEST=DHI   DEC[DP] DECODE ;;
11     6 ::  SOURCE=DLO   DEST=DS                          END ;;
12
13 ;;END
14
15


SCREEN #228
 0 \  MICROCODE --- D= (5/6)
 1 DECIMAL  CROSS-COMPILER
 2 147 OPCODE: D=   ( D1 D2 -> =FLAG )
 3 \ Save D2(lo) in DLO
 4     0 ::  SOURCE=DS   DEST=DLO    INC[DP] ;;
 5 \ Compare hi halves
 6     1 ::  SOURCE=DS   ALU=AxorB  DEST=DHI    INC[DP] ;;
 7     2 ::  SOURCE=DS   ALU=B  DEST=DHI   INC[DP]   JMP=10Z ;;
 8 \ Not =
 9     5 ::  DECODE JMP=111  ;;
10     7 ::  ALU=0  DEST=DHI  END ;;
```

```
11 \ Hi halves =
12     4 ::  SOURCE=DLO    ALU=AxorB  DEST=DHI   JMP=011 ;;
13     3 ::  JMP=11Z  DECODE ;;
14     6 ::  ALU=-1 DEST=DHI   END ;;     ( Results = )
15 ;;END

SCREEN #229
 0 \  MICROCODE --- <UNORM> (6+4*X)
 1 DECIMAL  CROSS-COMPILER
 2 148 OPCODE: <UNORM>    ( MANT EXP -> MANT EXP )
 3     0 ::  SOURCE=DS  DEST=DLO
 4             ALU=A+1    DEST=DHI     JMP=010 ;;
 5 \ Loop for justifying.  DHI = 0 after shift while in loop
 6     2 ::  ALU=A-1  DEST=DHI     JMP=100 ;;
 7     4 ::  SOURCE=DHI  DEST=DS
 8                ALU=0  CIN=0  SL[DLO]  SL[ALU] DEST=DHI ;;
 9     5 ::  ALU=A  DEST=DHI ;;  ( Allow Z bit to be set )
10     6 ::  SOURCE=DS    ALU=B     DEST=DHI     JMP=01Z ;;
11
12 \ Un-shift two bits
13     3 ::  ALU=-1   SR[DLO]  JMP=111    INC[MPC] ;;
14     7 ::  ALU=0    SR[DLO]             JMP=000  ;;
15


SCREEN #230
 0 \  MICROCODE --- <UNORM> --2
 1 DECIMAL  CROSS-COMPILER
 2 149 CURRENT-OPCODE !
 3 \ Finish up at end
 4     0 ::  SOURCE=DS    ALU=B  DEST=DHI                    ;;
 5     1 ::  SOURCE=DLO  DEST=DS    ALU=A+1 DEST=DHI   DECODE ;;
 6     2 ::  END ;;
 7 ;;END
 8
 9
10
11
12
13
14
15


SCREEN #231
 0 \  MICROCODE --- <UDNORM>
 1 DECIMAL  CROSS-COMPILER
 2 150 OPCODE: <UDNORM>    ( DMANT EXP -> DMANT EXP )
 3 \ SETUP:  DLO = DMANTLO  DHI = DMANTHI  DS = EXP
 4 \ Use RS as temp holding location for DMANTHI
 5     0 ::  SOURCE=DS  DEST=LATCH    ALU=A+1  DEST=DHI ;;
 6     1 ::  SOURCE=DHI  DEST=DS      INC[DP]     DEC[RP]
 7              SOURCE=LATCH  ALU=B  DEST=DHI      ;;
 8     2 ::  SOURCE=DS  DEST=DLO      DEC[DP]     INC[MPC] ;;
 9     3 ::  SOURCE=DHI  DEST=RS                  JMP=00S ;;
10
11
12
13
14
15
```

```
SCREEN #232
 0 \  MICROCODE --- <UDNORM>  --2
 1 DECIMAL  CROSS-COMPILER
 2 151 CURRENT-OPCODE !
 3 \ SETUP:  DLO = DMANTLO  DHI = RS = DMANTHI  DS = EXP
 4 \ Sign bit zero -- do another shift
 5    0 ::  SOURCE=DS  DEST=LATCH
 6            ALU=A  CIN=0 SL[DLO]  SL[ALU]  DEST=DHI JMP=100  ;;
 7    4 ::  SOURCE=DHI  DEST=RS    SOURCE=LATCH  ALU=B DEST=DHI ;;
 8    5 ::  SOURCE=RS  DEST=LATCH  ALU=A-1  DEST=DHI  ;;
 9    6 ::  SOURCE=DHI DEST=DS
10                SOURCE=LATCH ALU=B DEST=DHI ;;
11    7 ::  SOURCE=RS  DEST=LATCH       JMP=005  ;;
12
13
14
15

SCREEN #233
 0 \  MICROCODE --- <UDNORM>  --3
 1 DECIMAL  CROSS-COMPILER
 2 151 CURRENT-OPCODE !
 3 \ Done -- shift right 1 bit and end
 4    1 ::  SOURCE=LATCH ALU=B  SR[ALU]  SR[DLO]  CIN=0 DEST=DHI
 5            SOURCE=DHI   INC[DP] INC[RP]        INC[MPC]  ;;
 6    2 ::  SOURCE=DLO DEST=DS   DEC[DP]    JMP=000   ;;
 7
 8 152 CURRENT-OPCODE !
 9    0 ::  SOURCE=DS  DEST=LATCH                DECODE ;;
10    1 ::  SOURCE=DHI  DEST=DS
11            SOURCE=LATCH  ALU=B  DEST=DHI     END ;;
12 ;;END
13
14
15

SCREEN #234
 0 \  MICROCODE --- C+!(6)
 1 DECIMAL  CROSS-COMPILER
 2 153 OPCODE: C+!    ( N  ADDR -> )
 3  \ Fetch operand
 4    0 ::  SOURCE=DS  INC[DP]  ;;  \ Load N into bus latch
 5    1 ::  SOURCE=DHI  DEST=ADDRESS-LATCH   ALU=B  DEST=DHI ;;
 6    2 ::  SOURCE=RAM-BYTE   DEST=DLO ;;
 7    3 ::  SOURCE=DLO  ALU=A+B   DEST=DHI ;;
 8    4 ::  SOURCE=DHI  DEST=RAM-BYTE              DECODE  ;;
 9    5 ::  SOURCE=DS  ALU=B  DEST=DHI   INC[DP]    END  ;;
10 ;;END
11
12
13
14
15

SCREEN #235
 0 \  MICROCODE --- =0_?EXIT_0 (3/6)  Expert system support
 1 DECIMAL  CROSS-COMPILER
 2 154 OPCODE: =0_?EXIT_0   ( TRUE-FLAG -> )      \ Continue
 3                          ( 0    -> 0 )        \ Exit
 4
```

```
 5 \  0 flag => EXIT   non-0 flag => go to next instruction
 6 \ Must ALWAYS be used with a CALL attribute!!!!!
 7    0 ::  JMP=01Z ;;
 8
 9 \ FLAG<>0   Allow compiled subroutine call to take effect
10    3 ::                              JMP=001  DECODE ;;
11    1 ::  SOURCE=DS  ALU=B  DEST=DHI  INC[DP]  END ;;
12
13
14
15

SCREEN #236
 0 \  MICROCODE --- =0_?EXIT_0  --2
 1 DECIMAL  CROSS-COMPILER
 2 \ 154 CURRENT-OPCODE !
 3 \ FLAG=0   Override compiled CALL plus do an EXIT
 4    2 ::  SOURCE=RS INC[RP]  JMP=100   ;;   \ Get 2nd return addr
 5    4 ::  SOURCE=RS  DEST=ADDRESS-COUNTER ;; ( Also sets latch )
 6               \ Wait for RAM access
 7    5 ::  ;;
 8    6 ::  SOURCE=RAM  DEST=DECODE   INC[RP] DECODE ;;
 9    7 ::  ALU=0  DEST=DHI  END ;;
10
11 ;;END    POISON
12
13
14
15

SCREEN #237
 0 \  MICROCODE --- <>0_?EXIT_0 (3/6)  Expert system support
 1 DECIMAL  CROSS-COMPILER
 2 155 OPCODE: <>0_?EXIT_0   ( TRUE-FLAG -> )      \ Continue
 3                           ( 0    -> 0 )        \ Exit
 4
 5 \  0 flag => EXIT   non-0 flag => go to next instruction
 6 \ Must ALWAYS be used with a CALL attribute!!!!!
 7    0 ::  JMP=01Z ;;
 8
 9 \ FLAG=0    Allow compiled subroutine call to take effect
10    2 ::                              JMP=001  DECODE ;;
11    1 ::  SOURCE=DS  ALU=B  DEST=DHI  INC[DP]  END ;;
12
13
14
15

SCREEN #238
 0 \  MICROCODE --- <>0_?EXIT_0  --2
 1 DECIMAL  CROSS-COMPILER
 2 \ 155 CURRENT-OPCODE !
 3 \ FLAG<>0   Override compiled CALL plus do an EXIT
 4    3 ::  SOURCE=RS  INC[RP]   ;;   \ Get 2nd return address
 5    4 ::  SOURCE=RS  DEST=ADDRESS-COUNTER ;; ( Also sets latch )
 6               \ Wait for RAM access
 7    5 ::  ;;
 8    6 ::  SOURCE=RAM  DEST=DECODE   INC[RP] DECODE ;;
 9    7 ::  ALU=0  DEST=DHI  END ;;
10
```

```
11 ;;END     POISON
12
13
14
15

SCREEN #239
 0 \  MICROCODE --- <>0_?EXIT (3/6)  Expert system support
 1 DECIMAL  CROSS-COMPILER
 2 156 OPCODE: <>0_?EXIT      ( FALSE-FLAG -> )       \ Continue
 3                            ( TRUE-FLAG ->  )       \ Exit
 4
 5 \  non-0 flag => EXIT   0 flag => go to next instruction
 6 \ Must ALWAYS be used with a CALL attribute!!!!!
 7    0 ::  JMP=01Z ;;
 8
 9 \ FLAG=0     Allow compiled subroutine call to take effect
10    2 ::                              JMP=001  DECODE ;;
11    1 ::  SOURCE=DS  ALU=B  DEST=DHI  INC[DP]  END ;;
12
13
14
15


SCREEN #240
 0 \  MICROCODE --- <>0_?EXIT  --2
 1 DECIMAL  CROSS-COMPILER
 2 \ 156 CURRENT-OPCODE !
 3 \ FLAG=0   Override compiled CALL plus do an EXIT
 4    3 ::  SOURCE=RS  INC[RP]   JMP=100 ;;   \ Get 2nd return addr
 5    4 ::  SOURCE=RS  DEST=ADDRESS-COUNTER ;; ( Also sets latch )
 6               \ Wait for RAM access
 7    5 ::  ;;
 8    6 ::  SOURCE=RAM  DEST=DECODE   INC[RP] DECODE ;;
 9    7 ::  SOURCE=DS  ALU=B  DEST=DHI  INC[DP]  END ;;
10
11 ;;END     POISON
12
13
14
15


SCREEN #241
 0 \  MICROCODE --- -1(2)
 1 DECIMAL  CROSS-COMPILER        \  ( Used by xcompiler!)
 2 157 OPCODE: -1  ( -> TRUE-FLAG )
 3    0 ::  DEC[DP]  SOURCE=DHI  DEST=DLO
 4                                        DECODE ;;
 5    1 ::  SOURCE=DLO DEST=DS
 6                ALU=-1   DEST=DHI    END    ;;
 7 ;;END
 8
 9
10
11
12
13
14
15
```

```
SCREEN #242
 0 \  MICROCODE --- 0=_NOT(2)          \ Forces clean flag value
 1 DECIMAL  CROSS-COMPILER
 2 158 OPCODE: 0=_NOT    ( N -> 0/-1 )
 3    0 ::  JMP=01Z                      DECODE  ;;
 4
 5 ( <>0 )  3 ::    ALU=-1  DEST=DHI    END  ;;
 6 (  =0 )  2 ::    ALU=0   DEST=DHI    END  ;;
 7 ;;END
 8
 9
10
11
12
13
14
15


SCREEN #243
 0 \  MICROCODE --- SET-TRUE(8)    \ Expert system support
 1 DECIMAL  CROSS-COMPILER     \ Compile as a normal opcode
 2 159 OPCODE: SET-TRUE    ( -> )
 3 \ In-line parameter is pointer to flag location.  Flag <-  1
 4 \  Also performs an exit
 5    0 ::  DEC[DP]   SOURCE=RAM  DEST=DLO     ;;
 6    1 ::  SOURCE=DHI  DEST=DS     ALU=0  DEST=DHI    ;;
 7    2 ::  SOURCE=DLO  DEST=ADDRESS-COUNTER   ALU=A+1 DEST=DHI ;;
 8    3 ::  SOURCE=DHI  DEST=RAM ;;
 9    4 ::  SOURCE=RS  DEST=DLO ;;
10    5 ::  SOURCE=DLO DEST=ADDRESS-COUNTER  INC[RP] ;;
11    6 ::  SOURCE=RAM  DEST=DECODE             DECODE ;;
12    7 ::  SOURCE=DS   ALU=B  DEST=DHI    INC[DP]  END ;;
13 ;;END
14 POISON SPECIAL
15

SCREEN #244
 0 \  MICROCODE --- SET-FALSE(8)   \ Expert system support
 1 DECIMAL  CROSS-COMPILER     \ Compile as a normal opcode
 2 160 OPCODE: SET-FALSE   ( -> )
 3 \ In-line parameter is pointer to flag location.  Flag <- 0
 4 \  Also performs an exit
 5    0 ::  DEC[DP]   SOURCE=RAM  DEST=DLO     ;;
 6    1 ::  SOURCE=DLO  DEST=ADDRESS-COUNTER   ;;
 7    2 ::  SOURCE=DHI  DEST=DS     ALU=0   DEST=DHI    ;;
 8    3 ::  SOURCE=DHI  DEST=RAM ;;
 9    4 ::  SOURCE=RS  DEST=DLO ;;
10    5 ::  SOURCE=DLO DEST=ADDRESS-COUNTER  INC[RP] ;;
11    6 ::  SOURCE=RAM  DEST=DECODE    DECODE ;;
12    7 ::  SOURCE=DS   ALU=B  DEST=DHI    INC[DP]  END ;;
13 ;;END
14 POISON SPECIAL
15

SCREEN #245
 0 \  MICROCODE --- EXPERT-B (7)   \ Expert system support
 1 DECIMAL  CROSS-COMPILER     \ Fetch flag value
 2 \ Compile with a jump to the flag address
 3 161 OPCODE: EXPERT-B    ( -> FLAG-VALUE )
```

```
 4 \ In-line flag value fetched as target of jump.
 5 \  Also performs an exit
 6    0 ::  SOURCE=RAM  DEST=DLO    DEC[DP]  ;;
 7    1 ::  SOURCE=DHI  DEST=DS  ;;
 8    2 ::  SOURCE=DLO  ALU=B  DEST=DHI ;;
 9    3 ::  SOURCE=RS  DEST=DLO ;;
10    4 ::  SOURCE=DLO DEST=ADDRESS-COUNTER  INC[RP] ;;
11    5 ::  SOURCE=RAM  DEST=DECODE                  DECODE  ;;
12    6 ::                                           END ;;
13 ;;END
14 POISON
15
```

```
SCREEN #246
 0 \  MICROCODE --- EXPERT-C (6)  \ Expert system support
 1 DECIMAL  CROSS-COMPILER    \ Store flag value
 2 \ Compile with a call to the flag address
 3 162 OPCODE: EXPERT-C   ( FLAG-VALUE -> )
 4 \ In-line flag value stored as target of jump.
 5 \  Also performs an exit
 6    0 ::  SOURCE=ADDRESS-COUNTER DEST=ADDRESS-LATCH ;;
 7    1 ::  SOURCE=DHI  DEST=RAM   INC[RP]     ;;
 8    2 ::  SOURCE=RS  DEST=DLO ;;
 9    3 ::  SOURCE=DLO DEST=ADDRESS-COUNTER  INC[RP] ;;
10    4 ::  SOURCE=RAM  DEST=DECODE  DECODE    ;;
11    5 ::  SOURCE=DS   ALU=B  DEST=DHI  INC[DP]   END ;;
12 ;;END
13 POISON
14
15
```

```
SCREEN #247
 0 \  MICROCODE --- EXPERT-A (6/8) \ Expert system support
 1 \   5 clocks if unknown,  7 clocks if known
 2 DECIMAL  CROSS-COMPILER
 3 \ Compile with a CALL to the rule list
 4 163 OPCODE: EXPERT-A   ( -> FLAG-VALUE )  \ If value valid
 5  ( -> )  \ if value invalid -- perform call to rule list
 6 \ Fetch flag value and test for =0 or =-1
 7    0 ::  SOURCE=RAM  DEST=DLO  DEC[DP] ;;  \ Get value
 8    1 ::  SOURCE=DHI  DEST=DS ;;
 9    2 ::  SOURCE=DLO  ALU=B DEST=DHI
10       \ Increment address counter to point to 1st list cell
11            INC[ADC]    INC[RP]  INC[MPC] ;;
12
13 \ test for flag = -1
14    3 ::  SOURCE=ADDRESS-COUNTER  DEST=ADDRESS-LATCH JMP=OOS ;;
15
```

```
SCREEN #248
 0 \  MICROCODE --- EXPERT-A  --2  \ Expert system support
 1 DECIMAL  CROSS-COMPILER
 2 164 CURRENT-OPCODE !
 3 \ Flag is valid - true or false.  Perform an exit
 4    0 ::  SOURCE=RS  DEST=ADDRESS-COUNTER          JMP=100 ;;
 5    4 ::  INC[RP]                                  ;;
 6    5 ::  SOURCE=RAM  DEST=DECODE  DECODE ;;
 7    6 ::  END ;;
 8 \ Flag = -1  -- Unknown value
 9 \ Access rule list addr  s
```

```
10     1 ::  SOURCE=RAM    DEST=DECODE     DEC[RP] ' DECODE ;;
11 \ Flag invalid value -- allow call to take place
12 \ POP last value from DS
13     2 ::  SOURCE=DS  ALU=B  DEST=DHI   INC[DP]    END ;;
14
15 ;;END     POISON

SCREEN #249
 0 \  MICROCODE --- UNKNOWN_I_@_12_+_!_4   Expert system initing
 1 DECIMAL  CROSS-COMPILER
 2 165 OPCODE: UNKNOWN_I_@_12_+_!_4  ( -> 4 )
 3     0 ::  DEC[DP] ;;
 4     1 ::  SOURCE=RS  DEST=ADDRESS-LATCH ;;
 5     2 ::  SOURCE=ADDRESS-COUNTER  DEST=DS    ;;
 6     3 ::  SOURCE=RAM  DEST=DLO ;;
 7     4 ::  SOURCE=DLO  DEST=ADDRESS-COUNTER ;;
 8     5 ::  SOURCE=DHI  DEST=DLO  INC[ADC] ;;
 9     6 ::   INC[ADC]                    INC[MPC] ;;
10     7 ::   INC[ADC]  ALU=-1  DEST=DHI   JMP=000 ;;
11
12
13
14
15

SCREEN #250
 0 \  MICROCODE --- UNKNOWN_I_@_12_+_!_4   -- 2
 1 DECIMAL  CROSS-COMPILER
 2 166 CURRENT-OPCODE !
 3     0 ::  SOURCE=ADDRESS-COUNTER   DEST=ADDRESS-LATCH ;;
 4     1 ::  SOURCE=DHI  DEST=RAM    ALU=0  DEST=DHI    ;;
 5     2 ::  SOURCE=DS   DEST=ADDRESS-COUNTER   ALU=A+1 ;;
 6     3 ::              ALU=A+A    DEST=DHI   ;;
 7     4 ::     ALU=A+A  DEST=DHI           DECODE ;;
 8     5 ::   SOURCE=DLO  DEST=DS   END ;;
 9 ;;END
10 POISON
11
12
13
14
15

SCREEN #251
 0 \  MICROCODE --- <LOOP>(10/12)
 1 DECIMAL  CROSS-COMPILER
 2 \ Always use with CALL attribute!!!!!
 3 \ IMPLICIT LOOP RANGE SPAN OF 80000000 Hex
 4 167 OPCODE: <LOOP>   ( -> )    ( RS: LIMIT COUNT -> ... )
 5 \ Load address latch for destination of end-of-loop fall-through
 6     0 ::  DEC[DP]  ;;
 7     1 ::  SOURCE=DHI DEST=DS   ALU=0  DEST=DHI INC[RP] ;;
 8     2 ::  SOURCE=RS DEST=DLO   ALU=A+1  DEST=DHI  ;;
 9     3 ::  SOURCE=DLO  ALU=A+B  DEST=DHI  ;;  \ Add 1 to counter
10     4 ::  SOURCE=DHI  DEST=RS INC[RP] ;;
11     5 ::  SOURCE=RS   DEST=DLO    DEC[RP]   ;;
12     6 ::  SOURCE=DLO ALU=A-B   DEC[RP] INC[MPC] ;; \ Test count
13     7 ::  SOURCE=RS  DEST=DLO  INC[RP]  JMP=005 ;;
14
```

```
SCREEN #252
 0 \  MICROCODE --- <LOOP>  -- 2
 1 DECIMAL  CROSS-COMPILER
 2 168 CURRENT-OPCODE !
 3 ( LOOP  ) 1 ::  DECODE JMP=100 ;;
 4
 5 ( DONE ) 0 :: SOURCE=DLO DEST=ADDRESS-COUNTER INC[RP] JMP=010 ;;
 6           2 :: SOURCE=RAM  DEST=DECODE  INC[RP]
 7                           DECODE  JMP=100 ;;
 8
 9    4 ::  SOURCE=DS   ALU=B  DEST=DHI  INC[DP]     END ;;
10
11 ;;END
12 POISON SPECIAL
13
14
15



SCREEN #253
 0 \  MICROCODE --- D+(7)  --1
 1 DECIMAL  CROSS-COMPILER
 2 169 OPCODE: D+     ( D1 D2 -> D3 )
 3    0 ::  SOURCE=DS   DEST=LATCH    \ Place D2(lo) in ALU latch
 4                              INC[DP]  ;;
 5 \  Save D2(hi) in DLO register , Place D2(lo) in DHI register
 6    1 ::  SOURCE=DHI  DEST=DLO    INC[DP]
 7          SOURCE=LATCH  ALU=B  DEST=DHI  ;;
 8 \ Compute low half of result
 9    2 ::  SOURCE=DS  ALU=A+B  DEST=DHI     INC[MPC] ;;
10    3 ::  SOURCE=DHI  DEST=DS   DEC[DP]    JMP=10C   ;;
11
12
13
14
15



SCREEN #254
 0 \  MICROCODE --- D+ --2
 1 DECIMAL  CROSS-COMPILER
 2 170 CURRENT-OPCODE !
 3 ( CY<>0 ) 4 :: SOURCE=DS  ALU=B  DEST=DHI     JMP=000 ;;
 4           0 :: SOURCE=DLO ALU=A+B+1  DEST=DHI INC[DP] DECODE ;;
 5           1 ::  END ;;
 6
 7 ( CY=0 )  5 :: SOURCE=DS  ALU=B  DEST=DHI     ;;
 8           6 :: SOURCE=DLO ALU=A+B    DEST=DHI INC[DP] DECODE ;;
 9           7 ::  END ;;
10
11 ;;END
12
13
14
15
```

APPENDIX A

PART II

MVP-FORTH/32 CROSS-COMPILED KERNEL

PROGRAM LISTING

WISC CPU/32 (C) 1987 BY PHIL KOOPMAN JR.

```
SCREEN #1
 0 INDEX --- CPU/32  KERNEL & MATH SOURCE          PHIL KOOPMAN JR.
 1 LIBRARY VERSION    FILE: KERNEL.4TH          LAST UPDATE: 6/1/87
 2
 3 BETA TEST VERSION  (C) COPYRIGHT 1987
 4 BY PHIL KOOPMAN JR.
 5
 6 LOAD      SOURCE
 7 SCREEN    SCREENS     CONTENTS
 8 =======   ========    ========
 9 2 LOAD    4 - 76      CROSS-COMPILED KERNEL SOURCE
10 3 LOAD    81 - 105    COMPLETED KERNEL SOURCE FOR CPU/32 LOADING
11    ...    110 - 141   64-BIT INTEGER MATH PACKAGE
12    ...    143 - 199   64-BIT INTEGER MATH PACKAGE
13
14
15

SCREEN #2
 0 \ KERNEL LOAD SCREEN        FOR CROSS COMPILER
 1 DECIMAL
 2 CR ." (C) Copyright 1987 by Phil Koopman Jr." CR
 3 CR CR ." Cross compiling kernel to CPU/32" CR CR
 4 4 76 THRU
 5 CR CR ." Cross compiliation completed." CR CR
 6 ." Type in:  CPU32" CR
 7 ."           3 LOAD"  CR CR
 8 CROSS-COMPILER                FORTH DEFINITIONS
 9
10 \ LIB-FORTH STUFF
11 CR CR ." SET UP FILE1 AS KERNEL.4TH" CR
12 : SETUP  FILE1 OPEN  LOAD-ALL ;
13
14
15

SCREEN #3
 0 \  MVP-FORTH --  LOAD SCREEN TO FINISH COMPILATION FROM CPU/32
 1 DECIMAL
 2 81 105 THRU     \ Load remainder of kernel
 3 CR CR ." Remainder of CPU/32 kernel loaded." CR
 4 CR ." (C) Copyright 1987 by Phil Koopman Jr." CR
 5 CR CR ." Loading CPU/32 math support." CR
 6 110 141 THRU    \ 64-bit integer support
 7 143 199 THRU    \ 32-bit floating point support
 8 CR CR ." Type in:  BYE"
```

```
 9    CR ."            SAVE-ALL"  CR CR
10 ." Do a SAVE-FORTH to a DOS file name CPU32.COM if desired" CR
11
12
13
14
15

SCREEN #4
 0 \  MVP-FORTH SOURCE --  DOUSE
 1 HEX   CROSS-COMPILER
 2 VARIABLE UP    20.  ( UP ! )
 3
 4 : DOUSE   ( -> ADDR )
 5    R>  @  UP  @  +  ;
 6
 7 B' DOUSE  FORTH  D@ DROP OFFFC AND CROSS-COMPILER DOUSE-ADDR !
 8
 9
10
11
12
13
14
15

SCREEN #5
 0 \  MVP-FORTH SOURCE -- USER VARIABLES 0 - 07
 1 HEX   CROSS-COMPILER
 2
 3 ( 0 USER )
 4 ( 1 USER )
 5 ( 2 USER )
 6  3 USER SPO              0FF.    ( SPO ! )
 7  4 USER RO               0FF.    ( RO ! )
 8 05 USER TIB             MEM-SIZE -480. D+  ( TIB ! )
 9 06 USER WIDTH            1F.     ( WIDTH ! )
10 07 USER WARNING          1.      ( WARNING ! )
11
12 DECIMAL
13
14
15

SCREEN #6
 0 \  MVP-FORTH SOURCE -- USER VARIABLES 08 - 0F
 1 HEX   CROSS-COMPILER
 2
 3 08 USER  FENCE
 4 09 USER  DP              2000. ( DP ! )
 5 0A USER  VOC-LINK
 6 0B USER  '-FIND
 7 0C USER  '?TERMINAL
 8 0D USER  'ABORT
 9 0E USER  'BLOCK
10 0F USER  'CR
11
12 DECIMAL
13
14
15
```

```
SCREEN #7
 0 \  MVP-FORTH SOURCE -- USER VARIABLES 10 - 17
 1 HEX   CROSS-COMPILER
 2
 3 10 USER 'EMIT
 4 11 USER 'EXPECT
 5 12 USER 'INTERPRET
 6 13 USER 'KEY
 7 14 USER 'LOAD
 8 15 USER 'NUMBER
 9 16 USER 'PAGE
10 17 USER 'R/W
11
12 DECIMAL
13
14
15

SCREEN #8
 0 \  MVP-FORTH SOURCE -- USER VARIABLES 18 - 1F
 1 HEX   CROSS-COMPILER
 2
 3 18 USER 'T&SCALC
 4 19 USER 'VOCABULARY
 5 1A USER 'WORD
 6 1B USER >IN           0.  { >IN ! }
 7 1C USER BASE          0A. { BASE ! }
 8 1D USER BLK           0.  { BLK ! }
 9 1E USER CONTEXT
10 1F USER CSP
11
12 DECIMAL
13
14
15

SCREEN #9
 0 \  MVP-FORTH SOURCE -- USER VARIABLES 20 - 27
 1 HEX   CROSS-COMPILER
 2
 3 20 USER CURRENT
 4 21 USER DPL
 5 22 USER FLD
 6 23 USER HLD
 7 24 USER OFFSET
 8 25 USER OUT          0. { OUT ! }
 9 26 USER R#
10 27 USER SCR
11
12 DECIMAL
13
14
15

SCREEN #10
 0 \  MVP-FORTH SOURCE -- USER VARIABLES 28 - 2F
 1 HEX   CROSS-COMPILER
 2
 3 28 USER STATE         0. { STATE ! }
```

```
 4 29 USER 'ISERVICE    \ Interrupt service vector
 5
 6
 7
 8
 9
10
11
12
13 DECIMAL
14
15

SCREEN #11
 0 \  MVP-FORTH SOURCE --  SP@  SP!   RP@  RP!    DEPTH
 1 HEX   CROSS-COMPILER
 2 : SP@   ( -> N )
 3    %DP@%  OFFF  AND  ;
 4
 5 : SP!   ( -> )
 6    OFFE   %DP!%  ;
 7
 8 : RP@   ( -> N )
 9    NOP %RP@%  OFFF  AND  ;
10
11 : RP!   ( -> )
12    R>  OFFF   %RP!%  >R ;
13
14 : DEPTH  ( -> N )
15     OFFD  SP@  -  ;        DECIMAL


SCREEN #12
 0 \  MVP-FORTH SOURCE --  >  0>    D<  U<
 1 HEX   CROSS-COMPILER
 2 : 0>   ( N -> FLAG )
 3    0  >  ;
 4
 5 : D<   ( D1 D2 -> FLAG )
 6    ROT  DDUP =
 7    IF      -ROT  DNEGATE  D+  0<
 8    ELSE   SWAP  <   SWAP_DROP
 9    THEN   SWAP_DROP   ;
10
11 : U<   ( U1 U2 -> FLAG )
12    0 SWAP  0 D<  ;
13
14 DECIMAL
15

SCREEN #13
 0 \  MVP-FORTH SOURCE --  1  2  BL    2+ 2-    EPRINT
 1 HEX   CROSS-COMPILER
 2 VARIABLE EPRINT    { 0 EPRINT ! }
 3 2. CONSTANT 2
 4 4. CONSTANT 4
 5 20. CONSTANT BL
 6 DECIMAL
 7 : 2+   ( N1 -> N2 )
 8    1+  1+  ;
```

```
 9 : 2-    ( N1 -> N2 )
10     1-  1-  ;
11 : 4+    ( N1 -> N2 )
12     4 +  ;
13 : 4-    ( N1 -> N2 )
14     4 -  ;
15

SCREEN #14
 0 \  MVP-FORTH SOURCE --  EXECUTE   <EMIT> <CR> <PAGE>
 1 HEX   CROSS-COMPILER
 2 VARIABLE DO-EXECUTE  -4 BDP +!  1. DATA, 1. DATA,
 3 ( DO-EXECUTE 4- )  CONSTANT EXEC-ADDR
 4 : EXECUTE  ( ADDR -> ) ( 2 NOP's allow store before instr fetch)
 5    2 OR    EXEC-ADDR !  NOP DO-EXECUTE NOP ;
 6
 7 : <EMIT>  ( CHAR -> )
 8    EPRINT @
 9    IF  9 SYSCALL  ELSE  8 SYSCALL  THEN   DROP   1 OUT +! ;
10 : <CR>   ( -> )
11    EPRINT @   7 SYSCALL  DROP   0 OUT ! ;
12
13 : <PAGE>  ( -> )
14    1 SYSCALL  ;
15 DECIMAL
SCREEN #15
 0 \  MVP-FORTH SOURCE --  <KEY>   <?TERMINAL>    INDEX
 1 HEX   CROSS-COMPILER
 2 : <KEY>  ( -> CHAR )
 3    0  2 SYSCALL  ;
 4
 5 : <?TERMINAL>   ( -> FLAG )
 6    0  5 SYSCALL  ;
 7
 8 DECIMAL
 9
10
11
12
13
14
15

SCREEN #16
 0 \  MVP-FORTH SOURCE --  DECIMAL  HEX
 1 HEX   CROSS-COMPILER
 2 : DECIMAL    0A  BASE !  ;
 3 : HEX        10  BASE !  ;
 4
 5
 6 DECIMAL
 7
 8
 9
10
11
12
13
14
15
```

```
SCREEN #17
 0 \  MVP-FORTH SOURCE -- +-  D+-  DABS
 1 HEX   CROSS-COMPILER
 2 : +-    ( N1 N2 -> N3 )
 3    0<    IF   NEGATE  EXIT  THEN ;
 4
 5 : D+-    ( D1 N2 -> D3 )
 6    0<    IF  DNEGATE  EXIT  THEN ;
 7
 8 : DABS    ( D1 -> D2 )
 9    DUP  D+-  ;
10
11 : MIN    ( N1 N2 -> N3 )
12    DDUP >     IF   SWAP_DROP  EXIT  THEN  DROP ;
13 : MAX    ( N1 N2 -> N3 )
14    DDUP <     IF   SWAP_DROP  EXIT  THEN  DROP  ;
15 DECIMAL

SCREEN #18
 0 \  MVP-FORTH SOURCE -- M+ M*  M/
 1 HEX   CROSS-COMPILER
 2 : M+    ( D1  N2  -> D3 )
 3    S->D  D+  ;
 4
 5 : M*    ( N1 N2 -> D3 )
 6    DDUP XOR   >R   ABS  SWAP  ABS
 7    U*  R> D+-  ;
 8
 9 : M/    ( D1 N2 -> N3 N4 )
10    OVER  D>R   DUP D+-   R@  ABS  U/MOD
11    R>  R@  XOR +-  SWAP  R>  +-  SWAP  ;
12
13 DECIMAL
14
15

SCREEN #19
 0 \  MVP-FORTH SOURCE -- M*/
 1 HEX   CROSS-COMPILER
 2 : M*/  ( D1  N1 N2 -> D2 )
 3    DDUP  XOR  SWAP   ABS  >R  SWAP
 4    ABS  >R  OVER XOR  -ROT  DABS
 5    SWAP R@  U*  ROT  R>  U*  ROT   0 D+
 6    R@  U/MOD   -ROT   R>  U/MOD
 7    SWAP_DROP   SWAP  ROT  D+-  ;
 8
 9
10
11
12
13 DECIMAL
14
15

SCREEN #20
 0 \  MVP-FORTH SOURCE -- *   /MOD  /  MOD
 1 HEX   CROSS-COMPILER
 2 : *    ( N1 N2 -> N3 )
 3    U*  DROP  ;
 4
```

```
 5 : /MOD    ( N1 N2 -> NREM NQUOT )
 6     >R  S->D  R>  M/  ;
 7
 8 : /    ( N1 N2 -> N3 )
 9     /MOD    SWAP_DROP  ;
10
11 : MOD    ( N1 N2 -> N3 )
12     /MOD    DROP  ;
13
14 DECIMAL
15

SCREEN #21
 0 \  MVP-FORTH SOURCE -- */MOD  */  M/MOD
 1 HEX    CROSS-COMPILER
 2 : */MOD      N1 N2 N3 -> NREM NQUOT )
 3     >R  S->D  R>  M/  ;
 4
 5 : */    ( N1 N2 N3 -> N4 )
 6     */MOD  SWAP_DROP  ;
 7
 8 : M/MOD    ( UD1 U2 -> U3 UD4 )
 9     >R 0  R@  U/MOD    R>  SWAP  >R
10     U/MOD  R>  ;
11
12
13
14 DECIMAL
15

SCREEN #22
 0 \  MVP-FORTH SOURCE -- EMIT  CR  PAGE
 1 HEX    CROSS-COMPILER
 2 B' <EMIT> FORTH D@ CC  ( 'EMIT ! )
 3 : EMIT  ( CHAR -> )
 4     'EMIT @  EXECUTE  ;
 5
 6 B' <CR> FORTH  D@ CC ( 'CR ! )
 7 : CR    ( -> )
 8     'CR @     EXECUTE  ;
 9
10 B' <PAGE> FORTH D@ CC ( 'PAGE ! )
11 : PAGE    ( -> )
12     'PAGE @  EXECUTE  ;
13
14 DECIMAL
15

SCREEN #23
 0 \  MVP-FORTH SOURCE -- ?TERMINAL  KEY
 1 HEX    CROSS-COMPILER
 2 B' <?TERMINAL> FORTH D@ CC ( '?TERMINAL ! )
 3 : ?TERMINAL    ( -> FLAG )
 4     '?TERMINAL    @    EXECUTE ;
 5
 6 B' <KEY>  FORTH D@ CC  ( 'KEY ! )
 7 : KEY    ( -> CHAR )
 8     'KEY @     EXECUTE  ;
 9
10
```

```
11
12
13
14 DECIMAL
15

SCREEN #24
 0 \  MVP-FORTH SOURCE -- SPACE  COUNT  TYPE
 1 HEX   CROSS-COMPILER
 2 : SPACE  ( -> )
 3    BL  EMIT ;
 4
 5 : COUNT  ( ADDR -> ADDR+4 N )
 6    DUP  4+   SWAP @ OFFFF AND   ;
 7
 8 : TYPE   ( ADDR N -> )
 9    DUP 0>
10    IF  OVER + SWAP
11       DO  I C@  EMIT   1 +LOOP
12    ELSE   DDROP
13    THEN  ;
14
15 DECIMAL

SCREEN #25
 0 \  MVP-FORTH SOURCE --  -TRAILING   SPACES
 1 HEX   CROSS-COMPILER
 2 : -TRAILING  ( ADDR N1 -> ADDR N2 )
 3    DUP  0
 4    DO   DDUP +  1-  C@  BL -
 5       IF  LEAVE   ELSE  1-  THEN
 6    LOOP ;
 7
 8 : SPACES   ( COUNT -> )
 9    0  MAX   ?DUP
10    IF  0  DO   SPACE  LOOP
11    THEN  ;
12
13
14
15 DECIMAL

SCREEN #26
 0 \  MVP-FORTH SOURCE --  PAD  HERE
 1 HEX   CROSS-COMPILER
 2 : HERE   ( -> ADDR )
 3    DP @  ;
 4
 5 : PAD    ( -> ADDR )
 6    HERE  4C +  ;
 7
 8
 9
10
11 DECIMAL
12
13
14
15
```

```
SCREEN #27
 0 \  MVP-FORTH SOURCE --  HOLD   <#  #>  #
 1 HEX   CROSS-COMPILER
 2 : HOLD   ( CHAR -> )
 3    1 NEGATE  HLD  +!  HLD @  C!  ;
 4
 5 : <#   ( D1 -> D1 )
 6    PAD  HLD  !  ;
 7
 8 : #>   ( UD -> ADDR N )
 9    DDROP  HLD @   PAD   OVER  -  ;
10
11 : #   ( UD1 -> UD2 )
12    BASE @   M/MOD   ROT   9 OVER <
13    IF  7 +   THEN     30 +   HOLD  ;
14
15 DECIMAL

SCREEN #28
 0 \  MVP-FORTH SOURCE --  <ABORT">  <.">      <ABORT' >
 1 HEX   CROSS-COMPILER
 2 VARIABLE <QUIT-ADDR>
 3 VARIABLE <WHERE-ADDR>
 4 : <ABORT">   ( FLAG -> )
 5   <<ABORT">>  ( Automatically EXIT if FLAG=0 )
 6    ( IF )  <WHERE-ADDR> @  EXECUTE    CR
 7    R@  COUNT  TYPE  SP!   <QUIT-ADDR> @  EXECUTE     ;
 8 (       ELSE  R> DUP @  +  4 + >R  THEN ;  )
 9 B' <ABORT"> FORTH D@ CC DROP  ABORT"-ADDR !
10
11 : <."> ( -> )
12    R@   COUNT   DUP 7 + R> +  OFFFFFFFC AND  >R   TYPE ;
13 B' <."> FORTH D@ CC DROP   ."-ADDR !
14 DECIMAL
15

SCREEN #29
 0 \  MVP-FORTH SOURCE --  SIGN   #S   D.R  D.
 1 HEX   CROSS-COMPILER
 2 : SIGN  ( N -> )
 3    0<   IF   2D  HOLD  EXIT  THEN  ;
 4
 5 : #S   ( UD -> 0 0 )
 6    BEGIN  #   DDUP   OR  NOT  UNTIL  ;
 7
 8 : D.R   ( D N -> )
 9    >R   SWAP   OVER  DUP D+-
10    <#  #S   ROT SIGN  #>
11    R>  OVER  -   SPACES   TYPE  ;
12
13 : D.   ( D -> )
14    0  D.R  SPACE ;
15 DECIMAL

SCREEN #30
 0 \  MVP-FORTH SOURCE --  .R  U.   .   ?
 1 HEX   CROSS-COMPILER
 2 : .R  ( N1 N2 -> )
 3    SWAP  S->D  ROT   D.R  ;
 4
```

```
 5 : U.   ( N1 -> )
 6    0  D. ;
 7
 8 : .    ( N1 -> )
 9    S->D  D.  ;
10
11 : ?    ( N1 -> )
12    @ . ;
13
14 DECIMAL
15

SCREEN #31
 0 \  MVP-FORTH SOURCE --  ?COMP  ?CSP  ?LOADING  ?PAIRS
 1 HEX   CROSS-COMPILER
 2 : ?COMP    ( -> )
 3    STATE @ NOT   ABORT" COMPILE ONLY"  ;
 4
 5 : ?CSP    ( -> )
 6    SP@  CSP @  -
 7    ABORT" DEFINITION NOT FINISHED"  ;
 8
 9 : ?LOADING    ( -> )
10    BLK @ NOT   ABORT" LOADING ONLY"  ;
11
12 : ?PAIRS    ( N1 N2 -> )
13    -   ABORT" CONDITIONALS NOT PAIRED"  ;
14 DECIMAL
15

SCREEN #32
 0 \  MVP-FORTH SOURCE --  ?STREAM   ?STACK
 1 HEX   CROSS-COMPILER
 2 : ?STREAM    ( -> )
 3    ABORT" INPUT STREAM EXHAUSTED" ;
 4
 5 : ?STACK    ( -> )
 6    SP@     OFFF AND  0080 <  ABORT" STACK EMPTY"
 7    SP@     OFFF AND  0100 <  ABORT" STACK FULL"  ;
 8
 9
10
11
12
13
14 DECIMAL
15

SCREEN #33
 0 \  MVP-FORTH SOURCE --  PICK  ROLL
 1 HEX   CROSS-COMPILER
 2 : PICK    ( N1 -> N2 )
 3    DUP 1 <   ABORT" PICK ARGUMENT < 1"
 4    <PICK> ;
 5
 6 : ROLL    ( N1 -> )
 7    DUP 1 <   ABORT" ROLL ARGUMENT < 1"
 8    <ROLL> ;
 9
10
```

```
11
12 DECIMAL
13
14
15

SCREEN #34
 0 \  MVP-FORTH SOURCE --  <CMOVE>   CMOVE
 1 HEX   CROSS-COMPILER
 2 \ : <CMOVE>   ( ADDR1 ADDR2 U -> )
 3 \     OVER + SWAP
 4 \     DO  DUP C@  I C!  1+  LOOP   DROP ;
 5
 6 \ : CMOVE   ( ADDR1  ADDR2  N -> )
 7 \   DUP 1 <
 8 \   IF  DDROP   DROP
 9 \   ELSE  <CMOVE>   THEN ;
10
11 : CMOVE   ( ADDR1 ADDR2 COUNT -> )
12   BEGIN  <CM-STEP>  UNTIL   DROP DDROP ;
13
14 DECIMAL
15


SCREEN #35
 0 \  MVP-FORTH SOURCE --  <<CMOVE>   <CMOVE
 1 HEX   CROSS-COMPILER
 2 \ : <<CMOVE>   ( ADDR1 ADDR2 U -> )
 3 \   >R   SWAP   R@ + 1-    SWAP  R@ + 1-  R>
 4 \   BEGIN   ?DUP    WHILE    <<CM-STEP>  REPEAT
 5 \   DDROP  ;
 6 \
 7 \ : <CMOVE   ( ADDR1  ADDR2  N -> )
 8 \   DUP 1 <
 9 \   IF  DDROP   DROP
10 \   ELSE  <<CMOVE>   THEN  ;
11
12 : <CMOVE   ( ADDR1 ADDR2 COUNT -> )
13   >R   SWAP   R@ + 1-    SWAP  R@ + 1-  R>
14   BEGIN  <<CM-STEP>  UNTIL   DROP DDROP ;
15 DECIMAL


SCREEN #36
 0 \  MVP-FORTH SOURCE --  DISK ACCESS CONSTANTS/VARIABLES
 1 HEX   MATH CROSS-COMPILER
 2 MEM-SIZE  -4. D+  CONSTANT LIMIT
 3
 4 1. CONSTANT #BUFF
 5 408. CONSTANT BUF-SIZE
 6
 7 ( LIMIT  #BUFF  BUF-SIZE )  FORTH  D*  D- OFFFFFF. DAND
 8     CC CONSTANT FIRST
 9
10 40. CONSTANT C/L
11 VARIABLE USE    (  FIRST  USE !  )
12 VARIABLE PREV   (  FIRST  PREV !  )
13 ( 0  OFFSET ! )
14
 5 DECIMAL
```

```
SCREEN #37
 0 \  MVP-FORTH SOURCE --  <R/W>  R/W
 1 HEX   CROSS-COMPILER
 2 : <R/W>   ( ADDR BLK FLAG -> )     ( No wrap over drive limits )
 3    IF    3 SYSCALL    ( Read )
 4    ELSE  4 SYSCALL    ( Write )
 5    THEN   DDROP  ;
 6
 7 B' <R/W> FORTH D@ CC ( 'R/W ! )
 8
 9 : R/W   ( ADDR BLK FLAG -> )
10    'R/W @  EXECUTE ;
11
12 DECIMAL
13
14
15

SCREEN #38
 0 \  MVP-FORTH SOURCE --  +BUF   BUFFER  UPDATE
 1 HEX   CROSS-COMPILER
 2 : +BUF   ( ADDR -> ADDR2 FLAG )
 3    BUF-SIZE  +  DUP  LIMIT =
 4    IF   DROP  FIRST  THEN
 5    DUP  PREV @  -  ;
 6
 7 : BUFFER   ( N -> ADDR )   ( Single buffer support only!!!!! )
 8    USE @  >R    R@ @  0<
 9    IF   R@ 4+  R@ @  7FFFFFFF AND  0 R/W   THEN
10    R@ !   R@ PREV !  R>  4+ ;
11
12 : UPDATE  ( -> )
13    PREV @ @   80000000 OR   PREV  @  !  ;
14
15 DECIMAL

SCREEN #39
 0 \  MVP-FORTH SOURCE --  <BLOCK>  BLOCK
 1 HEX   CROSS-COMPILER
 2 : <BLOCK>   ( N -> ADDR )   ( Supports only one block buffer )
 3    PREV @  @  7FFFFFFF AND   OVER =
 4    IF   DROP
 5    ELSE   DUP  BUFFER    SWAP   1  R/W
 6    THEN    PREV @  4+  ;
 7
 8 B' <BLOCK> FORTH D@ CC ( 'BLOCK ! )
 9
10 : BLOCK   ( N -> ADDR )
11    'BLOCK @   EXECUTE  ;
12
13 DECIMAL
14
15

SCREEN #40
 0 \ FILL                                (Character based)
 1 DECIMAL  CROSS-COMPILER
 2 : FILL  (  ADDR COUNT VALUE -> )
 3    OVER 0>
 4    IF
```

```
 5       -ROT  OVER + SWAP  DO  DUP I C!  LOOP   DROP
 6    ELSE   DDROP  DROP  THEN ;
 7
 8
 9
10
11
12
13
14
15

SCREEN #41
 0 \  MVP-FORTH SOURCE --  SAVE-BUFFERS  EMPTY-BUFFERS  CLEAR
 1 HEX   CROSS-COMPILER
 2 : SAVE-BUFFERS  ( -> )
 3    #BUFF  1+  0 DO    7FFFFFFF  BUFFER   DROP  LOOP  ;
 4
 5 : EMPTY-BUFFERS  ( -> )
 6    FIRST LIMIT  OVER -   0  FILL
 7  ( #BUFF 0 DO )  7FFFFFFF ( BUF-SIZE I * ) 0 FIRST + ! ( LOOP) ;
 8
 9 ( { EMPTY-BUFFERS }  )
10
11 : CLEAR   ( N -> )
12    OFFSET @  +   BUFFER   400  BL FILL  UPDATE ;
13
14 DECIMAL
15

SCREEN #42
 0 \  MVP-FORTH SOURCE --  <LINE>  .LINE  LIST
 1 HEX   CROSS-COMPILER
 2 : <LINE>   ( N1  N2  -> ADDR  COUNT )
 3    BLOCK  SWAP  C/L  *  +  C/L  ;
 4
 5 : .LINE   ( LINE SCR -> )
 6    <LINE>  -TRAILING  TYPE  ;
 7
 8 : LIST   ( SCR -> )
 9    CR   DUP  SCR !
10   ." SCREEN #" U.  10   0
11   DO   CR  I  3 .R   SPACE I SCR @ .LINE
12          ?TERMINAL  IF  <QUIT-ADDR> @ EXECUTE  THEN
13    LOOP    CR  ;
14 DECIMAL
15

SCREEN #43
 0 \  MVP-FORTH SOURCE --.  DIGIT  CONVERT
 1 HEX   CROSS-COMPILER
 2 : DIGIT   ( C N1 -> N2 TF / FF )
 3    SWAP  30 -     DUP  9 >
 4    IF    DUP  11 <  IF   DROP -1  ELSE  7 -  THEN  THEN
 5    DUP  ROT  < NOT   IF  DROP -1  THEN
 6    DUP  0<  IF   DROP 0  ELSE  -1  THEN  ;
 7
 8 : CONVERT   ( UD1 ADDR1 -> UD2 ADDR2 )
 9    BEGIN  1+ DUP  >R  C@   BASE @  DIGIT
```

```
10    WHILE   SWAP  BASE  @ U*  DROP   ROT  BASE @  U*
11       D+  DPL @  1+
12       IF  1 DPL +!  THEN
13       R>
14    REPEAT   R>  ;
15 DECIMAL

SCREEN #44
 0 \  MVP-FORTH SOURCE --   <NUMBER>  NUMBER
 1 HEX   CROSS-COMPILER
 2 : <NUMBER>   ( ADDR -> D )
 3    3 +  0 0 ROT  DUP 1+  C@    02D =   DUP >R  -   -1  DPL !
 4    CONVERT   DUP C@ BL >
 5    IF   DUP C@  02E =  NOT
 6       ABORT" NOT RECOGNIZED"   0 DPL !
 7       CONVERT   DUP C@  BL >    ABORT" NOT RECOGNIZED"
 8    THEN   DROP  R>
 9    IF  DNEGATE  THEN  ;
10
11 B' <NUMBER> FORTH D@ CC ( 'NUMBER ! )
12
13 : NUMBER   ( ADDR -> D )
14    'NUMBER @ EXECUTE  ;
15 DECIMAL

SCREEN #45
 0 \  MVP-FORTH SOURCE --   <EXPECT>
 1 HEX   CROSS-COMPILER
 2 : <EXPECT>   ( ADDR N -> )
 3    OVER  0 SWAP  DDUP 1+ C! C!
 4    OVER + OVER
 5    DO  KEY  DUP  8 =
 6       IF  DROP  DUP  I =  1 AND  DUP  R>  2-  + >R
 7          IF    07
 8          ELSE  8  DUP  EMIT  BL EMIT  -3 OUT +!  THEN
 9       ELSE   DUP  0D =
10          IF    LEAVE  DROP  BL  0
11          ELSE  DUP  THEN   I C!      0 I 1+  DDUP 1+ C! C!
12       THEN   EMIT
13    LOOP   DROP ;
14 B' <EXPECT> FORTH D@ CC ( 'EXPECT ! )
15 DECIMAL

SCREEN #46
 0 \  MVP-FORTH SOURCE --   'STREAM    EXPECT   QUERY
 1 HEX   CROSS-COMPILER
 2 : 'STREAM   ( -> ADDR )
 3    BLK @  ?DUP
 4    IF  BLOCK   ELSE   TIB @   THEN
 5    >IN  @  +  ;
 6
 7 : EXPECT   ( ADDR N -> )
 8    'EXPECT @  EXECUTE ;
 9
10 : QUERY   ( -> )
11    TIB @  50 EXPECT   0 >IN  !  ;
12
13 DECIMAL
14
15
```

```
SCREEN #47
 0 \  MVP-FORTH SOURCE --  ENCLOSE
 1 HEX   CROSS-COMPILER
 2 : ENCLOSE  ( ADDR1 C -> ADDR1  N1 N2 N3 )
 3    >R DUP  BEGIN  <ENCLA> UNTIL     DDUP SWAP -  SWAP
 4         ( ..-> ADDR1 N1 ADDR1+N1 )  ( Returns ..-> C )
 5 ( >R    -1   OVER  1-                                    )
 6 ( BEGIN  SWAP 1+ SWAP 1+  DUP C@ R@ = NOT UNTIL   ( #### )
 7
 8     BEGIN    <ENCLB>    UNTIL
 9              ( ..-> ADDR1 N1 ADDRN2 )
10 (  BEGIN  DUP C@  DUP  R@ =  NOT  SWAP 0= NOT AND          )
11 (                 WHILE  1+  REPEAT   ( ######### )
12    3_PICK  -                                  \
13    OVER   4_PICK  +  C@   IF  DUP 1+  ELSE  0  THEN
14    R>  DROP  ;
15 DECIMAL

SCREEN #48
 0 \  MVP-FORTH SOURCE --  DOVOC  FORTH
 1 HEX   CROSS-COMPILER
 2 : DOVOC   ( -> )
 3    R>  CONTEXT !  ;
 4
 5 : FORTH  ( -> )   ;
 6  -4 BDP +!
 7    B' DOVOC  FORTH D@ CC  DATA,
 8    0. DATA,      ( LINK ADDR TO LAST WORD IN DICTIONARY )
 9    80000001.  DATA,
10    BL  0 DATA,
11    0. DATA,      ( VOCABULARY LINK  )
12
13 DECIMAL
14
15

SCREEN #49
 0 \ OPTIMIZING COMPILER VARIABLES, ETC.
 1 DECIMAL CROSS-COMPILER
 2
 3 VARIABLE OPTIMIZE?     -1. ( OPTIMIZE? ! )
 4
 5 VARIABLE OPT-STATUS
 6   \ 1 = previous instruction is a defaulted JMP
 7    \ 4 = previous instruction is a CALL
 8    \ 8 = previous instruction can not be changed
 9
10 : DEFAULT-JUMP   1 OPT-STATUS ! ;
11 : IS-A-CALL      4 OPT-STATUS ! ;
12 : DON'T-DISTURB  8 OPT-STATUS ! ;
13
14
15

SCREEN #50
 0 \  MVP-FORTH SOURCE --      ALLOT  ,  LATEST  DEFINITIONS
 1 DECIMAL CROSS-COMPILER
 2
 3 : ALLOT   ( N -> )
 4    DP +!  ;
```

```
 5
 6 : C,   ( B -> )   HERE C!   1 ALLOT DON'T-DISTURB ;
 7
 8 : WORD-ALIGN   ( -> )
 9    4  DP @  3 AND  -   3 AND
10    ?DUP  IF  0 DO BL C, LOOP  THEN ;
11
12 : ,    ( N -> )
13    WORD-ALIGN HERE !  4 ALLOT  DON'T-DISTURB ;
14 : X,    ( N -> )
15    WORD-ALIGN HERE !  4 ALLOT  ;

SCREEN #51
 0 \  MVP-FORTH SOURCE --      LATEST  DEFINITIONS
 1 DECIMAL CROSS-COMPILER
 2 : LATEST    ( -> )
 3    CURRENT  @  @  ;
 4
 5 : DEFINITIONS   ( -> )
 6    CONTEXT @  CURRENT ! ;
 7
 8 ( FORTH  DEFINITIONS )
 9
10
11
12
13
14
15

SCREEN #52
 0 \ COMPILER DICTIONARY HANDLING    --1
 1 HEX      CROSS-COMPILER
 2 : OPCODE,   ( MICRO-CODE-INSTRUCTION -> )
 3    FF800000 AND    HERE  4+  OR    X,    DEFAULT-JUMP ;
 4
 5 : CALL,     ( Subroutine-address -> )
 6    007FFFFC AND    OPTIMIZE? @
 7    IF  OPT-STATUS @ 1 =
 8       IF  -4 ALLOT  HERE @
 9            FF800000 AND  OR    THEN
10    THEN
11    2 OR X,  IS-A-CALL    ;
12 DECIMAL
13
14
15

SCREEN #53
 0 \ COMPILER DICTIONARY HANDLING    --2
 1 HEX       CROSS-COMPILER
 2 : EXIT,     ( -> )
 3    1
 4    OPTIMIZE? @
 5    IF OPT-STATUS @ 1 =     ( If 1 status, combine the EXIT      )
 6       IF  -4 ALLOT  HERE @
 7            FF800000    AND   OR    THEN
 8       OPT-STATUS @ 4 =     ( If 4 status, do tail recurs. elim.)
 9       IF  DROP  -4 ALLOT  HERE    @
10            FFFFFFFC    AND  THEN
```

```
11     THEN
12     X, DON'T-DISTURB ;
13
14 DECIMAL
15


SCREEN #54
 0 \  MVP-FORTH SOURCE --  <WORD>  WORD
 1 HEX   CROSS-COMPILER
 2 : <WORD>    ( CHAR -> ADDR )
 3     WORD-ALIGN
 4     'STREAM   SWAP   ENCLOSE   DDUP >
 5     IF  ( end-stream ) DDROP DDROP  0 HERE 4+ DDUP 4+ ! !
 6     ELSE    >IN  +!   OVER  -   DUP  >R
 7        HERE 4+ !   +   HERE  8 +   R>  DUP  OFFFF >
 8        ABORT" INPUT > 65535"  1+  CMOVE
 9     THEN  HERE  4+ ;
10
11 B' <WORD> FORTH D@ CC ( 'WORD ! )
12
13 : WORD    ( CHAR -> ADDR )
14     'WORD @  EXECUTE  ;
15 DECIMAL

SCREEN #55
 0 \  MVP-FORTH SOURCE --   TRAVERSE
 1 HEX   CROSS-COMPILER
 2 \ ADDR1 is presumed to be a word address
 3 : TRAVERSE  ( ADDR1  N ->  ADDR2 )
 4     0<
 5     IF    ( To length word   ADDR1 is last char in header string )
 6        BEGIN   4-   DUP  @  0<  UNTIL
 7     ELSE  ( To end of name string  ADDR1 is length word )
 8        DUP  C@   1 MAX  +  3 +   FFFFFFFC  AND
 9     THEN  ;
10
11 DECIMAL
12
13
14
15


SCREEN #56
 0 \  MVP-FORTH SOURCE --   LFA PFA NFA     CFA
 1 HEX   CROSS-COMPILER
 2 \  Header structure:
 3 \   LFA    NFA    ..TEXT..   PFA     ( Note: NO CFA !! )
 4 : NFA   ( PFA -> NFA )
 5     -1 TRAVERSE  ;
 6
 7 : LFA   ( PFA -> LFA )
 8     NFA  4-  ;
 9
10 : PFA   ( NFA -> PFA )
11     1 TRAVERSE  4+ ;
12
13 : CFA   ( PFA -> CFA  )  ;   ( This will work most of the time )
14 DECIMAL
15
```

```
SCREEN #57
 0 \  MVP-FORTH SOURCE --   <$MATCH>
 1 HEX   CROSS-COMPILER
 2 : <$MATCH>  ( TXTADDR1 NFAADDR2 -> MATCH-FLAG ) ( ADDR-LENGTH )
 3    OVER @ OFFFF AND    OVER @  2000FFFF AND  =
 4    IF     ( same length )
 5       -1  -ROT  4+  SWAP  DUP  4+  SWAP   @  OFFFF AND  1-
 6      BEGIN  <$=STEP>  DUP 0<  UNTIL
 7 (  BEGIN  ROT DUP C@ >R 1+   ROT DUP C@ >R 1+  ROT  R> R> =    )
 8 (         IF 1-  ELSE DROP >R >R  DROP 0 R> R> -1 THEN          )
 9 (     DUP 0< UNTIL                                              )
10       DROP  DDROP
11    ELSE
12       ( Different lengths ) DDROP  0
13    THEN  ;
14 DECIMAL
15

SCREEN #58
 0 \  MVP-FORTH SOURCE --   <FIND>  <-FIND>
 1 HEX   CROSS-COMPILER
 2 : <FIND>   ( TEXT-ADDR  NFA-ADDR -> PFA STATUS -1 / 0 )
 3    BEGIN   DDUP  <$MATCH>
 4       IF    SWAP DROP   DUP  PFA  SWAP  @  -1    -1
 5       ELSE  4-  @
 6             DUP  IF   0     ELSE   DDROP  0  -1   THEN
 7       THEN
 8     UNTIL    ;
 9
10 : <-FIND>   ( -> PFA STATUS -1 / 0 )
11    BL WORD  CONTEXT  @ @   <FIND> ;
12
13 B' <-FIND> FORTH D@ CC ( '-FIND ! )
14
15 DECIMAL

SCREEN #59
 0 \  MVP-FORTH SOURCE --  -FIND   FIND
 1 HEX   CROSS-COMPILER
 2 : -FIND  ( -> PFA STATUS -1 / 0 )
 3    '-FIND @  EXECUTE ;
 4
 5 : FIND   ( -> ADDR )
 6    -FIND   IF  DROP  ELSE  0  THEN  ;
 7
 8
 9
10
11
12 DECIMAL
13
14
15

SCREEN #60
 0 \  MVP-FORTH SOURCE --  COMPILE  [COMPILE]   [  ]
 1 HEX   CROSS-COMPILER
 2 : COMPILE  ( -> )
 3     R>  DUP 4+  >R  @
 4     DUP 3 AND  IF , ELSE  FF800003  AND  HERE 4+ OR  , THEN
```

```
 5          DON'T-DISTURB ;
 6 : [COMPILE]   : -> )          ( Not required when in cross-comp )
 7    ?COMP  -FIND  NOT    ABORT" NOT FOUND"
 8       10000000 AND IF    @  FF800000 AND  HERE 4 + OR
 9            ELSE  2 OR THEN   ,   DON'T-DISTURB ;    IMMEDIATE
10
11 : [   ( -> )
12    0 STATE !  ;    IMMEDIATE
13 : ]   ( -> )
14    C0000000  STATE !  ;  IMMEDIATE
15 DECIMAL

SCREEN #61
 0 \  MVP-FORTH SOURCE --  SMUDGE       SO  BLANK
 1 HEX   CROSS-COMPILER
 2 : SMUDGE   ( -> )
 3    LATEST  3 + 20  TOGGLE  ;
 4
 5 : SO   ( -> )
 6    SP0  @  ;
 7
 8 : BLANK   ( -> )
 9    BL FILL  ;
10
11 DECIMAL
12
13
14
15

SCREEN #62
 0 \  MVP-FORTH SOURCE --  DLITERAL  LITERAL
 1 HEX   CROSS-COMPILER
 2 : LITERAL   ( N -> N /  )
 3    STATE @
 4    IF   COMPILE  LIT     ,
 5         HERE 8 - @  FF800000 AND  HERE OR  2 OR
 6         HERE 8 -  !  DON'T-DISTURB  THEN  ;  IMMEDIATE
 7
 8 : DLITERAL   ( N -> N /  )
 9    STATE @
10    IF  SWAP   ( [COMPILE] )  LITERAL   ( [COMPILE] )  LITERAL
11    THEN ;   IMMEDIATE
12
13 DECIMAL
14
15


 SCREEN #63
  0 \  MVP-FORTH SOURCE --  <INTERPRET>
  1 HEX   CROSS-COMPILER
  2 : <INTERPRET>   ( -> )
  3    BEGIN  -FIND
  4       IF  DUP  STATE @  U<
  5          IF   DUP  10000000 AND
  6             IF SWAP @ OPCODE, ELSE  SWAP CALL,  THEN
  7             8000000 AND  IF  DON'T-DISTURB THEN
  8          ELSE  4000000 AND  ABORT" POISON WORD"  EXECUTE THEN
  9       ELSE   HERE 4+  NUMBER  DPL @  1+
```

```
10          IF  ( [COMPILE] )  DLITERAL
11          ELSE  DROP  ( [COMPILE] )  LITERAL    THEN
12        THEN  ?STACK    AGAIN  ;
13 B' <INTERPRET> FORTH D@ CC ( 'INTERPRET ! )
14 DECIMAL
15

SCREEN #64
 0 \  MVP-FORTH SOURCE --      INTERPRET &  NULL
 1 HEX   CROSS-COMPILER
 2 : INTERPRET  ( -> )
 3    'INTERPRET @ EXECUTE  ;
 4
 5 : X   ( -> )     (  NULL )
 6    BLK @
 7    IF   STATE @   ?STREAM   THEN
 8    DR> DDROP R> DROP  ;     IMMEDIATE
 9
10 B' X  FORTH D@ CC
11  ( FFFFFFFC. AND   NFA  C0000000.  OVER !   0 SWAP 4+ !   )
12
13 DECIMAL
14
15

SCREEN #65
 0 \  MVP-FORTH SOURCE --   QUIT   <ABORT>
 1 HEX   CROSS-COMPILER
 2 : QUIT   ( -> )
 3    0 BLK !      ( [COMPILE] )  [
 4    BEGIN   RP!   CR   QUERY  INTERPRET
 5       STATE @  NOT
 6       IF  ."   (CPU/32)OK"   THEN
 7    AGAIN ;
 8 B' QUIT FORTH D@ CC ( <QUIT-ADDR> ! )     \ Vector for <ABORT">
 9
10 : <ABORT>   ( -> )
11    RP!  SP!    0 SET-FLAGS DROP  ( enable interrupts )
12    ( [COMPILE] )  FORTH  DEFINITIONS   QUIT  ;
13
14 B' <ABORT> FORTH D@ CC (  'ABORT ! )
15 DECIMAL

SCREEN #66
 0 \  MVP-FORTH SOURCE --   ABORT
 1 HEX   CROSS-COMPILER
 2 : ABORT  ( -> )
 3    'ABORT @  EXECUTE ;
 4
 5 : COLD  ( -> )
 6    EMPTY-BUFFERS  PAGE
 7    20 SPACES ." The WISC CPU/32" CR
 8    20 SPACES ." with MVPFORTH/32" CR
 9    25 SPACES ." 6/1/87" CR  CR
10    19 SPACES ." (C) 1987  By: Phil Koopman Jr." CR CR
11     0 EPRINT !
12     FIRST  USE !   FIRST  PREV !  0 OFFSET !
13     ."   (CPU/32)OK"
14    DECIMAL   ABORT  ;
15 DECIMAL
```

```
SCREEN #67
 0 \ INTERRUPT SERVICE ROUTINES  --1
 1 HEX     CROSS-COMPILER
 2 : INTERRUPT-DECODE ( FLAG -> )
 3 ( Assume stack pointers are close to value that cause problem )
 4    DUP   01000000 AND  IF CR ." DATA STACK ERROR="
 5      SP@ 80 + OFFF AND  0100 <  IF ." STACK EMPTY"
 6                                 ELSE ." STACK FULL"  THEN  THEN
 7    DUP   02000000 AND  IF CR ." RETURN STACK ERROR="
 8      RP@ 80 + OFFF AND  0100 <  IF ." STACK EMPTY"
 9                                 ELSE ." STACK FULL"  THEN  THEN
10    DUP   04000000 AND  IF CR ." HOST REQUEST=" DUP FF AND . THEN
11    DUP   08000000 AND  IF CR ." PARITY ERROR" THEN
12    DUP   70000000 AND  IF CR ." SOFTWARE INTERRUPT" THEN
13    DROP ;
14 DECIMAL
15

SCREEN #68
 0 \ INTERRUPT SERVICE ROUTINES
 1 DECIMAL  CROSS-COMPILER
 2 : ISERVICE    'ISERVICE @ EXECUTE ;
 3
 4 : INTERRUPT-SERV   ( ADDRESS FLAGS -> )
 5    -1 SET-FLAGS DROP  ( Clear & mask interrupts )
 6    CR CR ." ***** INTERRUPT *****" CR     HEX
 7    DUP ." FLAGS=" U.   SWAP ." ADDRESS=" U.
 8    INTERRUPT-DECODE  DECIMAL
 9    ABORT ;
10 HEX
11 B' ISERVICE FORTH D@  DROP FFFC AND 0
12 CC ! 0 ! }   \ Vector for INTERRUPT SERVICE
13    \ Initialize to a JUMP to INTERRUPT-SERV
14 B' INTERRUPT-SERV  FORTH D@  CC ! 'ISERVICE ! }
15 DECIMAL

SCREEN #69
 0 \  MVP-FORTH SOURCE --   <LOAD>  LOAD   THRU
 1 HEX   CROSS-COMPILER
 2 : <LOAD>   ( N -> )
 3    ?DUP NOT   ABORT" UNLOADABLE"
 4    BLK @  >R  >IN  @  >R
 5    0 >IN !   BLK !  INTERPRET
 6    R>  >IN  !  R>  BLK !  ;
 7
 8 B' <LOAD> FORTH D@ CC ! 'LOAD ! }
 9 : LOAD   ( N -> )
10    'LOAD @  EXECUTE ;
11
12 : THRU  ( N1 N2 -> )
13    1+ SWAP   DO I U.  I LOAD
14       ?TERMINAL  ABORT" BREAK..."  LOOP  ;
15 DECIMAL

SCREEN #70
 0 \  MVP-FORTH SOURCE --   CREATE
 1 HEX   CROSS-COMPILER
 2 : CREATE   ( -> )
 3    WORD-ALIGN   BL WORD    DUP   4+  C@
 4    0= ABORT" ATTEMPTED TO REDEFINE NULL"
```

```
 5    DUP  CONTEXT @  @  <FIND>
 6    IF  DDROP  WARNING @
 7       IF  DUP  COUNT  TYPE  SPACE  ." ISN'T UNIQUE" CR
 8       THEN
 9    THEN
10    LATEST ,  ( Store LFA )    HERE  CURRENT  @  !
11    C@      DUP   80000000 OR  ,
12    ALLOT  ( HERE happens to align with text of string )
13    WORD-ALIGN     22000002 ( Call DOVAR )  HERE 4 +  OR    ,
14    DON'T-DISTURB     ;
15 DECIMAL

SCREEN #71
 0 \  MVP-FORTH SOURCE --   WHERE
 1 HEX   CROSS-COMPILER
 2 : WHERE  ( -> )
 3    BLK @
 4    IF  BLK @  DUP  SCR !   CR  CR  ." SCREEN #"
 5       DUP .  >IN @  3FF MIN  C/L  /MOD  DUP
 6       ." LINE #"  .  C/L  *  ROT  BLOCK  +
 7       CR CR  C/L  -TRAILING  TYPE
 8       >IN  @   3FF >  1 AND  +
 9    ELSE  >IN @
10    THEN   CR  HERE 4+  @   DUP >R  -  HERE 4+ R@  +
11    DROP  1-  SPACES  R>  0
12    DO  5E  EMIT  LOOP  ;
13 B' WHERE FORTH D@ CC ( <WHERE-ADDR> ! )   \ Vector for <ABORT">
14 DECIMAL
15

SCREEN #72
 0 \  MVP-FORTH SOURCE --   :     VARIABLE
 1 HEX   CROSS-COMPILER
 2 : :   ( -> )
 3    SP@  CSP !    CURRENT @  CONTEXT !
 4    CREATE  -4  ALLOT  SMUDGE  ]  ;
 5
 6 : VARIABLE  ( -> )
 7    CREATE  4 ALLOT  ;
 8
 9 DECIMAL
10
11
12
13
14
15

SCREEN #73
 0 \  MVP-FORTH SOURCE --   \   (
 1 HEX   CROSS-COMPILER
 2 : $   ( -> )     ( alias for "("  )
 3    -1 >IN +!   29  WORD  DUP @  +  4+
 4    C@  29 =   NOT  ?STREAM  ;    IMMEDIATE
 5
 6 28.  BLATEST @ 4 +  0 ( ! )
 7
 8 : %   ( -> )     ( alias for "\" )
 9    ?LOADING  >IN   @  C/L  /   1+
10    C/L   *  >IN  !  ;         IMMEDIATE
11
```

```
12 5C.  BLATEST @  4 +  0 { ! }
13
14 DECIMAL
15

SCREEN #74
 0 \  MVP-FORTH SOURCE --   '
 1 HEX    CROSS-COMPILER
 2 : '    ( -> ADDR )
 3    -FIND  NOT    ABORT" NOT FOUND"
 4    DROP  ( [COMPILE] ) LITERAL  ;  IMMEDIATE
 5
 6
 7
 8
 9 DECIMAL
10
11
12
13
14
15

SCREEN #75
 0 \  MVP-FORTH SOURCE --    BYE  ;
 1 HEX    CROSS-COMPILER
 2 : BYE  ( -> )
 3       SAVE-BUFFERS  HALT  ;
 4
 5 : &    ( -> )          ( Redefine this word to be ,; on board )
 6    ?CSP    ( COMPILE  EXIT )  EXIT,  SMUDGE
 7    ( [COMPILE] )  [    ;         IMMEDIATE
 8 03B. ( ";" )    BLATEST @ 4 +  0 { ! }
 9
10 DECIMAL
11
12
13
14
15

SCREEN #76
 0 \  MVP-FORTH SOURCE --   LAST SCREEN LOADED
 1 HEX    CROSS-COMPILER
 2 BLATEST @  0 { CONTEXT @  !  }
 3 BHERE  0 { DP ! }
 4 \ Set up COLD vector for cold entry point at addr 5
 5 B' COLD FORTH D@ CC { 14. !    FF800018. 18. ! }
 6 DECIMAL
 7
 8
 9
10
11
12
13
14
15
```

```
SCREEN #77
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15

SCREEN #78
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15

SCREEN #79
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15

SCREEN #80
 0
 1
 2
 3
 4
```

```
 5
 6
 7
 8
 9
10
11
12
13
14
15

SCREEN #81
 0 \  BOARD FORTH  ---   CONSTANT       USER      IMMEDIATE
 1 HEX
 2 : CONSTANT  ( N -> )
 3    CREATE  -4 ALLOT
 4    [  ' DOCON  @  [illegible] AND  ] LITERAL
 5          OPCODE,     ,  DON'T-DISTURB  ;
 6
 7 : USER   ( N -> )
 8    CREATE   -4 ALLOT
 9    [ ' DOUSE  2 OR  ]   LITERAL  CALL,
10                         4 *   ,    DON'T-DISTURB ;
11
12 : IMMEDIATE   ( -> )
13    LATEST 3 +  40 TOGGLE  ;
14 DECIMAL
15

SCREEN #82
 0 \  BOARD FORTH  ---   DOES>     POISON  SPECIAL
 1 HEX
 2 : POISON       ( -> )
 3    LATEST 3 + 04 TOGGLE  ;
 4
 5 : SPECIAL      ( -> )
 6    LATEST 3 + 08 TOGGLE  ;
 7
 8 : <DOES>  ( -> )   ( Compilation helper for DOES> )
 9    R>  07FFFFC AND  2 OR  LATEST PFA  !  DON'T-DISTURB  ;
10
11 : DOES>   ( -> PFA )
12    ?CSP   COMPILE <DOES>
13    COMPILE  R>   ;   IMMEDIATE
14
15 DECIMAL

SCREEN #83
 0 \ IF..THEN
 1 HEX
 2 : IF      ( -> PATCH-ADDR 2 2 )
 3    HERE  COMPILE OBRANCH   2 2  ;
 4 IMMEDIATE
 5
 6 : THEN   ( PATCH-ADDR OR-VALUE 2 -> )
 7    ?COMP   2 ?PAIRS
 8    HERE  3 PICK @  FF800000 AND    OR
 9       OR   SWAP ! DON'T-DISTURB ;
10 IMMEDIATE
```

```
11 DECIMAL
12
13
14
15

SCREEN #84
 0 \ ELSE
 1 HEX
 2 : ELSE    ( PATCH-ADDR1 2 2 -> PATCH-ADDR2 0 2 )
 3     2 ?PAIRS              0 ,   ( Place for JUMP instruction )
 4     OPTIMIZE? @
 5     IF  OPT-STATUS @    1 =
 6        IF   ( Redirect jump to after the THEN )
 7            -4 ALLOT      HERE 4- DUP @
 8                    FF800000  AND  SWAP !
 9        THEN
10     THEN
11     HERE  4-
12     -ROT    2  [COMPILE] THEN     0  2    ;
13 IMMEDIATE
14 DECIMAL
15

SCREEN #85
 0 \ BEGIN  AGAIN
 1 HEX
 2 : BEGIN      ( -> JMP-ADDR  1 )
 3     ?COMP  HERE    1  DON'T-DISTURB  ;
 4 IMMEDIATE
 5
 6 : AGAIN      ( JMP-ADDR 1 -> )
 7     1 ?PAIRS
 8     ,   DON'T-DISTURB ;
 9 IMMEDIATE                DECIMAL
10
11
12
13
14
15

SCREEN #86
 0 \    UNTIL    & COUNT-DOWN
 1 HEX
 2 : UNTIL      ( JMP-ADDR  1 -> )
 3     1 ?PAIRS    COMPILE OBRANCH    -4 ALLOT
 4     HERE @  FF800000  AND    OR  2 OR    , ;
 5 IMMEDIATE
 6
 7 \ Count-down top of stack element & exit/drop when 0
 8 : COUNT-DOWN   ( JMP-ADDR  1 -> )
 9     1 ?PAIRS    COMPILE <COUNT-DOWN>    -4 ALLOT
10     HERE @  FF800000  AND    OR  2 OR    , ;
11 IMMEDIATE
12
13 DECIMAL
14
15
```

```
SCREEN #87
 0  \  WHILE  REPEAT
 1 HEX
 2 : WHILE      ( JMP-ADDR 1 ->  JMP-ADDR 1  PATCH-ADDR 2 3 )
 3    [COMPILE]  IF   2+  ;
 4 IMMEDIATE
 5
 6 : REPEAT     ( JMP-ADDR 1  PATCH-ADDR 2 3 -> )
 7    >R >R >R  [COMPILE] AGAIN
 8    R> R> R>  2-  [COMPILE] THEN  ;
 9 IMMEDIATE
10
11 : EXIT  ( -> )
12    STATE @
13    IF ( compiling )  EXIT,
14    ELSE ( interpretting ) R> R> R> DDROP DROP THEN ;
15 IMMEDIATE           DECIMAL

SCREEN #88
 0 \   DO
 1 HEX
 2 : DO      ( -> JMP-ADDR 4 )
 3    COMPILE <DO>    HERE  4  ;
 4 IMMEDIATE
 5
 6 DECIMAL
 7
 8
 9
10
11
12
13
14
15

SCREEN #89
 0 \  LOOP  /LOOP  +LOOP
 1 HEX
 2 : +LOOP   ( JMP-ADDR  4 -> )
 3    4 ?PAIRS  COMPILE <+LOOP>   -4 ALLOT
 4    HERE @  FF800000  AND   OR  2 OR   , ;
 5 IMMEDIATE
 6
 7 : LOOP    ( JMP-ADDR  4 -> )
 8    4 ?PAIRS  COMPILE <LOOP>   -4 ALLOT
 9    HERE @  FF800000  AND   OR  2 OR   , ;
10 IMMEDIATE
11
12 : /LOOP   ( JMP-ADDR  4 -> )
13    [COMPILE] +LOOP ;
14 IMMEDIATE
15 DECIMAL

SCREEN #90
 0 \  BOARD FORTH  ---   ABORT"
 1 HEX
 2 : ABORT"   ( FLAG -> )
 3    ?COMP    'STREAM   C@  22 =
 4    IF   1 >IN  +!    0 ,
 5    ELSE  22  WORD  DUP  @  4+   SWAP  OVER
```

```
 6       +  C@  22 =  NOT  ?STREAM  COMPILE <ABORT">
 7  ( Even multiple of 4 bytes )  HERE @  3 + FFFC AND  HERE !
 8       ALLOT  WORD-ALIGN
 9    THEN  DON'T-DISTURB  ;   IMMEDIATE
10
11 DECIMAL
12
13
14
15

SCREEN #91
 0 \  BOARD FORTH  ---   ."
 1 HEX
 2 : ."   ( -> )
 3    'STREAM   C@  22 =
 4    IF    1 >IN  +!  0 ,
 5    ELSE  22 WORD  DUP  @  4+  OVER  + C@
 6       22 = NOT  ?STREAM   STATE @
 7       IF   @ 4+   COMPILE <."> ALLOT   WORD-ALIGN
 8       ELSE  COUNT TYPE
 9       THEN
10    THEN  DON'T-DISTURB  ;   IMMEDIATE
11
12 DECIMAL
13
14
15

SCREEN #92
 0 \  WORD DUMP  FOR CPU/32
 1 HEX         \ Dump 32-bit words
 2 : WDUMP      ( waddr wcount --- )
 3    SWAP FFFFFFFC AND  SWAP   BASE @ >R   HEX  4 *  0
 4    DO  CR DUP I + DUP 0 6 D.R 2 SPACES DUP
 5      10 0  DO DUP I + @  0 9 D.R   4 /LOOP
 6      DROP 3 SPACES   10 0
 7      DO DUP I + C@ DUP 20 < OVER 7E > OR
 8        IF DROP 2E THEN    EMIT
 9      LOOP  DROP
10      ?TERMINAL IF LEAVE THEN      10
11    /LOOP   DROP  CR R> BASE ! ;
12
13  DECIMAL
14
15

SCREEN #93
 0 \  DUMP  FOR CPU/32
 1 HEX
 2 : DUMP      ( addr count --- )
 3    BASE @ >R   HEX       0
 4    DO  CR DUP I + DUP 0 6 D.R 2 SPACES DUP  8 0
 5      DO DUP I + C@   3 .R  LOOP
 6      DROP SPACE  DUP 8 +    8 0
 7      DO DUP I + C@   3 .R  LOOP
 8      DROP 3 SPACES   10 0
 9      DO DUP I + C@ DUP 20 < OVER 7E > OR
10        IF DROP 2E THEN    EMIT
11      LOOP  DROP
```

```
12      ?TERMINAL IF LEAVE THEN  10
13    /LOOP   DROP  CR R> BASE ! ;
14
15  DECIMAL

SCREEN #94
 0 \  VOCABULARY -- fig
 1 HEX
 2 : VOCABULARY
 3    CREATE
 4       CURRENT @  4+  ,
 5       80C00001 ,
 6       BL ,
 7       HERE  VOC-LINK @ ,   VOC-LINK !
 8    DOES> CONTEXT  ! ;
 9 DECIMAL
10
11 ' FORTH  16 + VOC-LINK !
12
13
14
15

SCREEN #95
 0 \  FORGET
 1 DECIMAL
 2 : FORGET     BL WORD CURRENT @ @ <FIND> 0=
 3    ABORT" Not in current vocabulary."
 4    DROP  NFA DUP FENCE @ U<   ABORT" In protected dictionary."
 5    >R  R@ CONTEXT @ U<
 6    IF [COMPILE] FORTH THEN
 7    R@ CURRENT @ U<
 8    IF [COMPILE] FORTH DEFINITIONS THEN
 9    VOC-LINK @
10    BEGIN R@ OVER U< WHILE @ REPEAT
11    DUP VOC-LINK !
12    BEGIN DUP 8 -
13       BEGIN 4- @ DUP R@ U< UNTIL
14       OVER 12 - ! @ ?DUP 0=
15    UNTIL   R>  DP ! ;          ' FORGET  NFA  FENCE !

SCREEN #96
 0 \ TEXT  PP  COPY
 1 HEX
 2 : TEXT     ( c -> )
 3    HERE C/L 8 +  BLANK  WORD  BL OVER
 4    DUP @  +  4+  C!    PAD  C/L   4+  CMOVE ;
 5
 6 : PP      ( n -> <text> )
 7    DUP  FFFFFFF0 AND
 8    ABORT" Off screen."
 9    0 TEXT   PAD  4+ SWAP
10    SCR @ <LINE>  CMOVE  UPDATE  ;
11
12 : COPY
13    SWAP BLOCK 4- !  UPDATE ;
14 DECIMAL
15
```

```
SCREEN #97
 0 \  DOUBLE PRECISION MATH --- DU<  D-  D0=  D>
 1 HEX
 2 : DU<   ( ud1 ud2 -> flag )  ( MVP-FORTH UTILITY )
 3   D>R  80000000 +  DR>  80000000 +  D<  ;
 4
 5 : D-   ( d1 d2 -> d3 )  ( MVP-FORTH UTILITY )
 6   DNEGATE  D+  ;
 7
 8 : D0=   ( d1 -> flag )
 9   OR 0= ;
10
11 : D>   ( d1 d2 -> flag )    ( MVP-FORTH UTILITY )
12   DSWAP D< ;
13 DECIMAL
14
15

SCREEN #98
 0 \  DOUBLE PRECISION MATH --- D,  DCONSTANT  DMAX  DMIN
 1 DECIMAL
 2 : D,   ( d -> )
 3    SWAP ,  ,  ;
 4
 5 : DCONSTANT   ( d -> )  ( compiling )
 6               ( -> d )  ( executing )
 7   CREATE  D,
 8    DOES>  D@ ;
 9
10 : DMAX   ( d1 d2 -> d3 )  ( MVP-FORTH UTILITY )
11   DOVER DOVER  D<  IF  DSWAP  THEN   DDROP  ;
12
13 : DMIN   ( d1 d2 -> d3 )  ( MVP-FORTH UTILITY )
14   DOVER DOVER  D< NOT   IF  DSWAP  THEN   DDROP ;
15

SCREEN #99
 0 \ PAUSE
 1 DECIMAL
 2 : PAUSE   ?TERMINAL
 3    IF   100000 0 DO LOOP
 4         BEGIN ?TERMINAL UNTIL
 5         170000 0 DO   LOOP
 6    THEN ;
 7
 8
 9
10
11
12
13
14
15

SCREEN #100
 0 \  DVARIABLE  ID.  INDEX
 1 DECIMAL
 2 : DVARIABLE   ( -> )        ( compiling )
 3               ( -> addr )   ( executing )
 4   CREATE  8 ALLOT ;
```

```
 5
 6 : ID.   COUNT  255 AND  OVER  + SWAP
 7    DO  I C@ 127 AND EMIT  LOOP  32 EMIT ;
 8
 9 : INDEX   ( First  Last -> )
10    1+ SWAP
11    DO  CR  I 4 .R  2 SPACES  I BLOCK 64  -TRAILING TYPE
12       PAUSE  ?TERMINAL IF  LEAVE THEN
13    LOOP  CR ;
14
15

SCREEN #101
 0 \  BOARD FORTH  ---   VLIST
 1 HEX
 2 : VLIST  ( -> )
 3      81  OUT !   CONTEXT  @ @
 4    BEGIN     C/L  OUT @ -  OVER  C@  03F AND   4 + <
 5       IF   CR   0 OUT !   THEN
 6       DUP  ID.  SPACE  SPACE   4- @  DUP
 7       0=   PAUSE   ?TERMINAL  OR
 8    UNTIL  DROP ;
 9
10
11 DECIMAL
12
13
14
15

SCREEN #102
 0 \  .S
 1 VARIABLE .SS
 2 : .SL 1 .SS ! ;
 3
 4 : .SR 0 .SS ! ;
 5
 6 .SR
 7
 8 : .S  DEPTH
 9      IF
10       .SS @ IF  1  DEPTH 1+  2   ELSE  -1  2  DEPTH 1-  THEN
11       DO I PICK U. DUP +LOOP DROP
12      ELSE ." Empty Stack. "
13      THEN CR ;
14
15

SCREEN #103
 0 \  'S  -TEXT  2!  2@  2CONSTANT  2DROP  2DUP
 1 DECIMAL
 2 : 'S   SP@ ;
 3
 4 : -TEXT  DDUP +  SWAP DO DROP 1+ DUP 1- C@  I C@ - DUP
 5    IF  DUP ABS / LEAVE  THEN LOOP  SWAP DROP ;
 6
 7 : 2!     D! ;
 8 : 2@     D@ ;
 9
10 : 2CONSTANT    DCONSTANT ;
11 : 2DROP        DDROP ;
```

```
12 : 2DUP          DDUP ;
13
14
15

SCREEN #104
 0 \  2OVER 2SWAP 2VARIABLE  >BINARY  >TYPE
 1 DECIMAL
 2 : 2OVER         DOVER        ;
 3
 4 : 2SWAP         DSWAP        ;
 5
 6 : 2VARIABLE     DVARIABLE    ;
 7
 8 : >BINARY       CONVERT      ;
 9
10 : >TYPE      ."  >TYPE USED IN MULTIPROGRAMMED SYSTEMS ONLY. " ;
11 IMMEDIATE
12
13
14
15

SCREEN #105
 0 \  ERASE  FLUSH  H  OCTAL  U.R  ['']
 1 DECIMAL
 2 : ERASE         0 FILL       ;
 3
 4 : FLUSH         SAVE-BUFFERS ;
 5
 6 : H             DP           ;
 7
 8 : OCTAL         8 BASE !     ;
 9
10 : U.R           0  SWAP  D.R ;
11
12 : ['']          ?COMP   [COMPILE]  '  ;  IMMEDIATE
13
14
15

SCREEN #106
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15

SCREEN #107
 0
 1
```

```
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15

SCREEN #108
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15

SCREEN #109
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15

SCREEN #110
 0 ( internal use only variables and holding places )
 1 DECIMAL
 2 VARIABLE TEMP-ADDR  ( temporary address storage )
 3
 4 VARIABLE TSUML 16 ALLOT     ( division temp storage - quotient )
 5 TSUML 8 + CONSTANT TSUMH  ( division temp storage - remainder )
 6
 7 VARIABLE TSUMQ 8 ALLOT ( quad precision temp storage )
 8
```

```
 9 VARIABLE TEMP-CARRY  ( temporary carry storage )
10 VARIABLE SIGDIG   ( number significant digits )
11      7 SIGDIG !
12 VARIABLE TERM  4 ALLOT  ( temp storage for transcendentals )
13 VARIABLE FTERM 4 ALLOT  ( temp storage for transcendentals )
14 VARIABLE <?MODE>  ( numeric input mode )
15

SCREEN #111
 0 ( shift/rotate operations  LSL  ASR  LSR )
 1 HEX   ( may be replaced with CODE definitions )
 2 \  : 2/   2  / ;
 3
 4 : LSL   ( n1 -> n2 )
 5   2*    ;
 6
 7 \  : ASR   ( n1 -> n2 )
 8 \    FFFFFFFE AND  2/  ;
 9
10 \  : LSR   ( n1 -> n2 )
11 \    ASR  7FFFFFFF AND  ;
12
13 DECIMAL
14
15

SCREEN #112
 0 ( SGN  U>  )
 1 DECIMAL
 2 : SGN    ( n -> signum.of.n )
 3   DUP  IF   0<  IF -1 ELSE 1 THEN THEN ;
 4
 5 : U>     ( un1 un2 -> flag )
 6   SWAP U< ;
 7
 8
 9
10
11
12
13
14
15

SCREEN #113
 0 ( RRC  RLC  ADC  )
 1 HEX     ( may be replaced with CODE definitions )
 2 \ : RRC    ( n1 carry.in -> n2 carry.out )
 3 \   SWAP  DUP LSR  ROT
 4 \   IF  8000 OR  THEN    SWAP 1 AND ;
 5
 6 \ : RLC    ( n1 carry.in -> n2 carry.out )
 7 \  SWAP  DUP LSL  ROT
 8 \  IF  1 OR  THEN     SWAP 0< ;
 9
10 \ : ADC  ( n1 n2 carry.in -> n3  carry.out )
11 \  >R  0 ROT 0  D+  R> IF  1 0 D+  THEN ;
12
13 DECIMAL
14
15
```

```
SCREEN #114
 0 ( single precision multiple rotates  ASRN  LSRN  LSLN )
 1 DECIMAL
 2 : ASRN    ( n1 count -> n2 )
 3   DUP 0>  IF    0 DO  ASR  LOOP
 4           ELSE  DROP  THEN ;
 5
 6 \ : LSRN    ( n1 count -> n2 )
 7 \   DUP 0>  IF    0 DO  LSR  LOOP
 8 \           ELSE  DROP  THEN  ;
 9
10 \ : LSLN    ( n1 count -> n2 )
11 \   DUP 0>  IF    0 DO  LSL  LOOP
12 \         ELSE  DROP  THEN  ;
13
14
15

SCREEN #115
 0 ( conversions   S->Q  D->Q  D->S  Q->S  Q->D  )
 1 DECIMAL
 2 \ : S->Q    ( n -> q )
 3 \   S->D  DUP DUP ;
 4
 5 \ : D->Q    ( d -> q )
 6 \   S->D  DUP ;
 7
 8 : D->S    ( d -> n )
 9   DROP ;
10
11 \ : Q->S    ( q -> n )
12 \   DDROP    DROP ;
13
14 \ : Q->D    ( q -> d )
15 \   DDROP ;

SCREEN #116
 0 ( double stack ops  DOVER  DSWAP  DROT  )
 1 DECIMAL      ( may be replaced with CODE definitions )
 2 \ : DOVER    ( d1 d2 -> d1 d2 d1 )  ( MVP-FORTH UTILITY )
 3 \   4_PICK  4_PICK  ;
 4
 5 \ : DSWAP    ( d1 d2 -> d2 d1 )  ( MVP-FORTH UTILITY )
 6 \   4 ROLL  4 ROLL ;
 7
 8 \ : DROT   ( d1 d2 d3 -> d2 d3 d1 )  ( MVP-FORTH UTILITY )
 9 \    6 ROLL  6 ROLL ;
10
11
12
13
14
15

SCREEN #117
 0 ( double precision stack ops  D@  D>R  DR>  DR@  )
 1 DECIMAL      ( may be replaced with CODE definitions )
 2 \ : D>R    ( d -> )
 3 \   R>  SWAP >R  SWAP >R  >R  ;
```

```
 5 \ : DR>   ( -> d )
 6 \   R>  R> R>  ROT  >R  ;
 7
 8 : DR@    ( -> d )
 9   I' J  ;
10
11
12
13
14
15

SCREEN #118
 0 ( double precision stack ops  DPICK  DROLL  D? )
 1 HEX
 2 : DPICK    ( d1 .. dn count -> d1 .. dn dm )
 3   DUP 1 <  ABORT" DPICK ARGUMENT < 1"
 4   2*  DUP 1+   PICK  SWAP PICK  ;
 5
 6 : DROLL    ( d1 .. dn count -> d1 ..<omit dm>.. dn dm )
 7   DUP 1 <  ABORT" DROLL ARGUMENT < 1"
 8   2*  DUP 1+  ROLL  SWAP ROLL  ;
 9
10 : D?    ( addr -> )
11   D@  D.  ;
12
13 DECIMAL
14
15

SCREEN #119
 0 EXIT  ( D,  DCONSTANT DVARIABLE )
 1 DECIMAL
 2 : D,    ( d -> )
 3    SWAP ,  ,  ;
 4
 5 : DCONSTANT    ( d -> )  ( compiling )
 6                ( -> d )  ( executing )
 7   CREATE  D,
 8    DOES>  D@ ;
 9
10 : DVARIABLE    ( -> )        ( compiling )
11                ( -> addr )   ( executing )
12   CREATE  8 ALLOT ;
13
14
15

SCREEN #120
 0 ( D-  D+!     )
 1 DECIMAL
 2 \ : D-    ( d1 d2 -> d3 )   ( MVP-FORTH UTILITY )
 3 \   DNEGATE  D+  ;
 4
 5 : D+!    ( d1 addr -> )
 6    DUP >R  D@ D+  R> D!  ;
 7
 8
 9
10
11
```

```
12
13
14
15

SCREEN #121
 0 ( double precision rotates/add  DADC  DRRC  DRLC )
 1 HEX
 2 : DADC   ( d1 d2 carry.in -> d3 carry.out )
 3    SWAP  >R  ROT  >R
 4    ADC  DR>   ROT   ADC ;
 5
 6 : DRRC  ( d1 carry.in -> d2 carry.out )
 7   RRC  SWAP >R  RRC  R> SWAP  ;
 8
 9 : DRLC   ( d1 carry.in -> d2 carry.out )
10   SWAP >R  RLC  R> SWAP  RLC ;
11
12 DECIMAL
13
14
15

SCREEN #122
 0 ( double precision shifts  DASR  DLSR  DLSL  )
 1 HEX
 2 : DASR   ( d1 -> d2 )
 3  SWAP LSR  OVER 1 AND
 4  IF 80000000 OR  THEN    SWAP ASR  ;
 5
 6 \ : DLSR   ( d1 -> d2 )
 7 \    DASR  7FFF AND  ;
 8
 9 \ : DLSL   ( D1 -> D2 )
10 \   DDUP D+ ;
11
12 DECIMAL
13
14
15

SCREEN #123
 0 ( double precision multiple shifts  DASRN  DLSRN  DLSLN )
 1 DECIMAL
 2 : DASRN   ( d1 n -> d2 )
 3   DUP 0>  IF    0 DO  DASR LOOP
 4           ELSE  DROP  THEN  ;
 5
 6 : DLSRN   ( d1 n -> d2 )
 7   DUP 0>  IF    0 DO  DLSR  LOOP
 8           ELSE  DROP  THEN  ;
 9
10 : DLSLN   ( d1 n -> d2 )
11   DUP 0>  IF    0 DO  DLSL  LOOP
12           ELSE  DROP  THEN  ;
13
14
15

SCREEN #124
 0 ( DOR  DAND  DXOR  BYTESWAP  )
 1 DECIMAL
```

```
 2 : DOR    ( d1 d2 -> d3 )
 3  >R  SWAP >R  OR  DR> OR  ;
 4
 5 : DAND    ( d1 d2 -> d3 )
 6  >R  SWAP >R  AND  DR>  AND ;
 7
 8 : DXOR    ( d1 d2 -> d3 )
 9  >R  SWAP >R  XOR  DR> XOR ;
10
11 \ : BYTESWAP  ( n1 -> n2 )
12 \    DUP  8 DLSRN DROP  ;
13
14
15

SCREEN #125
 0 \ double precision comparisons  D0<  D0>  D0=  D>  D= )
 1 DECIMAL
 2 : D0<    ( d1 -> flag )
 3   SWAP_DROP  0<  ;
 4
 5 : D0>    ( d1 -> flag )
 6   DNEGATE D0<  ;
 7
 8 \ : D0=    ( d1 -> flag )
 9 \   OR 0= ;
10
11 \ : D>    ( d1 d2 -> flag )     ( MVP-FORTH UTILITY )
12 \   DSWAP D< ;
13
14 \ : D=    ( d1 d2 -> flag )   ( MVP-FORTH UTILITY )
15 \     D- D0= ;

SCREEN #126
 0 ( DMAX  DMIN  DU<  DU> )
 1 HEX
 2 \ : DMAX    ( d1 d2 -> d3 )   ( MVP-FORTH UTILITY )
 3 \   DOVER DOVER  D<  IF  DSWAP  THEN   DDROP  ;
 4
 5 \ : DMIN    ( d1 d2 -> d3 )   ( MVP-FORTH UTILITY )
 6 \   DOVER DOVER  D< NOT   IF  DSWAP  THEN   DDROP ;
 7
 8 \ : DU<    ( ud1 ud2 -> flag )   ( MVP-FORTH UTILITY )
 9 \   D>R  80000000 +  DR>  80000000 +  D<  ;
10
11 : DU>    ( ud1 ud2 -> flag )
12   DSWAP DU<  ;
13
14 DECIMAL
15

SCREEN #127
 0 ( quad precision stack ops  QDROP QDUP QOVER QSWAP QROT )
 1 DECIMAL      ( may be replaced with CODE definitions )
 2 : QDROP    ( q -> )
 3   DDROP DDROP  ;
 4
 5 : QDUP    ( q1 -> q1 q1 )
 6   DOVER DOVER  ;
 7
 8 : QOVER    ( q1 q2 -> q1 q2 q1 )
```

```
 9   4 DPICK  4 DPICK  ;
10
11 : QSWAP   ( q1 q2 -> q2 q1 )
12   4 DROLL  4 DROLL  ;
13
14 : QROT    ( q1 q2 q3 -> q2 q3 q1 )
15   6 DROLL  6 DROLL  ;

SCREEN #128
 0 ( quad precision store and fetch  Q!  Q@  )
 1 DECIMAL      ( may be replaced with CODE definitions )
 2 \ Bytes stored in low to high order, offset 0 = first byte
 3 : Q!    ( q addr -> )
 4    DUP  >R  8 + D!  R>  D! ;
 5
 6 : Q@    ( addr -> q )
 7   DUP  D@  ROT  8
 8
 9
10
11
12
13
14
15

SCREEN #129
 0 ( quad return stack operations  Q>R  QR>  QR@  )
 1 DECIMAL      ( may be replaced with CODE definitions )
 2 : Q>R   ( q -> )
 3   R> TEMP-ADDR !  D>R D>R  TEMP-ADDR @  >R  ;
 4
 5 : QR>   ( -> q )
 6   R> TEMP-ADDR !  DR> DR>  TEMP-ADDR @  >R  ;
 7
 8 : QR@  ( -> Q )
 9   R>  TEMP-ADDR !  DR> DR>
10   QDUP  D>R D>R  TEMP-ADDR @  >R  ;
11
12
13
14
15

SCREEN #130
 0 ( quad logical ops  QOR  QAND  QXOR )
 1 DECIMAL
 2 : QOR         ( q1 q2 -> q3 )
 3    D>R  DSWAP   D>R  DOR    DR>   DR> DOR  ;
 4
 5 : QAND        ( q1 q2 -> q3 )
 6    D>R  DSWAP   D>R  DAND   DR>   DR> DAND  ;
 7
 8 : QXOR        ( q1 q2 -> q3 )
 9    D>R  DSWAP   D>R  DXOR   DR>   DR> DXOR  ;
10
11
12
13
14
15
```

```
SCREEN #131
 0 ( QADC )
 1 DECIMAL      ( may be replaced with CODE definitions )
 2 : QADC    ( q1 q2 carry.in -> q3 carry.out )
 3   TEMP-CARRY !  D>R  DSWAP D>R  TEMP-CARRY @    DADC
 4    DR> DR>  5 ROLL  DADC  ;
 5
 6
 7
 8
 9
10
11
12
13
14
15

SCREEN #132
 0 ( Q+  QNEGATE  Q-    DM+  )
 1 DECIMAL
 2 : Q+    ( q1 q2 -> qsum )
 3    0 QADC  DROP  ;
 4
 5 : QNEGATE      ( q1 -> -q1 )
 6   -1. -1. QXOR     1. 0. Q+    ;
 7
 8 : Q-    ( q1 q2 -> q3 )
 9   QNEGATE Q+  ;
10
11 \ : DM+     ( q1 d2 -> q3 )
12 \   D->Q Q+  ;
13
14
15

SCREEN #133
 0 ( Q+!  Q+-  QABS )
 1 DECIMAL
 2 : Q+!    ( q addr -> )
 3   DUP >R  Q@ Q+  R> Q!  ;
 4
 5 : Q+-    ( q1 n2 -> q3 )
 6   0<  IF  QNEGATE  THEN  ;
 7
 8 : QABS    ( q1 -> q2 )
 9   DUP Q+-  ;
10
11
12
13
14
15

SCREEN #134
 0 ( QASR  QLSR  QLSL )
 1 HEX
 2 : QASR    ( q1 -> q2 )
 3   DSWAP  DLSR  4_PICK  1 AND
 4   IF  80000000 OR  THEN     DSWAP DASR  ;
```

```
 5
 6 : QLSR    ( q1 -> q2 )
 7   QASR  7FFFFFFF AND ;
 8
 9 : QLSL    ( q1 -> q2 )
10    QDUP Q+  ;
11
12 DECIMAL
13
14
15
```

```
SCREEN #135
 0 EXIT ( QASRN  QLSRN  QLSLN  )
 1 DECIMAL
 2 : QASRN    ( q1 n2 -> q3 )
 3   DUP 0>  IF    0 DO  QASR  LOOP
 4           ELSE  DROP  THEN  ;
 5
 6 : QLSRN    ( q1 n2 -> q3 )
 7   DUP 0>  IF    0 DO  QLSR  LOOP
 8           ELSE  DROP  THEN  ;
 9
10 : QLSLN    ( q1 n2 -> q3 )
11   DUP 0>  IF    0 DO  QLSL  LOOP
12           ELSE  DROP  THEN  ;
13
14
15
```

```
SCREEN #136
 0 ( basic double multiplication  DU* )
 1 DECIMAL      ( may be replaced with CODE definitions )
 2 : DU*    ( ud1 ud2 -> uq3 )
 3   ( adds 4 partial products to get result )
 4   OVER  5 PICK  U*  D>R  SWAP  3 PICK  U*  D>R
 5   ROT  OVER  U*  D>R  U*  0 0  DSWAP
 6   0  DR>  0 Q+  0 DR>  0 Q+  DR>  0 0 Q+  ;
 7
 8
 9
10
11
12
13
14
15
```

```
SCREEN #137
 0 ( double precision multiplication  D*  DM* )
 1 DECIMAL
 2 : D*    ( d1 d2 -> d3 )
 3   DU* DDROP  ;
 4
 5 : DM*    ( d1 d2 -> q3 )
 6   DUP  4_PICK  XOR >R
 7   DABS  DSWAP DABS  DU*  R> Q+-  ;
 8
 9
10
```

```
11
12
13
14
15

SCREEN #138
 0 ( double precision unsigned division  DU/MOD )
 1 HEX      ( may be replaced with CODE definitions )
 2 : DU/MOD    ( uq1 ud2 -> ud3 ud4 )
 3   >R >R TSUML 0! R> R>
 4   DDUP  DNEGATE  TSUMH D+!  41 >R
 5   BEGIN   TSUMH 4+ @  0<  R> 1- DUP >R
 6   WHILE  TSUML 0@  0LSL  TSUML 0!
 7      IF  DDUP
 8      ELSE  DDUP DNEGATE    1 TSUML +!  THEN
 9      TSUMH  D+!  REPEAT
10   R> DROP
11   TSUML D@  3_PICK NOT   DRLC DROP  ROT
12   IF  DSWAP  TSUMH D@ D+
13   ELSE   DSWAP DDROP   TSUMH D@  THEN  DSWAP ;
14
15 DECIMAL

SCREEN #139
 0 ( double precision mixed division  DM/MOD  DM/  D/MOD  D*/MOD )
 1 DECIMAL
 2 : DM/MOD    ( uq1 ud2 -> ud3 uq4 )
 3   D>R  0 0 DR@  DU/MOD  DR> DSWAP D>R  DU/MOD DR>  ;
 4
 5 : DM/    ( q1 d2 -> d3 d4 )
 6   3_PICK >R  D>R  QABS  R> R@  DABS  DU/MOD
 7   R> R@  XOR D+-  DSWAP  R>  D+-  DSWAP  ;
 8
 9 : D/MOD    ( d1 d2 -> d3 d4 )
10   D>R  S->D  S->D  DR>  DM/  ;
11
12 : D*/MOD    ( d1 d2 d3 -> d4 d5 )
13   D>R  DM*  DR>  DM/  ;
14
15

SCREEN #140
 0 ( D/  D*/  DMOD )
 1 DECIMAL
 2 : D/    ( d1 d2 -> d3 )
 3   D/MOD  DSWAP DDROP  ;
 4
 5 : D*/    ( d1 d2 d3 -> d4 )
 6   D*/MOD  DSWAP DDROP  ;
 7
 8 : DMOD    ( d1 d2 -> d3 )
 9   D/MOD DDROP  ;
10
11
12
13
14
15
```

```
SCREEN #141
 0 ( DINP#  DMODE  ?MODE )
 1 DECIMAL
 2 : DINP#     ( -> D )   ( "state-smart" word )
 3   BL WORD  <NUMBER>
 4   [COMPILE] DLITERAL  ;  IMMEDIATE
 5
 6 : DMODE   ( -> )
 7   0 <?MODE> !
 8   ' <NUMBER> CFA      'NUMBER !
 9   ' <INTERPRET> CFA   'INTERPRET !
10   R> DROP  INTERPRET  ;      IMMEDIATE
11
12 : ?MODE    ( -> flag=0,1,2 )
13   <?MODE> @ ;
14 DMODE
15

SCREEN #142
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15

SCREEN #143
 0 EXIT  \ Note:  32-bit floating point number is 1 stack element!!
 1 \ And temporary numbers are d-nums!!!
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15

SCREEN #144
 0 EXIT \ floating point return stack operations  F>R  FR>  FR@  )
 1 DECIMAL      ( may be replaced with CODE definitions )
 2 : F>R    ( f1 -> )
 3   R>  SWAP >R  SWAP >R   >R ;
 4
 5 : FR>    ( -> f1 )
```

```
 6   R> R> R>  ROT  >R  ;
 7
 8 : FR@   ( -> f1 )
 9   I' J ;
10
11
12
13
14
15

SCREEN #145
 0 EXIT ( floating aliases -- NOT NEEDED -- 32 BITS=SINGLE PREC )
 1 DECIMAL
 2 : F@            @  ;
 3 : F!            !  ;
 4 : FDROP         DROP  ;
 5 : FDUP          DUP  ;
 6 : FSWAP         SWAP  ;
 7 : FOVER         OVER  ;
 8 : FROT          ROT  ;
 9 : FPICK         PICK  ;
10 : FROLL         ROLL  ;
11 : FCONSTANT     CONSTANT  ;
12 : FVARIABLE     VARIABLE  ;
13 : FLITERAL      [COMPILE] LITERAL  ;   IMMEDIATE
14 : F,            ,  ;
15

SCREEN #146
 0 ( FABS  F0=  and temporary absolute values & zero test )
 1 HEX
 2 : FABS   ( f1 -> f2 )
 3   7FFFFFFF AND  ;
 4
 5 : F0=    ( f1 -> flag )
 6   FABS  0=  ;
 7
 8 : TABS   ( t1 -> t2 )
 9    >R FABS R>  ;
10
11 : T0=    ( t1 -> t1 flag )
12   ( Note:  does NOT remove t1 from stack !!! )
13   >R  DUP F0= R> SWAP  ;
14
15 DECIMAL

SCREEN #147
 0 ( various T operations )
 1 HEX
 2 \ : TDROP   ( t1 -> )
 3 \   DDROP  ;
 4
 5 : CHK0   ( t1 -> t2 )'   ( forces clean zero )
 6    T0=  IF  DDROP 0 0  THEN  ;
 7
 8 : TNEGATE   ( t1 -> t2 )
 9    >R  80000000 XOR  R> CHK0  ;
10
11 : T+-   ( t1 n2 -> t3 )
```

```
12   0< IF  TNEGATE  THEN  ;
13
14 DECIMAL
15

SCREEN #148
 0 EXIT ( TSWAP  TOVER  TDUP  T@  T! )
 1 DECIMAL      ( may be replaced with CODE definitions )
 2 : TSWAP   ( t1 t2 -> t2 t1 )
 3    DSWAP ;
 4
 5 : TOVER   ( t1 t2 -> t1 t2 t1 )
 6    DOVER ;
 7
 8 : TDUP    ( t1 -> t1 t1 )
 9    DDUP ;
10
11 : T@    ( addr -> t1 )
12    D@ ;
13
14 : T!    ( t1 addr -> )
15    D!  ;

SCREEN #149
 0 ( floating point to temporary conversion )
 1 HEX      ( may be replaced with CODE definitions )
 2 : F->T   ( f1 -> t2 )
 3   DUP  F0=
 4   IF  DROP  0 0
 5   ELSE  DUP 7F800000 AND  17 LSRN  7F -  >R  DUP >R
 6     7FFFFF AND  800000 OR  7 LSLN  R> R>
 7     SWAP T+-  THEN  ;
 8
 9 DECIMAL
10
11
12
13
14
15

SCREEN #150
 0 \ 32-bit normalization of mantissa
 1 HEX      ( may be replaced with CODE definitions )
 2 : UTNORMALIZE   ( ut1 -> ut2 )
 3   OVER  IF  <UNORM>    ELSE  DROP 0 THEN ;
 4 \ >R  DUP 0<
 5 \ IF   ( shift right )  LSR R>  1+
 6 \ ELSE  DUP 0=
 7 \     IF   ( zero )  R> DROP 0
 8 \     ELSE  ( shift left )
 9 \         \   BEGIN   DUP  40000000 AND NOT
10 \         \   WHILE   LSL  R> 1- >R    REPEAT
11 \         \   R>
12 \     THEN
13 \ THEN  ;
14
15 DECIMAL
```

```
SCREEN #151
 0 ( temporary to floating point conversion )
 1 HEX      ( may be replaced with CODE definitions )
 2 : T->F    ( t1 -> f2 )
 3   CHKO  OVER ( sign ) >R  7F +  ( exponent )  >R   FABS
 4   DUP 0=
 5   IF  R> R> DDROP
 6   ELSE  ( round )  40 + R> UTNORMALIZE
 7         17 LSLN  7F800000 AND
 8         SWAP 7 LSRN   7FFFFF AND OR
 9      R> 80000000 AND OR      THEN ;
10
11 DECIMAL
12
13
14
15

SCREEN #152
 0 ( floating to temp conversions and quad normalize )
 1 HEX
 2 : SEPARATE2   ( f1 f2 -> t1 t2 )
 3   >R  F->T   R>  F->T  ;
 4
 5 : UDNORMALIZE  ( ud1 n2 -> ud3 n4 )
 6   >R DDUP D0=  IF R> DROP 0   ELSE  R> <UDNORM> THEN ;
 7 \ >R  DUP  0<
 8 \ IF  DLSR  R>  1+
 9 \ ELSE DDUP  D0=
10 \    IF  ( zero ) R> DROP 0
11 \    ELSE ( shift left )
12 \      BEGIN  DUP  40000000 AND  NOT
13 \      WHILE  DLSL  R> 1- >R  REPEAT   R>  THEN  THEN ;
14 \      R>  <UDNORM>   THEN ;
15  DECIMAL

SCREEN #153
 0 ( temporary floating point addition )
 1 DECIMAL      ( may be replaced with CODE definitions )
 2 : T+    ( t1 t2 -> t3 )
 3   T0=  IF  DDROP
 4   ELSE  DSWAP  T0=
 5      IF  DDROP
 6      ELSE  ROT   DDUP >
 7         IF  D>R SWAP DR>  ELSE  SWAP  THEN
 8         OVER >R  4_PICK  DUP >R  4_PICK  XOR  0< >R
 9         4 ROLL FABS  4 ROLL  FABS  DSWAP  -  32 MIN LSRN
10         DUP 0= IF R> DROP 0 >R  THEN     R>
11         IF R> 0<    IF SWAP THEN
12            NEGATE  0 ADC NOT     >R ABS R>
13         ELSE  + R>  0<   THEN
14         R> SWAP >R UTNORMALIZE R>   IF TNEGATE THEN
15   THEN THEN   CHKO  ;

SCREEN #154
 0 ( temporary floating point multiplication )
 1 HEX          ( may be replaced with CODE definitions )
 2 : T*    ( t1 t2 -> t3 )
 3   ROT  +  OVER  4_PICK  XOR  D>R
 4   FABS  SWAP  FABS  U* R> 2+ UDNORMALIZE
```

```
 5   T0= NOT  IF
 6     >R  ( round )  80000000 0 D+  R> UDNORMALIZE  THEN
 7   >R   SWAP_DROP  DR>  T+-   CHKO  ;
 8
 9 DECIMAL
10
11
12
13
14
15

SCREEN #155
 0 ( temporary floating point division )
 1 HEX      ( may be replaced with CODE definitions )
 2 : T/    ( t1 t2 -> t3 )
 3   ( check for divide by zero )  T0=
 4   IF  DSWAP DDROP  ( result is zero )
 5   ELSE  ROT SWAP  - OVER  4_PICK  XOR   >R >R
 6      FABS  SWAP  FABS  0 SWAP  DLSR ROT
 7      U/MOD SWAP_DROP
 8      R> 1- UTNORMALIZE  R>  T+-  THEN   CHKO  ;
 9
10 DECIMAL
11
12
13
14
15

SCREEN #156
 0 ( F+  F*  F/  F2/  F2* )
 1 DECIMAL
 2 : F+    ( f1 f2 -> f3 )
 3   SEPARATE2  T+ T->F  ;
 4
 5 : F*    ( f1 f2 -> f3 )
 6   SEPARATE2  T* T->F  ;
 7
 8 : F/    ( f1 f2 -> f3 )
 9   SEPARATE2  T/ T->F  ;
10
11 : F2/    ( f1 -> f2 )
12   F->T  1-  T->F  ;
13
14 : F2*    ( f1 -> f2 )
15   F->T  1+  T->F  ;

SCREEN #157
 0 ( FNEGATE  F-  F+! )
 1 HEX
 2 : FNEGATE    ( f1 -> f2 )
 3   DUP IF  80000000 XOR  THEN  ;
 4 DECIMAL
 5 : T-    ( t1 t2 -> t3 )
 6   TNEGATE  T+  ;
 7
 8 : F-    ( f1 f2 -> f3 )
 9   FNEGATE F+  ;
10
```

```
11 : F+!    ( f1 addr -> )
12   DUP   >R  @ F+  R>  !  ;
13
14 : T+!   ( t1 addr -> )
15   DUP >R  D@ T+  R>  D! ;
```

```
SCREEN #158
 0 ( conversions  D->F )
 1 HEX      ( may be replaced with CODE definitions )
 2 : N->T    ( n1 -> t2 )    ( floats the integer value )
 3   DUP >R  ABS  01E UTNORMALIZE   R> T+-  CHKO  ;
 4
 5 : N->F    ( n1 -> f2 )
 6    N->T  T->F  ;
 7
 8 : T->N    ( t1 -> n2 )
 9   CHKO  01E -  DUP ABS  01E >
10   IF  DDROP 0
11   ELSE  DUP 0>
12      IF  LSLN
13      ELSE  OVER  D>R  7FFFFFFF AND  R> ABS LSRN
14         R> +-  THEN THEN ;
15 DECIMAL
```

```
SCREEN #159
 0 ( conversions  F->N     )
 1 DECIMAL
 2 : F->N    ( f1 -> d2 )
 3    F->T T->N  ;
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

```
SCREEN #160
 0 \ floating comparisons  F0< F0>   F= F< F> )
 1 DECIMAL
 2 : F0<          0<  ;
 3
 4 : F0>          0>  ;
 5
 6 : F=           =   ;
 7
 8 : F<    ( f1 f2 -> flag )
 9   F- F0<  ;
10
11 : F>    ( f1 f2 -> flag )
12   SWAP  F<  ;
13
14
15
```

```
SCREEN #161
 0 ( FMIN  FMAX  F+-  FSGN  )
 1 DECIMAL
 2 : FMIN   ( f1 f2 -> f3 )
 3   OVER OVER  F>  IF  SWAP  THEN  DROP  ;
 4
 5 : FMAX   ( f1 f2 -> f3 )
 6   OVER OVER  F<  IF  SWAP  THEN  DROP  ;
 7
 8 : F+-   ( f1 n2 -> f3 )
 9   0<  IF  FNEGATE  THEN  ;
10
11 : FSGN   ( f1 -> n2 )
12   SGN  ;
13
14
15


SCREEN #162
 0 ( integer & fractional portion  INT  FRAC  REM )
 1 HEX
 2 : INT   ( f1 -> f2 )
 3   F->T  DUP  01F <  IF  T->N N->T THEN   T->F  ;
 4
 5 : TFRAC   ( t1 -> t2 )
 6   DUP 01F <  IF  DDUP T->N N->T T-
 7              ELSE  DDROP 0 0  THEN  ;
 8
 9 : FRAC   ( f1 -> f2 )
10   F->T TFRAC  T->F  ;
11
12 : REM   ( f1 f2 -> f3 )
13    OVER OVER   F/   INT  F* F-  ;
14
15 DECIMAL

SCREEN #163
 0 ( floating point input  FCONVERT )
 1 DECIMAL
 2 : FCONVERT   ( f1 addr2 -> f3 addr4 )
 3   >R F->T   BASE @  N->T  DSWAP R>
 4   BEGIN  1+ DUP >R   C@ BASE @ DIGIT
 5   WHILE  >R DOVER T*  R> N->T  T+   DPL @ 1+
 6      IF  1 DPL +!  THEN      R>
 7   REPEAT  DSWAP DDROP T->F  R> ;
 8
 9
10
11
12
13
14
15

SCREEN #164
 0 ( floating point input  <FNUMBER> )
 1 HEX
 2 : <FNUMBER>   ( addr1 -> f2 )
 3   3 +  0 SWAP  DUP 1+  C@  2D = DUP >R  ABS  +  -1 DPL !
 4   FCONVERT  DUP  C@ BL >
```

```
 5   IF DUP  C@ 2E =  NOT  ABORT" NOT RECOGNIZED"
 6      0 DPL !  FCONVERT DUP C@ BL >  ABORT" NOT RECOGNIZED"
 7   THEN    DROP R>
 8   IF  FNEGATE  THEN     F->T     DPL @
 9   BEGIN   DUP 0>
10   WHILE  >R  BASE @ N->T  T/  R> 1-
11   REPEAT          DROP T->F  ;
12
13 DECIMAL
14
15

SCREEN #165
 0 \  <FINTERPRET>   - FLOATING POINT VERSION OF <INTERPRET>
 1 HEX
 2 : <FINTERPRET>   ( -> )
 3    BEGIN  -FIND
 4       IF  DUP  STATE @  U<
 5          IF   DUP  10000000 AND
 6             IF SWAP @ OPCODE, ELSE  SWAP CALL,  THEN
 7             8000000 AND  IF  DON'T-DISTURB THEN
 8          ELSE  4000000 AND  ABORT" POISON WORD"  EXECUTE THEN
 9       ELSE   HERE 4+  NUMBER  DPL @  1+
10          IF    [COMPILE] LITERAL
11          ELSE  F->N   [COMPILE] LITERAL    THEN
12       THEN  ?STACK    AGAIN  ;
13
14 DECIMAL
15

SCREEN #166
 0 \ FMODE  FINP# )
 1 DECIMAL
 2 : FMODE   ( -> )
 3    2 <?MODE> !
 4    ' <FNUMBER> CFA     'NUMBER !
 5    ' <FINTERPRET> CFA  'INTERPRET !
 6    R> DROP   INTERPRET  ;     IMMEDIATE
 7
 8 : FINP#   ( -> f )  ( "state-smart" word )
 9   BL WORD  <FNUMBER>
10   [COMPILE] LITERAL  ;     IMMEDIATE
11
12
13
14
15

SCREEN #167
 0 ( floating point input  TCONVERT )
 1 DECIMAL
 2 : TCONVERT   ( t1 addr2 -> t3 addr4 )
 3   >R    BASE @  N->T  DSWAP R>
 4   BEGIN  1+ DUP >R   C@ BASE @ DIGIT
 5   WHILE  >R DOVER T*  R> N->T  T+   DPL @ 1+
 6      IF  1 DPL +!  THEN      R>
 7   REPEAT  DSWAP DDROP   R> ;
 8
 9
10
```

```
11
12
13
14
15

SCREEN #168
 0 ( floating point input  <TNUMBER> )
 1 HEX
 2 : <TNUMBER>   ( addr1 -> t2 )
 3   3 +  0 SWAP  DUP 1+  C@  2D = DUP >R  ABS  +  -1 DPL !
 4   FCONVERT  DUP  C@ BL >
 5   IF DUP  C@ 2E =  NOT  ABORT" NOT RECOGNIZED"
 6      0 DPL !  FCONVERT DUP C@ BL >  ABORT" NOT RECOGNIZED"
 7   THEN    DROP R>
 8   IF  FNEGATE  THEN     F->T     DPL @
 9   BEGIN   DUP 0>
10   WHILE  >R  BASE @ N->T  T/  R> 1-
11   REPEAT          DROP  ;
12
13 DECIMAL
14
15

SCREEN #169
 0 ( TINP# )
 1 DECIMAL
 2 : TINP#   ( -> f )  ( "state-smart" word )
 3   BL WORD  <TNUMBER>
 4   STATE @   IF
 5      [COMPILE] DLITERAL
 6   THEN ;     IMMEDIATE
 7
 8
 9
10 : T.   TINP# 100000 T* T->N . ;
11
12
13
14
15

SCREEN #170
 0 \ TSLOG2 calculation  0.5 <= x <= 1.0  taylor series ln/ln2 )
 1 DECIMAL
 2 : TSLOG2     ( t1 -> t2 )        ( 0.5 <= t1 <=1 )
 3    TABS   TO= NOT  IF
 4    DDUP  TINP# -1 T+   DSWAP   TINP# 1 T+  T/
 5    DDUP DDUP T* TERM D!
 6    DDUP TINP# 2.885390082 ( 2/ln2 ) T*  FTERM D! TERM D@ T*
 7    DDUP TINP# .9617966939 T*  ( 2/3ln2 ) FTERM T+!  TERM D@ T*
 8    DDUP TINP# .5770780164 T*  ( 2/5ln2 ) FTERM T+!  TERM D@ T*
 9    DDUP TINP# .4121985831 T*  ( 2/7ln2 ) FTERM T+!  TERM D@ T*
10    DDUP TINP# .3205988980 T*  ( 2/9ln2 ) FTERM T+!  TERM D@ T*
11    DDUP TINP# .2623081893 T*  ( 2/11ln2 ) FTERM T+! TERM D@ T*
12    DDUP TINP# .2219530832 T*  ( 2/13ln2 ) FTERM T+! TERM D@ T*
13    DDUP TINP# .1923593388 T*  ( 2/15ln2 ) FTERM T+! TERM D@ T*
14    TINP# .1697288283 T*  ( 2/17ln2 ) FTERM D@ T+  THEN ;
15
```

```
SCREEN #171
 0 ( chebyshev TS2**   calculation   0.0 <= x <= 1.0 )
 1 DECIMAL
 2 : TS2**      ( t1 -> t2 )        ( 0 <= t1 <=1 )
 3   TABS   DDUP TERM D!     TINP# 1.0  FTERM D!    ( x**0 )
 4   DDUP TINP# .69314718 T*  ( x**1 ) FTERM T+!   TERM D@ T*
 5   DDUP TINP# .24022636 T*  ( x**2 ) FTERM T+!   TERM D@ T*
 6   DDUP TINP# .055505294 T*  ( x**3 ) FTERM T+!   TERM D@ T*
 7   DDUP TINP# .0096135358 T*  ( x**4 ) FTERM T+!   TERM D@ T*
 8   DDUP TINP# .0013429811 T*  ( x**5 ) FTERM T+!   TERM D@ T*
 9   DDUP TINP# .00014299401 T*  ( x**6 ) FTERM T+!   TERM D@ T*
10   TINP# .000021651724 T*  ( x**7 ) FTERM D@ T+  ;
11
12
13
14
15

SCREEN #172
 0 ( basic logarithm  LOG2 )
 1 DECIMAL
 2 : TLOG2   ( t1 -> t2 )
 3   T0=  OVER 0< OR   DUP
 4   IF ( BAD OPERATION )    DROP TABS T0= THEN
 5   NOT   IF  >R  -1 TSLOG2
 6      R> 1+  N->T  T+  THEN  ;
 7
 8 : LOG2   ( f1 -> f2 )
 9    F->T  TLOG2  T->F  ;
10
11
12
13
14
15

SCREEN #173
 0 ( basic exponentiation  2** )
 1 HEX
 2 : T2**   ( t1 -> t2 )
 3   OVER >R ( sign )
 4   TABS  DDUP T->N  DUP >R  N->T  T-
 5   TS2**   R>  +   R> 0<
 6   IF  40000000  0 ( t=1.0 )    DSWAP T/  THEN  ;
 7
 8 : 2**   ( f1 -> f2 )
 9   F->T  T2**  T->F  ;
10
11 DECIMAL
12
13
14
15

SCREEN #174
 0 ( LOGB  F** )
 1 DECIMAL
 2 : TLOGB   ( t1 tbase -> t3 )   ( log to a base )
 3   DSWAP  TLOG2  DSWAP  TLOG2  T/  ;
 4
```

```
 5 : LOGB    ( f1 f2 -> f3 )
 6   SEPARATE2  TLOGB  T->F  ;
 7
 8 : F**    ( f1 f2 -> f3 )
 9   SWAP  SEPARATE2  TLOG2  T*  T2**    T->F  ;
10
11
12
13
14
15

SCREEN #175
 0 ( LOG  LN  10**  E**  )
 1 HEX
 2 : LOG    ( f1 -> f2 )
 3   41200000    ( f=10.0 )  LOGB  ;
 4
 5 : LN    ( f1 -> f2 )
 6   402DF854    ( f=2.7182818 )    LOGB ;
 7
 8 : 10**    ( f1 -> f2 )
 9   41200000    ( f=10.0 ) SWAP  F**  ;
10
11 : E**    ( f1 -> f2 )
12   402DF854    ( f=2.7182818 )  SWAP  F**  ;
13
14 DECIMAL
15

SCREEN #176
 0 ( ROOT  **2  1/X  EXP  )
 1 DECIMAL
 2 : ROOT    ( f1 f2 -> f3 )
 3   SEPARATE2 DSWAP  TLOG2  DSWAP T/  T2**  T->F  ;
 4
 5 : **2    ( f1 -> f2 )
 6   DUP F*  ;
 7
 8 : 1/X    ( f1 -> f2 )
 9   FINP# 1  SWAP F/  ;
10
11 : EXP    ( f1 n -> f2 )
12   N->F  BASE @ N->F  SWAP  F**  F*  ;
13
14
15

SCREEN #177
 0 ( floating to alpha conversion  F->ME )
 1 HEX
 2 : F->ME    ( f1 -> d2 n3 )
 3   F->T  TO= NOT
 4   IF  OVER >R TABS  BASE @ N->T DSWAP DOVER
 5      TLOGB  DDUP TFRAC  DSWAP  T->N  3_PICK 0<
 6      IF  1- >R  40000000 0 ( t=1.0 ) T+  R> THEN
 7      >R  ( log almost 1? ) BASE @ DUP
 8      SIGDIG @  1  DO  OVER *  LOOP
 9      -1 +  >R N->T R>  N->T
10      DSWAP TLOGB TFRAC  DOVER  T- DROP F0<
```

```
11       IF  DDROP 0 0  R> 1+ >R  THEN
12       SIGDIG @ 1-  N->T
13       T+ DSWAP TLOG2 T* T2**   40000000  -1  ( 0.5 )
14       T+ T->N S->D  R> R> SWAP  >R D+- R>  ELSE 0 THEN ;
15 DECIMAL

SCREEN #178
 0 ( exponent print  F.ER  F.E  )
 1 HEX
 2 : F.ER   ( f1 n2 -> )
 3   >R  F->ME   DUP >R ABS  S->D
 4   <# #S  DDROP  R> SIGN  BL HOLD
 5   50 HOLD  58 HOLD  45 HOLD  BL HOLD   DUP >R DABS
 6   SIGDIG @  1  DO  #  LOOP
 7   2E HOLD  #  R> SIGN  #>
 8   R>  OVER -  SPACES TYPE  ;
 9
10 : F.E   ( f1 -> )
11   0 F.ER  SPACE  ;
12
13 DECIMAL
14
15

SCREEN #179
 0 ( fixed point numeric printing  <F.>  F.XR  F.X )
 1 HEX
 2 : <F.>   ( d1 n2 -> addr3 n4 n5 )
 3   SIGDIG @ - 1+  NEGATE DUP  0 MAX  >R OVER >R
 4   >R  DABS   <#  R@ 0 MAX    ?DUP
 5   IF  0  DO  #  LOOP  THEN
 6   2E HOLD    R@ 0<
 7   IF  R@ ABS  0  DO  30 HOLD  LOOP  THEN
 8   R> DROP   #S  R> SIGN  #>   R>  ;
 9
10 : F.XR   ( f1 n2 -> )
11   >R F->ME  <F.> DROP  R> OVER -  SPACES TYPE  ;
12
13 : F.X   ( f1 -> )
14   0 F.XR  SPACE  ;
15 DECIMAL

SCREEN #180
 0 ( aligned fixed point print  F.AR  F.A )
 1 HEX
 2 : F.AR   ( f1 n2 n3 -> )
 3   >R  0 MAX  >R  F->ME   SIGDIG @  OVER - 1-   R@ -  DUP 0>
 4   IF  SWAP >R  N->F  10**  F2/  F->N 0  D+
 5      R>  <F.>  R> - -
 6      ELSE  DROP <F.>  R>   DDUP <
 7         IF  SWAP -  3_PICK 3_PICK +  OVER  30 FILL +
 8         ELSE  DDROP   THEN     THEN
 9   R>  OVER -  SPACES TYPE  ;
10
11 : F.A   ( f1 n2 -> )
12   0 F.AR  SPACE  ;
13
14 DECIMAL
15
```

```
SCREEN #181
 0 ( smart floating point prints  F.R  F.  F?  )
 1 HEX
 2 : F.R   ( f1 n2 -> )
 3   >R  DUP F->T   DUP 17 >  SWAP -4 <  OR
 4   IF  DROP R>  F.ER
 5   ELSE  DROP  F->ME  <F.>  DROP
 6      BEGIN  DDUP + 1- C@     30 =
 7      WHILE  1-  REPEAT
 8   R>  OVER -  SPACES TYPE  THEN  ;
 9
10 : F.   ( FP# -> )
11   0 F.R SPACE ;
12
13 : F?     @ F. ;
14
15 DECIMAL

SCREEN #182
 0 ( SQRT  FACTORIAL )
 1 DECIMAL
 2 : SQRT   ( f1 -> f2 )
 3   FABS    F->T  DDUP TERM D!
 4   ( initial approximation is f1/2 )     ASR 1-
 5   5 0  DO  TERM D@  DOVER T/  T+  1-  LOOP
 6   T->F  ;
 7
 8 : FACTORIAL   ( f1 -> f2 )
 9   FINP# 1  SWAP  F->N  ABS
10   1+ 2 DO   I  N->F  F*   LOOP  ;
11
12
13
14
15

SCREEN #183
 0 ( PI  PI/2  PI/4  2*PI  RAD->DEG  DEG->RAD )
 1 DECIMAL
 2 FINP# 3.14159265 CONSTANT PI
 3 PI   F2/ CONSTANT PI/2
 4 PI/2 F2/ CONSTANT PI/4
 5 PI   F2* CONSTANT 2*PI
 6
 7 : RAD->DEG   ( f1 -> f2 )
 8   FINP# 57.29577951 F* ;
 9
10 : DEG->RAD   ( f1 -> f2 )
11   FINP# 0.0174532925 F*  ;
12
13
14
15

SCREEN #184
 0 ( chebyshev sine routine )
 1 DECIMAL
 2 : TSIN   ( t1 -> t2 )
 3   ( input from -pi/4 TO pi/4 )
 4   DDUP  DDUP T* TERM D!
```

```
 5   DDUP TINP# .9999999995 ( x**1 ) T*  FTERM D!
 6   TERM D@ T*  DDUP  TINP# -.1666666663 ( x**3 ) T* FTERM T+!
 7   TERM D@ T*  DDUP  TINP# .008333328785 ( x**5 ) T* FTERM T+!
 8   TERM D@ T*  DDUP  TINP# -.0001983920268 ( x**7 ) T* FTERM T+!
 9   TERM D@ T*  TINP# .000002717349463 ( x**9 )  T* FTERM D@ T+ ;
10
11
12
13
14
15

SCREEN #185
 0 ( chebyshev cosine routine )
 1 DECIMAL
 2 : TCOS   ( t1 -> t2 )
 3   ( input from -pi/4 to pi/4 )
 4   DDUP T* DDUP TERM D!
 5   DDUP TINP# -.4999999943 ( x**2 ) T*  FTERM D!
 6   TERM D@ T*  DDUP  TINP# .0416666167 ( x**4 ) T* FTERM T+!
 7   TERM D@ T*  DDUP  TINP# -0.001388661862 ( x**6 ) T* FTERM T+!
 8   TERM D@ T*  TINP# .00002437988031 ( x**8 )  T* FTERM D@ T+
 9   TINP# 1.0  T+ ;
10
11
12
13
14
15

SCREEN #186
 0 ( full range cosine and sine  <COS>  <SIN>  )
 1 DECIMAL
 2 : <COS>  ( FP# -> FP# )
 3   FABS  2*PI REM   DUP PI F>
 4   IF  FNEGATE  2*PI F+  THEN
 5   DUP  PI/2  F>
 6   IF  FNEGATE  PI F+  -1 >R   ELSE   1 >R  THEN
 7   DUP  PI/4 F>
 8   IF  FNEGATE  PI/2 F+  F->T TSIN T->F
 9   ELSE  F->T TCOS T->F  THEN
10   R> F+-  ;
11
12 : <SIN>   ( f1 -> f2 )
13   FNEGATE   PI/2 F+   <COS>  ;
14
15

SCREEN #187
 0 ( derived trig functions  <TAN>  <SEC>  <CSC>  <COT> )
 1 DECIMAL
 2 : <TAN>   ( f1 -> f2 )
 3   DUP <SIN>  SWAP <COS>  F/  ;
 4
 5 : <SEC>   ( f1 -> f2 )
 6   <COS>  1/X  ;
 7
 8 : <CSC>   ( f1 -> f2 )
 9   <SIN>  1/X  ;
10
```

```
11 : <COT>    ( f1 -> f2 )
12   DUP <COS>  SWAP <SIN>  F/  ;
13
14
15

SCREEN #188
 0 ( trig functions  COS  SIN  TAN  SEC  CSC  COT  )
 1 DECIMAL
 2 : COS    ( f1 -> f2 )
 3   DEG->RAD  <COS>  ;
 4 : SIN    ( f1 -> f2 )
 5   DEG->RAD  <SIN>  ;
 6 : TAN    ( f1 -> f2 )
 7   DEG->RAD  <TAN>  ;
 8
 9 : SEC    ( f1 -> f2 )
10   DEG->RAD  <SEC>  ;
11 : CSC    ( f1 -> f2 )
12   DEG->RAD  <CSC>  ;
13 : COT    ( f1 -> f2 )
14   DEG->RAD  <COT>  ;
15

SCREEN #189
 0 ( chebyshev arctangent routine )
 1 DECIMAL
 2 : TATAN    ( t1 -> t2 )
 3   ( input from -1 to 1 )
 4   DDUP  DDUP T* TERM D!
 5   DDUP TINP# .9999999842 ( x**1 ) T*  FTERM D!
 6   TERM D@ T*  DDUP  TINP# -.3333306679 ( x**3 ) T* FTERM T+!
 7   TERM D@ T*  DDUP  TINP# .1999248354 ( x**5 ) T* FTERM T+!
 8   TERM D@ T*  DDUP  TINP# -.1420257041 ( x**7 ) T* FTERM T+!
 9   TERM D@ T*  DDUP  TINP# .1063675406 ( x**9 ) T* FTERM T+!
10   TERM D@ T*  DDUP  TINP# -.0749544546 ( x**11 ) T* FTERM T+!
11   TERM D@ T*  DDUP  TINP# .0425876076 ( x**13 ) T* FTERM T+!
12   TERM D@ T*  DDUP  TINP# -.0160050306 ( x**15 ) T* FTERM T+!
13   TERM D@ T*  TINP# .0028340643 ( x**17 )  T* FTERM D@ T+ ;
14
15

SCREEN #190
 0 ( basic inverse trig function  <ATAN> <ATAN2>  )
 1 DECIMAL
 2 : <ATAN>    ( f1 -> f2 )
 3   DUP FABS  FINP# 1  F>
 4   IF  DUP >R  FINP# 1  SWAP F/  F->T TATAN T->F
 5      FNEGATE   PI/2  R> F+-  F+
 6   ELSE  F->T TATAN T->F  THEN  ;
 7
 8 : <ATAN2>    ( fx fy -> f3 )
 9   OVER F0=
10   IF  >R DROP DROP  PI/2  R> F+-
11   ELSE  OVER F0<
12      IF  DUP F0<  >R  SWAP F/  <ATAN>  PI  R>
13         IF  F-  ELSE  F+  THEN
14      ELSE   SWAP F/  <ATAN>   THEN THEN ;
15
```

```
SCREEN #191
 0 ( derived inverse trig  <ASIN>  <ACOS>  <ACOT>  )
 1 DECIMAL
 2 : <ASIN>   ( f1 -> f2 )
 3   DUP FABS  FINP# 1  F=
 4   IF  SWAP DROP  PI/2  ROT F+-
 5   ELSE  FINP# 1  OVER  **2  F-  SQRT F/  <ATAN>  THEN  ;
 6
 7 : <ACOS>   ( f1 -> f2 )
 8   <ASIN> FNEGATE   PI/2 F+  ;
 9
10 : <ACOT>   ( f1 -> f2 )
11   <ATAN> FNEGATE   PI/2 F+  ;
12
13
14
15

SCREEN #192
 0 ( derived inverse trig  <ASEC>  <ACSC>  )
 1 DECIMAL
 2 : <ASEC>   ( f1 -> f2 )
 3   DUP  **2  FINP# -1  F+  SQRT  <ATAN> SWAP   FO<
 4   IF  PI F- THEN   ;
 5
 6 : <ACSC>  ( FP# -> FP# )
 7   DUP  FABS  FINP# 1  F=
 8   IF  SWAP DROP  PI/2  ROT F+-
 9   ELSE  DUP  **2 FINP# -1  F+
10      SQRT 1/X    <ATAN> SWAP   FO<
11      IF  PI F- THEN  THEN  ;
12
13
14
15

SCREEN #193
 0 ( trig functions  ACOS  ASIN  ATAN  ASEC  ACSC  ACOT  ATAN2 )
 1 DECIMAL
 2 : ACOS   ( f1 -> f2 )
 3   <ACOS>  RAD->DEG  ;
 4 : ASIN   ( f1 -> f2 )
 5   <ASIN>  RAD->DEG  ;
 6 : ATAN   ( f1 -> f2 )
 7   <ATAN>  RAD->DEG  ;
 8 : ASEC   ( f1 -> f2 )
 9   <ACSC>  RAD->DEG  ;
10 : ACSC   ( f1 -> f2 )
11   <ACSC>  RAD->DEG  ;
12 : ACOT   ( f1 -> f2 )
13   <ACOT>  RAD->DEG  ;
14 : ATAN2   ( fx fy -> f3 )
15   <ATAN2>  RAD->DEG  ;

SCREEN #194
 0 ( <P->R>  <R->P>  P->R  R->P  )
 1 DECIMAL
 2 : <P->R>   ( frad fang -> fx fy )
 3   OVER OVER  <SIN>  F* >R
 4   <COS>  F* R>  ;
```

```
 5
 6 : <R->P>    ( fx fy -> frad fang )
 7   OVER OVER  <ATAN2> >R
 8   **2 SWAP  **2 F+  SQRT R>  ;
 9
10 : P->R    ( frad fang -> fx fy )
11   DEG->RAD  <P->R>  ;
12
13 : R->P    ( fx fy -> frad fang )
14   <R->P>  RAD->DEG ;
15
```

SCREEN #195

```
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

SCREEN #196

```
 0 ( Calculation speed and accuracy benchmark )
 1 ( Adapted from BYTE magazine )
 2 (   Volume 10, No. 5, MAY 1985, page 282 )
 3 FINF# 2.71828 CONSTANT FA
 4 FINF# 3.14159 CONSTANT FB
 5
 6 : CALCULATIONS    ( -> )
 7    FINF# 1.0
 8    5000 0 DO     FA F*  FB  F*
 9                  FA F/  FB  F/   LOOP
10   ( CR  ." DONE"  CR )  ."      ERROR="  FINF# 1.0  F- F.  ;
11
12
13
14
15
```

SCREEN #197

```
 0 \ Trig table test  -- This is a good stress of the board!
 1 DECIMAL
 2 : TEST-TRIG  ( -> )
 3    CR
 4    ."   X       SIN(X)       COS(X)              TAN(X)" CR
 5    ."  ====  ==========  ==========        ==========" CR
 6    361 0 DO
 7       I N->F  DUP   5 F.R
 8       DUP  SIN  7 12 F.AR
 9       DUP  COS  7 12 F.AR
10            TAN  7 19 F.AR    CR
```

```
11        ?TERMINAL ABORT" ..BREAK.."
12     LOOP ;
13
14
15


SCREEN #198
 0 \ Stress test based on trig functions - 1
 1 DECIMAL
 2 : STRESS-COMPARE    ( D1 N2 D3 N4 -> )
 3     >R  ROT >R
 4     D=  R> R> =  AND
 5     NOT  ABORT" BAD COMPARE" ;
 6
 7
 8
 9
10
11
12
13
14
15

SCREEN #199
 0 \ Stress test based on trig functions - 2
 1 DECIMAL
 2 : STRESS  ( -> )
 3     1000 0 DO CR I .
 4     361 0 DO    ." ."
 5        I N->F  DUP   F->ME  4 PICK F->ME  STRESS-COMPARE
 6          DUP SIN  F->ME  4 PICK  SIN F->ME  STRESS-COMPARE
 7          DUP COS  F->ME  4 PICK  COS F->ME  STRESS-COMPARE
 8          DUP TAN  F->ME  4 PICK  TAN F->ME  STRESS-COMPARE
 9        DROP
10        ?TERMINAL ABORT" ..BREAK.."
11     LOOP LOOP ;
12
13
14
15
```

APPENDIX A

PART III

DIAGNOSTICS & TESTS

PROGRAM LISTING

```
SCREEN #1
 0 INDEX --- CPU/32 TEST SOURCE                          PHIL KOOPMAN JR.
 1 LIBRARY VERSION    FILE: TEST.4TH                  LAST UPDATE: 6/1/87
 2
 3 BETA TEST VERSION  (C) COPYRIGHT 1987
 4 BY PHIL KOOPMAN JR.
 5
 6 LOAD      SOURCE
 7 SCREEN    SCREENS    CONTENTS
 8 =======   ========   ========
 9   2       2 - 66     COMPREHENSIVE STATIC TEST SUITE FROM HOST
10                      NOTE: TRASHES CPU/32 MEMORY!!!!!!!!!!!!!!
11   3       70 - 75    RAM TEST FROM CPU/32
12  ...      80 - 124   STRESS TEST FROM CPU32
13
14
15

SCREEN #2
 0 \  STATIC TEST SUITE FOR FORTH BOARD -- LOAD FROM HOST PC
 1 DECIMAL          MATH      STOP      : TASKZZ ;
 2 VARIABLE DISP-PASS    1 DISP-PASS !
 3 : ZZ     ( DVALUE  DVALUE  N -> )
 4      >R  QDUP  D=
 5      IF    DISP-PASS @  IF    ." PASS "  THEN    QDROP  R> DROP 0
 6      ELSE    CR ."  *** ERROR #" R> .
 7               ." EXPECTED: "  DU.
 8              ."     ACTUAL: "  DU.  KEY 13 = CR THEN
 9    ?TERMINAL  OR  ABORT" BREAK" ;
10 : QZ 66 4 DO CR  1 0 DO CR J 3 .R ." :"  J LOAD
11    ?TERMINAL ABORT" BREAK" LOOP LOOP ;          CR CR
12 ( ERROR MESSAGE HERE IF BREAK KEY USED ==> )  QZ
13 FORGET TASKZZ  CR CR
14 LOAD-ALL CR CR ." Test complete." CR CR ." Type:  CPU32" CR
15 ."        3 LOAD" CR ." for stress testing." CR CR

SCREEN #3
 0 \  STRESS TEST LOAD SCREEN   -- ** DO A LOAD-ALL FIRST!! **
 1 DECIMAL
 2 10 CONSTANT REPS                100000 CONSTANT #TIMES
 3 CR CR ." STRESS TEST.  ITERATIONS = "    REPS #TIMES * . CR CR
 4 70 75 THRU      \ Load RAM TEST
 5 : FORGET-MARKER ;
 6 CR ." Executing 1 RAM test." CR
 7 HEX CR RAM-TEST CR CR DECIMAL    \ Run single RAM test
 8 CR ." Executing stress tests." CR
 9 80 124 THRU      \ Load & execute other stress tests
10 FORGET  FORGET   RKER
11 CR ." Executing ' REPS . ." RAM tests." CR
```

```
12 HEX REPEAT-RAM-TEST  DECIMAL    \ Run N RAM-TESTS
13 CR ." Executing trig function stress test." CR
14 ." Press any key to stop." CR
15 STRESS                          \ Run trig stress test
```

```
SCREEN #4
 0 ." TEST MIR REGISTER" CR
 1 HEX    MATH
 2 00000000. MIR!  MIR@     00000000.    1 ZZ
 3 FFFFFFFF. MIR!  MIR@     FFFFFFFF.    2 ZZ
 4 12345678. MIR!  MIR@     12345678.    3 ZZ
 5 EDCBA987. MIR!  MIR@     EDCBA987.    4 ZZ
 6
 7
 8 DECIMAL
 9
10
11
12
13
14
15
```

```
SCREEN #5
 0 ." TEST STATUS REGISTER" CR
 1 HEX          MATH
 2 : XXA   0 ::  DEST=STATUS  ;SET   X!  ;
 3 : XXB   STATUS  0 ;
 4
 5 0000. XXA   XXB  OFF. DAND     0000.        1 ZZ
 6 00FF. XXA   XXB  OFF. DAND     00FF.        2 ZZ
 7 0012. XXA   XXB  OFF. DAND     0012.        3 ZZ
 8 00ED. XXA   XXB  OFF. DAND     00ED.        4 ZZ
 9
10 DECIMAL
11
12
13
14
15
```

```
SCREEN #6
 0 ." TEST PC REQUEST REGISTER" CR
 1 HEX          MATH
 2 : XXC   DROP  PCREQ  ;
 3 : XXD   0 :: SOURCE=FLAGS ;SET X@ ;
 4
 5 0000. XXC   XXD  OFF. DAND     0000.        1 ZZ
 6 005A. XXC   XXD  OFF. DAND     005A.        2 ZZ
 7 00A5. XXC   XXD  OFF. DAND     00A5.        3 ZZ
 8 00DE. XXC   XXD  OFF. DAND     00DE.        4 ZZ
 9
10 DECIMAL
11
12
13
14
15
```

```
SCREEN #7
 0 ." DP TEST" CR
 1 HEX          MATH
 2 : XXE   0 :: DEST=DP  ;SET  X!  ;
 3 : XXF   1 :: SOURCE=DP  ;SET  X@  ;
 4 5A5.  XXE  XXF   00FFF. DAND   05A5.  1 ZZ
 5 A5A.  XXE  XXF   00FFF. DAND   0A5A.  2 ZZ
 6 000. XXE  0 :: INC[DP] ;DO     XXF  0FFF. DAND   0001.  3 ZZ
 7 00F. XXE  0 :: INC[DP] ;DO     XXF  0FFF. DAND   0010.  4 ZZ
 8 0FF. XXE  0 :: INC[DP] ;DO     XXF  0FFF. DAND   0100.  5 ZZ
 9 FFF. XXE  0 :: INC[DP] ;DO     XXF  0FFF. DAND   0000.  6 ZZ
10 000. XXE  0 :: DEC[DP] ;DO     XXF  0FFF. DAND   0FFF.  7 ZZ
11 100. XXE  0 :: DEC[DP] ;DO     XXF  0FFF. DAND   00FF.  8 ZZ
12 010. XXE  0 :: DEC[DP] ;DO     XXF  0FFF. DAND   000F.  9 ZZ
13 001. XXE  0 :: DEC[DP] ;DO     XXF  0FFF. DAND   0000. 0A ZZ
14 DECIMAL
15  .

SCREEN #8
 0 ." RP TEST"  CR
 1 HEX          MATH
 2 : XXG   0 :: DEST=RP  ;SET  X!  ;
 3 : XXH   1 :: SOURCE=RP  ;SET  X@  ;
 4
 5 05A5.  XXG  XXH   00FFF. DAND   05A5.  1 ZZ
 6 0A5A.  XXG  XXH   00FFF. DAND   0A5A.  2 ZZ
 7 0000. XXG  0 :: INC[RP] ;DO     XXH  0FFF. DAND   0001.  3 ZZ
 8 000F. XXG  0 :: INC[RP] ;DO     XXH  0FFF. DAND   0010.  4 ZZ
 9 00FF. XXG  0 :: INC[RP] ;DO     XXH  0FFF. DAND   0100.  5 ZZ
10 0FFF. XXG  0 :: INC[RP] ;DO     XXH  0FFF. DAND   0000.  6 ZZ
11 0000. XXG  0 :: DEC[RP] ;DO     XXH  0FFF. DAND   0FFF.  7 ZZ
12 0100. XXG  0 :: DEC[RP] ;DO     XXH  0FFF. DAND   00FF.  8 ZZ
13 0010. XXG  0 :: DEC[RP] ;DO     XXH  0FFF. DAND   000F.  9 ZZ
14 0001. XXG  0 :: DEC[RP] ;DO     XXH  0FFF. DAND   0000. 0A ZZ
15 DECIMAL

SCREEN #9
 0 ." TEST DLO REGISTER (STORE/FETCH)" CR
 1 HEX
 2 : XXJ   0 ::  DEST=DLO  ;SET   X!  ;
 3 : XXK   1 :: SOURCE=DLO  ;SET  X@ ;
 4
 5 \ : XX@ HEX BEGIN  0 DISP-PASS !
 6 00000000. XXJ  XXK   00000000. ( D= NOT IF ." X"  THEN ) 1 ZZ
 7 FFFFFFFF. XXJ  XXK   FFFFFFFF. ( D= NOT IF ." Y"  THEN ) 2 ZZ
 8 12345678. XXJ  XXK   12345678. ( D= NOT IF ." Z"  THEN ) 3 ZZ
 9 EDCBA987. XXJ  XXK   EDCBA987. ( D= NOT IF ." W"  THEN ) 4 ZZ
10 \   ?TERMINAL UNTIL 1 DISP-PASS ! DECIMAL  ;
11
12
13
14 DECIMAL
15

SCREEN #10
 0 ." TEST DHI REGISTER (STORE/FETCH)" CR
 1 HEX
 2 : XXL   0 :: ALU=B  DEST=DHI   ;SET  X!  ;
 3 : XXM   1 :: SOURCE=DHI        ;SET  X@  ;
 4
```

```
 5 55555555. XXL   XXM    55555555.        1 ZZ
 6 AAAAAAAA. XXL   XXM    AAAAAAAA.        2 ZZ
 7 23456789. XXL   XXM    23456789.        3 ZZ
 8 DCBA9876. XXL   XXM    DCBA9876.        4 ZZ
 9
10
11
12 DECIMAL
13
14
15

SCREEN #11
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15

SCREEN #12
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15

SCREEN #13
 0 ." ALU CONSTANTS" CR
 1 HEX
 2 : XXW  ( A B -> )
 3       0 ::  DEST=DLO  ;SET  X!
 4       1 ::  ALU=B DEST=DHI  ;SET  X!
 5       2 ::  SOURCE=DLO  ;
 6
 7 : XXX   ( ->  RESULT )
 8                              DEST=DHI  ;DO
 9       3 ::  SOURCE=DHI             ;SET  X@   ;
10
```

```
11 0 :: SOURCE=DHI   ALU=0  DEST=DHI ;DO  X@   00000000.  1 ZZ
12 1 :: SOURCE=DHI   ALU=-1 DEST=DHI ;DO  X@   FFFFFFFF.  2 ZZ
13
14 DECIMAL
15

SCREEN #14
 0 ." ALU A & B SIDE PASS THRU" CR
 1 HEX
 2 84512347.  00000000. XXW  ALU=A      XXX  84512347.  1 ZZ
 3 84512347.  FFFFFFFF. XXW  ALU=A      XXX  84512347.  2 ZZ
 4 7BAEDCB8.  FFFFFFFF. XXW  ALU=A      XXX  7BAEDCB8.  3 ZZ
 5
 6 00000000.  84512347. XXW  ALU=B      XXX  84512347.  4 ZZ
 7 FFFFFFFF.  84512347. XXW  ALU=B      XXX  84512347.  5 ZZ
 8 FFFFFFFF.  7BAEDCB8. XXW  ALU=B      XXX  7BAEDCB8.  6 ZZ
 9
10 FFFFFFFF.  7BAEDCB8. XXW  ALU=A      XXX  FFFFFFFF.  7 ZZ
11
12
13 DECIMAL
14
15

SCREEN #15
 0 ." ALU A & B SIDE COMPLEMENT" CR
 1 HEX
 2 00000001.    0.   XXW  ALU=notA  XXX   FFFFFFFE.  1 ZZ
 3 FFFFFFFE.    0.   XXW  ALU=notA  XXX   00000001.  2 ZZ
 4 12345678.   00.   XXW  ALU=notA  XXX   EDCBA987.  3 ZZ
 5 12345678.   FFFFFFFF.  XXW  ALU=notA  XXX   EDCBA987.  4 ZZ
 6 0.   00000000.   XXW  ALU=notB  XXX   FFFFFFFF.  5 ZZ
 7 0.   FFFFFFFF.   XXW  ALU=notB  XXX   00000000.  6 ZZ
 8 0.   12345678.   XXW  ALU=notB  XXX   EDCBA987.  7 ZZ
 9 -1.  12345678.   XXW  ALU=notB  XXX   EDCBA987.  8 ZZ
10 DECIMAL
11
12
13
14
15

SCREEN #16
 0 ." ALU NOR & NAND" CR
 1 HEX
 2 FFFFFFFF. FFFFFFFF.  XXW  ALU=AnorB   XXX   00000000.  1 ZZ
 3 00000000. FFFFFFFF.  XXW  ALU=AnorB   XXX   00000000.  2 ZZ
 4 FFFFFFFF. 00000000.  XXW  ALU=AnorB   XXX   00000000.  3 ZZ
 5 00000000. 00000000.  XXW  ALU=AnorB   XXX   FFFFFFFF.  4 ZZ
 6 6432ACE4. 43221252.  XXW  ALU=AnorB   XXX   98CD4109.  5 ZZ
 7
 8 00000000. 00000000.  XXW  ALU=AnandB  XXX   FFFFFFFF.  6 ZZ
 9 FFFFFFFF. 00000000.  XXW  ALU=AnandB  XXX   FFFFFFFF.  7 ZZ
10 00000000. FFFFFFFF.  XXW  ALU=AnandB  XXX   FFFFFFFF.  8 ZZ
11 FFFFFFFF. FFFFFFFF.  XXW  ALU=AnandB  XXX   00000000.  9 ZZ
12 43239835. 436587AE.  XXW  ALU=AnandB  XXX   BCDE7FDB.  A ZZ
13
14 DECIMAL
15
```

```
SCREEN #17
 0 ." ALU XOR & XNOR" CR
 1 HEX
 2 FFFFFFFF. FFFFFFFF.  XXW  ALU=AxorB   XXX   00000000.  1 ZZ
 3 00000000. FFFFFFFF.  XXW  ALU=AxorB   XXX   FFFFFFFF.  2 ZZ
 4 FFFFFFFF. 00000000.  XXW  ALU=AxorB   XXX   FFFFFFFF.  3 ZZ
 5 00000000. 00000000.  XXW  ALU=AxorB   XXX   00000000.  4 ZZ
 6 6432ACE4. 43221252.  XXW  ALU=AxorB   XXX   2710BEB6.  5 ZZ
 7
 8 00000000. 00000000.  XXW  ALU=AxnorB  XXX   FFFFFFFF.  6 ZZ
 9 FFFFFFFF. 00000000.  XXW  ALU=AxnorB  XXX   00000000.  7 ZZ
10 00000000. FFFFFFFF.  XXW  ALU=AxnorB  XXX   00000000.  8 ZZ
11 FFFFFFFF. FFFFFFFF.  XXW  ALU=AxnorB  XXX   FFFFFFFF.  9 ZZ
12 43239835. 436587AE.  XXW  ALU=AxnorB  XXX   FFB9E064.  A ZZ
13
14 DECIMAL
15

SCREEN #18
 0 ." ALU OR & AND" CR
 1 HEX
 2 FFFFFFFF. FFFFFFFF.  XXW  ALU=AorB    XXX   FFFFFFFF.  1 ZZ
 3 00000000. FFFFFFFF.  XXW  ALU=AorB    XXX   FFFFFFFF.  2 ZZ
 4 FFFFFFFF. 00000000.  XXW  ALU=AorB    XXX   FFFFFFFF.  3 ZZ
 5 00000000. 00000000.  XXW  ALU=AorB    XXX   00000000.  4 ZZ
 6 6432ACE4. 43221252.  XXW  ALU=AorB    XXX   6732BEF6.  5 ZZ
 7
 8 00000000. 00000000.  XXW  ALU=AandB   XXX   00000000.  6 ZZ
 9 FFFFFFFF. 00000000.  XXW  ALU=AandB   XXX   00000000.  7 ZZ
10 00000000. FFFFFFFF.  XXW  ALU=AandB   XXX   00000000.  8 ZZ
11 FFFFFFFF. FFFFFFFF.  XXW  ALU=AandB   XXX   FFFFFFFF.  9 ZZ
12 43239835. 436587AE.  XXW  ALU=AandB   XXX   43218024.  A ZZ
13
14 DECIMAL
15

SCREEN #19
 0 ." ALU A+1 & CARRY PROPAGATE TEST" CR
 1 HEX
 2 12345678.  0. XXW  ALU=A+1  XXX   12345679.  1 ZZ
 3 00000000.  0. XXW  ALU=A+1  XXX   00000001.  2 ZZ
 4 0000000F.  0. XXW  ALU=A+1  XXX   00000010.  3 ZZ
 5 000000FF.  0. XXW  ALU=A+1  XXX   00000100.  4 ZZ
 6 00000FFF.  0. XXW  ALU=A+1  XXX   00001000.  5 ZZ
 7 0000FFFF.  0. XXW  ALU=A+1  XXX   00010000.  6 ZZ
 8 000FFFFF.  0. XXW  ALU=A+1  XXX   00100000.  7 ZZ
 9 00FFFFFF.  0. XXW  ALU=A+1  XXX   01000000.  8 ZZ
10 0FFFFFFF.  0. XXW  ALU=A+1  XXX   10000000.  9 ZZ
11 FFFFFFFF.  0. XXW  ALU=A+1  XXX   00000000. 10 ZZ
12
13 DECIMAL
14
15

SCREEN #20
 0 ." ALU A+1 & BORROW PROPAGATE TEST" CR
 1 HEX
 2 12345678.  0. XXW  ALU=A-1  XXX   12345677.  1 ZZ
 3 00000001.  0. XXW  ALU=A-1  XXX   00000000.  2 ZZ
 4 00000010.  0. XXW  ALU=A-1  XXX   0000000F.  3 ZZ
```

```
 5 00000100.   0. XXW  ALU=A-1   XXX    000000FF.   4 ZZ
 6 00001000.   0. XXW  ALU=A-1   XXX    00000FFF.   5 ZZ
 7 00010000.   0. XXW  ALU=A-1   XXX    0000FFFF.   6 ZZ
 8 00100000.   0. XXW  ALU=A-1   XXX    000FFFFF.   7 ZZ
 9 01000000.   0. XXW  ALU=A-1   XXX    00FFFFFF.   8 ZZ
10 10000000.   0. XXW  ALU=A-1   XXX    0FFFFFFF.   9 ZZ
11 00000000.   0. XXW  ALU=A-1   XXX    FFFFFFFF.  10 ZZ
12
13 DECIMAL
14
15

SCREEN #21
 0 ." ALU  A-B  A-B-1" CR
 1 HEX
 2 FFFFFFFF. FFFFFFFF.  XXW  ALU=A-B    XXX   00000000.   1 ZZ
 3 00000000. FFFFFFFF.  XXW  ALU=A-B    XXX   00000001.   2 ZZ
 4 FFFFFFFF. 00000000.  XXW  ALU=A-B    XXX   FFFFFFFF.   3 ZZ
 5 00000000. 00000000.  XXW  ALU=A-B    XXX   00000000.   4 ZZ
 6 6432ACE4. 43221252.  XXW  ALU=A-B    XXX   21109A92.   5 ZZ
 7
 8 00000000. 00000000.  XXW  ALU=A-B-1  XXX   FFFFFFFF.   6 ZZ
 9 FFFFFFFF. 00000000.  XXW  ALU=A-B-1  XXX   FFFFFFFE.   7 ZZ
10 00000000. FFFFFFFF.  XXW  ALU=A-B-1  XXX   00000000.   8 ZZ
11 FFFFFFFF. FFFFFFFF.  XXW  ALU=A-B-1  XXX   FFFFFFFF.   9 ZZ
12 43239835. 436587AE.  XXW  ALU=A-B-1  XXX   FFBE1086.   A ZZ
13
14 DECIMAL
15

SCREEN #22
 0 ." ALU  A+B  A+B+1" CR
 1 HEX
 2 FFFFFFFF. FFFFFFFF.  XXW  ALU=A+B    XXX   FFFFFFFE.   1 ZZ
 3 00000000. FFFFFFFF.  XXW  ALU=A+B    XXX   FFFFFFFF.   2 ZZ
 4 FFFFFFFF. 00000000.  XXW  ALU=A+B    XXX   FFFFFFFF.   3 ZZ
 5 00000000. 00000000.  XXW  ALU=A+B    XXX   00000000.   4 ZZ
 6 6432ACE4. 43221252.  XXW  ALU=A+B    XXX   A754BF36.   5 ZZ
 7
 8 00000000. 00000000.  XXW  ALU=A+B+1  XXX   00000001.   6 ZZ
 9 FFFFFFFF. 00000000.  XXW  ALU=A+B+1  XXX   00000000.   7 ZZ
10 00000000. FFFFFFFF.  XXW  ALU=A+B+1  XXX   00000000.   8 ZZ
11 FFFFFFFF. FFFFFFFF.  XXW  ALU=A+B+1  XXX   FFFFFFFF.   9 ZZ
12 43239835. 436587AE.  XXW  ALU=A+B+1  XXX   86891FE4.   A ZZ
13
14 DECIMAL
15

SCREEN #23
 0 ." ALU  A+A  A+A+1" CR
 1 HEX
 2 FFFFFFFF. FFFFFFFF.  XXW  ALU=A+A    XXX   FFFFFFFE.   1 ZZ
 3 00000000. FFFFFFFF.  XXW  ALU=A+A    XXX   00000000.   2 ZZ
 4 FFFFFFFF. 00000000.  XXW  ALU=A+A    XXX   FFFFFFFE.   3 ZZ
 5 00000000. 00000000.  XXW  ALU=A+A    XXX   00000000.   4 ZZ
 6 6432ACE4. 43221252.  XXW  ALU=A+A    XXX   C86559C8.   5 ZZ
 7
 8 00000000. 00000000.  XXW  ALU=A+A+1  XXX   00000001.   6 ZZ
 9 FFFFFFFF. 00000000.  XXW  ALU=A+A+1  XXX   FFFFFFFF.   7 ZZ
10 00000000. FFFFFFFF.  XXW  ALU=A+A+1  XXX   00000001.   8 ZZ
```

```
11 FFFFFFFF. FFFFFFFF.  XXW  ALU=A+A+1  XXX   FFFFFFFF.  9 ZZ
12 43239835. 436587AE.  XXW  ALU=A+A+1  XXX   8647306B.  A ZZ
13
14 DECIMAL
15

SCREEN #24
 0 ." TEST 32-BIT BYTE ROTATE FROM ALU" CR
 1 HEX
 2 : YYA  0 :: ALU=B ROLL[ALU] DEST=DHI ;SET X!
 3        1 :: SOURCE=DHI ;SET X@  ;
 4
 5 12345678. YYA          78123456.        1 ZZ
 6 EDCBA987. YYA          87EDCBA9.        2 ZZ
 7
 8 DECIMAL
 9
10
11
12
13
14
15

SCREEN #25
 0 ." TEST 64-BIT SHIFT LEFT OF DLO/DHI -- 1" CR
 1 HEX
 2 : XXN  0 ::  ALU=B  DEST=DHI  ;SET   X!
 3        1 ::  DEST=DLO  ;SET   X!  ;
 4 : XXP  2 :: SOURCE=DLO  ;SET  X@
 5        3 :: SOURCE=DHI  ;SET  X@  ;
 6 : XXQ 0 DO   4 ::  SL[DLO] ALU=A CIN=0  SL[ALU] ;DO   LOOP  ;
 7 00000000. 00000000.  XXN  XXP  00000000. 1 ZZ  00000000. 2 ZZ
 8 0 :: SL[DLO]  ALU=A CIN=1  SL[ALU]  ;DO
 9          XXP   00000000. 3 ZZ   00000001. 4 ZZ
10  7 XXQ   XXP   00000000. 5 ZZ   00000080. 6 ZZ
11  1 XXQ   XXP   00000000. 7 ZZ   00000100. 8 ZZ
12  7 XXQ   XXP   00000000. 9 ZZ   00008000. A ZZ
13  1 XXQ   XXP   00000000. B ZZ   00010000. C ZZ
14  7 XXQ   XXP   00000000. D ZZ   00800000. E ZZ
15 DECIMAL

SCREEN #26
 0 ." TEST 64-BIT SHIFT LEFT OF DLO/DHI -- 2" CR
 1 HEX
 2  00800000.  00000000. XXN
 3  1 XXQ   XXP   00000000.  1 ZZ   01000000.  2 ZZ
 4  7 XXQ   XXP   00000000.  3 ZZ   80000000.  4 ZZ
 5  1 XXQ   XXP   00000001.  5 ZZ   00000000.  6 ZZ
 6  7 XXQ   XXP   00000080.  7 ZZ   00000000.  8 ZZ
 7  1 XXQ   XXP   00000100.  9 ZZ   00000000.  A ZZ
 8  7 XXQ   XXP   00008000.  B ZZ   00000000.  C ZZ
 9  1 XXQ   XXP   00010000.  D ZZ   00000000.  E ZZ
10  7 XXQ   XXP   00800000.  F ZZ   00000000. 10 ZZ
11  1 XXQ   XXP   01000000. 11 ZZ   00000000. 12 ZZ
12  7 XXQ   XXP   80000000. 13 ZZ   00000000. 14 ZZ
13  1 XXQ   XXP   00000000. 15 ZZ   00000000. 16 ZZ
14 DECIMAL
15
```

```
SCREEN #27
 0 ." TEST 64-BIT SHIFT RIGHT OF DLO/DHI -- 1" CR
 1 HEX
 2 : XXR  0 ::  ALU=B  DEST=DHI  ;SET   X!
 3         1 ::  DEST=DLO  ;SET   X!  ;
 4 : XXS  2 :: SOURCE=DLO  ;SET  X@
 5         3 :: SOURCE=DHI  ;SET  X@  ;
 6 : XXT 0 DO   4 ::  SR[DLO] ALU=A CIN=0  SR[ALU] ;DO   LOOP ;
 7 00000000. 00000000.  XXR  XXS  00000000. 1 ZZ  00000000. 2 ZZ
 8 0 :: SR[DLO]  ALU=A CIN=1  SR[ALU]   ;DO
 9          XXS   80000000. 3 ZZ   00000000. 4 ZZ
10  7 XXT   XXS   01000000. 5 ZZ   00000000. 6 ZZ
11  1 XXT   XXS   00800000. 7 ZZ   00000000. 8 ZZ
12  7 XXT   XXS   00010000. 9 ZZ   00000000. A ZZ
13  1 XXT   XXS   00008000. B ZZ   00000000. C ZZ
14 DECIMAL
15  .

SCREEN #28
 0 ." TEST 64-BIT SHIFT RIGHT OF DLO/DHI -- 2" CR
 1 HEX
 2  00000000.  00008000. XXR
 3  7 XXT   XXS   00000100.  1 ZZ   00000000.  2 ZZ
 4  1 XXT   XXS   00000080.  3 ZZ   00000000.  4 ZZ
 5  7 XXT   XXS   00000001.  5 ZZ   00000000.  6 ZZ
 6  1 XXT   XXS   00000000.  7 ZZ   80000000.  8 ZZ
 7  7 XXT   XXS   00000000.  9 ZZ   01000000.  A ZZ
 8  1 XXT   XXS   00000000.  B ZZ   00800000.  C ZZ
 9  7 XXT   XXS   00000000.  D ZZ   00010000.  E ZZ
10  1 XXT   XXS   00000000.  F ZZ   00008000. 10 ZZ
11  7 XXT   XXS   00000000. 11 ZZ   00000100. 12 ZZ
12  1 XXT   XXS   00000000. 13 ZZ   00000080. 14 ZZ
13  7 XXT   XXS   00000000. 15 ZZ   00000001. 16 ZZ
14  1 XXT   XXS   00000000. 17 ZZ   00000000. 18 ZZ
15 DECIMAL

SCREEN #29
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15

SCREEN #30
 0
 1
 2
 3
 4
 5
```

```
 6
 7
 8
 9
10
11
12
13
14
15

SCREEN #31
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15

SCREEN #32
 0 ." TEST RAM ADDRESS LATCH VIA ADDR COUNTER" CR
 1 HEX              MATH
 2 : YYG   0 :: DEST=ADDRESS-COUNTER ;SET X!
 3         1 :: SOURCE=ADDRESS-COUNTER ;SET X@ ;
 4
 5 12345678. YYG  07FFFFC. DAND   00345678.          1 ZZ
 6 EDCBA987. YYG  07FFFFC. DAND   004BA984.          1 ZZ
 7
 8 DECIMAL
 9
10
11
12
13
14
15

SCREEN #33
 0 ." TEST ADDR COUNTER VIA RAM ADDRESS LATCH" CR
 1 HEX
 2 : YYH   0 :: DEST=ADDRESS-COUNTER ;SET X!
 3         2 :: INC[ADC] ;DO
 4         3 :: SOURCE=ADDRESS-COUNTER ;SET X@ ;
 5
 6 12545978. YYH  07FFFFC. DAND   0054597C.          1 ZZ
 7 00000000. YYH  07FFFFC. DAND   00000004.          2 ZZ
 8 0000003C. YYH  07FFFFC. DAND   00000040.          3 ZZ
 9 000003FC. YYH  07FFFFC. DAND   00000400.          4 ZZ
10 00003FFC. YYH  07FFFFC. DAND   00004000.          5 ZZ
11 0003FFFC. YYH  07FFFFC. DAND   00040000.          6 ZZ
12 003FFFFC. YYH  07FFFFC. DAND   00400000.          7 ZZ
```

```
13 007FFFFC. YYH  07FFFFC. DAND   00000000.        8 ZZ
14 DECIMAL
15

SCREEN #34
 0 ." TEST ADDR COUNTER TO RAM ADDRESS LATCH" CR
 1 HEX
 2 : YYHA  ( RAM.BIT.PATTERN  ADDRESS -> BIT.PATTERN   )
 3         >> DEST=PAGE ;SET 0. X!
 4         DSWAP DOVER   RAM!
 5         >> DEST=ADDRESS-COUNTER ;SET X!
 6         >> SOURCE=ADDRESS-COUNTER  DEST=ADDRESS-LATCH ;DO
 7         >> SOURCE=RAM ;SET  X@ ;
 8
 9 00000000. 432C. YYHA  00000000.        1 ZZ
10 FFFFFFFF. 432C. YYHA  FFFFFFFF.        2 ZZ
11 AA55AA55. 432C. YYHA  AA55AA55.        3 ZZ
12 55AA55AA. 432C. YYHA  55AA55AA.        4 ZZ
13 DECIMAL
14
15

SCREEN #35
 0 ." TEST PAGE REGISTER" CR
 1 HEX
 2 : YYJ  0 :: DEST=PAGE ;SET X!
 3         1 :: SOURCE=ADDRESS-COUNTER ;SET X@ ;
 4
 5 000000000. YYJ  07F800000. DAND   000000000.        1 ZZ
 6 0FFFFFFFF. YYJ  07F800000. DAND   07F800000.        2 ZZ
 7 012345678. YYJ  07F800000. DAND   012000000.        3 ZZ
 8 0EDCBA987. YYJ  07F800000. DAND   06D800000.        4 ZZ
 9
10 DECIMAL
11
12
13
14
15

SCREEN #36
 0 ." ALU INPUT HOLDING LATCH" CR
 1 HEX
 2 : YYK    ( N1 -> notN1  N1 )
 3        1 ::  ALU=notB DEST=DHI  ;SET  X!
 4        2 ::  SOURCE=DHI ALU=B DEST=DHI  ;SET X@
 5        CYCLE  X@ ;
 6
 7 55555555. YYK     55555555.  1 ZZ
 8                   AAAAAAAA.  2 ZZ
 9 AAAAAAAA. YYK     AAAAAAAA.  3 ZZ
10                   55555555.  4 ZZ
11 12345678. YYK     12345678.  5 ZZ
12                   EDCBA987.  6 ZZ
13 DECIMAL
14
15

SCREEN #37
 0 ." TEST FLAG REGISTER" CR
 1 HEX
```

```
 2 : YYL     ( Clear all possible errors before testing flags )
 3         0 :: DEST=DP ;SET  -1. X!     1 :: DEST=RP ;SET -1. X!
 4         2 :: SOURCE=FLAGS ;DO
 5         3 :: DEST=FLAGS ;SET X!
 6         4 :: SOURCE=FLAGS ;SET  X@ ;
 7
 8 00000000. YYL  0FF000000. DAND   000000000.          1 ZZ
 9 FFFFFFFF. YYL  0FF000000. DAND   0FF000000.          2 ZZ
10 5A345678. YYL  0FF000000. DAND   05A000000.          3 ZZ
11 A5CBA980. YYL  0FF000000. DAND   0A5000000.          4 ZZ
12
13 DECIMAL
14
15

SCREEN #38
 0 ." TEST DP & RP OVERFLOW ERROR INTERRUPTS" CR
 1 HEX
 2 : YYM  0 :: DEST=DP ;SET X!
 3        1 :: DEST=FLAGS ;SET 0. X!
 4        2 :: SOURCE=FLAGS ;SET X@ ;
 5 : YYN  0 :: DEST=RP ;SET X!
 6        1 :: DEST=FLAGS ;SET 0. X!
 7        2 :: SOURCE=FLAGS ;SET X@ ;
 8
 9 0FFF. YYM  01000000. DAND   00000000.          1 ZZ
10 0000. YYM  01000000. DAND   01000000.          2 ZZ
11
12 0FFF. YYN  02000000. DAND   00000000.          3 ZZ
13 0000. YYN  02000000. DAND   02000000.          4 ZZ
14
15 DECIMAL

SCREEN #39
 0 ." TEST HOST INTERRUPTS" CR
 1 HEX
 2 \  NOTE: Parity interrupt currently untestable
 3 : YYP  1 :: DEST=FLAGS ;SET 0. X!
 4        2 :: SOURCE=FLAGS ;SET X@ ;
 5
 6 0 :: SOURCE=FLAGS ;DO
 7 YYP  04000000. DAND   00000000.          1 ZZ
 8 -1 PCREQ
 9 YYP  04000000. DAND   04000000.          2 ZZ
10 0 :: SOURCE=FLAGS ;DO
11
12
13 DECIMAL
14
15

SCREEN #40
 0 ." TEST NEXT ADDRESS REGISTER VIA DEST=" CR
 1 HEX   MATH
 2 : YYQ  0 :: DEST=FLAGS ;SET -1. X!  ( Mask interrupts )
 3        ( Ensure JMP instruction )    FFFFFFFC. DAND
 4        2 :: ALU=B DEST=DHI ;SET  X!
 5            ( Automatically clk into counter during decode )
 6        3 :: SOURCE=DHI DEST=DECODE ;DO
 7        4 :: DECODE ;DO
```

```
 8         5 ::          ;DO  CYCLE
 9         6 :: SOURCE=ADDRESS-COUNTER ;SET X@ ;
10
11 00000000. YYQ  007FFFF8. DAND   00000000.          1 ZZ
12 7FFFFFFF. YYQ  007FFFF8. DAND   007FFFF8.          2 ZZ
13 12345678. YYQ  007FFFF8. DAND   00345678.          3 ZZ
14 EDCBA987. YYQ  007FFFF8. DAND   004BA980.          4 ZZ
15 DECIMAL

SCREEN #41
 0 ." TEST NEXT ADDRESS REGISTER VIA DECODE" CR
 1 HEX
 2 : YYR   0 :: DEST=FLAGS ;SET -1. X!   ( Mask interrupts )
 3         1 :: DEST=DLO ;SET  X!
 4             ( Automatically clk into counter during decode )
 5         2 :: DECODE SOURCE=DLO DEST=RAM           ;DO
 6         3 ::        SOURCE=DLO DEST=RAM           ;DO  CYCLE
 7         4 :: DECODE SOURCE=DLO DEST=RAM           ;DO
 8         5 ::        SOURCE=DLO DEST=RAM           ;DO  CYCLE
 9         6 :: SOURCE=ADDRESS-COUNTER ;SET X@ ;
10
11 00000000. YYR  007FFFF8. DAND   00000000.          1 ZZ
12 7FFFFFF8. YYR  007FFFF8. DAND   007FFFF8.          2 ZZ
13 12345678. YYR  007FFFF8. DAND   00345678.          3 ZZ
14 EDCBA980. YYR  007FFFF8. DAND   004BA980.          4 ZZ
15 DECIMAL

SCREEN #42
 0 ." TEST INSTRUCTION LATCH & CALL/EXIT BITS VIA DEST=" CR
 1 HEX   MATH
 2 : YYS   >> DEST=FLAGS ;SET 80000000. X!   ( Mask interrupts )
 3         >> DEST=DECODE ;SET X!
 4         >> SOURCE=MPC ;SET X@ ;
 5
 6 00000000. YYS  FF800003. DAND   00000000.          1 ZZ
 7 FFFFFFFF. YYS  FF800003. DAND   FF800003.          2 ZZ
 8 12345679. YYS  FF800003. DAND   12000001.          3 ZZ
 9 EDCBA986. YYS  FF800003. DAND   ED800002.          4 ZZ
10 80000000. YYS  FF800003. DAND   80000000.          5 ZZ
11 40000000. YYS  FF800003. DAND   40000000.          6 ZZ
12 20000000. YYS  FF800003. DAND   20000000.          7 ZZ
13 DECIMAL
14
15

SCREEN #43
 0 ." TEST INSTRUCTION LATCH & CALL/EXIT BITS VIA DECODE" CR
 1 HEX
 2 : YYT   0 :: DEST=FLAGS ;SET -1. X!   ( Mask interrupts )
 3         1 :: DEST=DLO ;SET  X!
 4             ( Automatically clk into counter during decode )
 5         2 :: DECODE SOURCE=DLO DEST=RAM ;DO
 6         3 ::        SOURCE=DLO DEST=RAM ;DO  CYCLE
 7         4 :: SOURCE=MPC ;SET X@ ;
 8
 9 00000000. YYT  F8000003. DAND   00000000.          1 ZZ
10 FFFFFFFF. YYT  F8000003. DAND   F8000003.          2 ZZ
11 12345679. YYT  F8000003. DAND   10000001.          3 ZZ
12 EDCBA986. YYT  F8000003. DAND   E8000002.          4 ZZ
13 DECIMAL
```

```
14
15

SCREEN #44
 0 ." TEST MICROPROGRAM COUNTER -- LOAD/READ" CR
 1 HEX
 2 : YYU  0 :: DEST=FLAGS ;SET -1. X!  ( Mask interrupts )
 3        1 :: DEST=DLO ;SET  X!
 4        2 :: DECODE  SOURCE=DLO  DEST=RAM  JMP=000 ;DO
 5        3 ::         SOURCE=DLO  DEST=RAM  JMP=000 ;DO
 6        4 :: DECODE  SOURCE=DLO  DEST=RAM  JMP=000 ;DO
 7        5 :: SOURCE=MPC ;SET X@ ;
 8
 9 00000000. YYU  000FF800. DAND   00000000.        1 ZZ
10 FFFFFFFF. YYU  000FF800. DAND   000FF800.        2 ZZ
11 12345679. YYU  000FF800. DAND   00012000.        3 ZZ
12 EDCBA986. YYU  000FF800. DAND   000ED800.        4 ZZ
13 DECIMAL
14
15

SCREEN #45
 0 ." TEST MICROPROGRAM COUNTER -- INCREMENT" CR
 1 HEX
 2 : YYUA  0 :: DEST=FLAGS ;SET -1. X!  ( Mask interrupts )
 3         1 :: DEST=DLO ;SET  X!
 4         2 :: DECODE  SOURCE=DLO  DEST=RAM  JMP=000 ;DO
 5         3 ::         SOURCE=DLO  DEST=RAM  JMP=000 ;DO
 6         4 :: DECODE  SOURCE=DLO  DEST=RAM  JMP=000 ;DO
 7         6 :: INC[MPC]  ;DO    6 :: ;DO
 8         6 :: SOURCE=MPC ;SET X@ ;
 9
10 00000000. YYUA  000FF800. DAND   00000800.        1 ZZ
11 FFFFFFFF. YYUA  000FF800. DAND   000F8000.        2 ZZ
12 12345679. YYUA  000FF800. DAND   00012800.        3 ZZ
13 EDCBA986. YYUA  000FF800. DAND   000EE000.        4 ZZ
14 DECIMAL
15

SCREEN #46
 0 ." TEST INTERRUPT INSTRUCTION LATCH & CALL/EXIT BITS" CR
 1 HEX
 2 : YYV  0 :: DEST=FLAGS ;SET 40000000. X!  ( UN-Mask interrupts )
 3        1 :: DEST=DLO ;SET  X!
 4            ( Automatically clk into counter during decode )
 5        2 :: DECODE SOURCE=DLO DEST=RAM ;DO
 6        3 ::        SOURCE=DLO DEST=RAM ;DO  CYCLE
 7        4 :: SOURCE=MPC ;SET X@ ;
 8
 9 00000000. YYV  F8000003. DAND   00000000.        1 ZZ
10 FFFFFFFF. YYV  F8000003. DAND   00000000.        2 ZZ
11 12345679. YYV  F8000003. DAND   00000000.        3 ZZ
12 EDCBA986. YYV  F8000003. DAND   00000000.        4 ZZ
13 DECIMAL
14
15

SCREEN #47
 0 ." TEST CARRY CONDITION CODE" CR
 1 HEX
 2 : YYW  0 :: DEST=FLAGS ;SET -1. X!  ( Mask interrupts )
```

```
 3         1 :: ALU=B     DEST=DHI   ;SET        X!
 4         2 :: ALU=A+B   DEST=DHI   ;SET        X!
 5         6 :: JMP=00C    SOURCE=MPC ;SET X@ ;
 6
 7  10.   20.   YYW    00000700. DAND    00000100.    1 ZZ
 8 -10.  -20.   YYW    00000700. DAND    00000000.    2 ZZ
 9
10
11 DECIMAL
12
13
14
15

SCREEN #48
 0 ." TEST ALU=0 CONDITION CODE" CR
 1 HEX
 2 : YYX   0 :: DEST=FLAGS ;SET -1. X!   ( Mask interrupts )
 3         1 :: ALU=B     DEST=DHI   ;SET        X!
 4         6 :: JMP=01Z    SOURCE=MPC ;SET X@ ;
 5
 6  10.   YYX    00000700. DAND    00000300.    1 ZZ
 7   0.   YYX    00000700. DAND    00000200.    2 ZZ
 8
 9
10 DECIMAL
11
12
13
14
15

SCREEN #49
 0 ." TEST ALU SIGN CONDITION CODE" CR
 1 HEX
 2 : YYY   0 :: DEST=FLAGS ;SET -1. X!   ( Mask interrupts )
 3         1 :: ALU=B     DEST=DHI   ;SET        X!
 4         6 :: JMP=10S    SOURCE=MPC ;SET X@ ;
 5
 6 80000000.   YYY    00000700. DAND    00000500.    1 ZZ
 7 00000000.   YYY    00000700. DAND    00000400.    2 ZZ
 8
 9
10 DECIMAL
11
12
13
14
15

SCREEN #50
 0 ." TEST DLO SHIFT OUT LOW BIT CONDITION CODE" CR
 1 HEX
 2 : YYZ   0 :: DEST=FLAGS ;SET -1. X!   ( Mask interrupts )
 3         1 :: DEST=DLO ;SET         X!
 4         2 :: ;DO
 5         6 :: JMP=11L    SOURCE=MPC ;SET X@ ;
 6
 7   1.   YYZ    00000700. DAND    00000700.    1 ZZ
 8   0.   YYZ    00000700. DAND    00000600.    2 ZZ
```

```
 9
10
11 DECIMAL
12
13
14
15

SCREEN #51
 0 \  PRIMITIVE TO FORM 32-BIT PATTERN FROM A 0..4K #
 1 HEX
 2 \ Repeat the 12 bits to form full 32-bit pattern for RAM testing
 3 CODE 4KWD  ( N=0..4K -> DPATTERN )
 4    AX POP   BX , AX MOV    CX , AX MOV
 5    BX , 1 SHL   BX , 1 SHL   BX , 1 SHL  BX , 1 SHL
 6    AH , BL OR
 7    CX , 1 SHR   CX , 1 SHR   CX , 1 SHR  CX , 1 SHR
 8    CH , AL MOV
 9    AX PUSH  CX PUSH
10    NEXT JMP  END-CODE
11 DECIMAL
12
13
14
15

SCREEN #52
 0 ." TEST MRAM " CR
 1 HEX
 2 : WWA 1000 0 DO  I 4KWD  I MRAM!
 3                          I OFF AND 0= IF ." !" THEN  LOOP
 4    CR 1000 0 DO  I MRAM@    I 4KWD I  ZZ
 5                          I OFF AND 0= IF ." @" THEN  LOOP
 6    CR 1000 0 DO  I OFFFF XOR   4KWD I MRAM!
 7                          I OFF AND 0= IF ." !" THEN  LOOP
 8    CR 1000 0 DO  I MRAM@    I OFFFF XOR 4KWD I  ZZ
 9                          I OFF AND 0= IF ." @" THEN  LOOP ;
10
11 0 DISP-PASS !  WWA   1 DISP-PASS !
12 CR
13 12345678. 0 MRAM!   0 MRAM@   12345678. 999 ZZ
14
15 DECIMAL

SCREEN #53
 0 ." TEST DATA STACK --1" CR
 1 HEX
 2 : WWB  ( Ddata addr -> )      0 :: DEST=DP ;SET  0 X!
 3        0 :: DEST=DS  ;SET  X! ;
 4 : WWC  ( addr -> Ddata )      0 :: DEST=DP ;SET  0 X!
 5       0 :: SOURCE=DS ;SET X@ ;
 6 : WWD 1000 0 DO  I 4KWD  I WWB
 7                          I OFF AND 0= IF ." !" THEN  LOOP
 8    CR 1000 0 DO  I WWC      I 4KWD I  ZZ
 9                          I OFF AND 0= IF ." @" THEN  LOOP ;
10
11 0 DISP-PASS !  WWD   1 DISP-PASS !
12 CR
13 12345678.  0 WWB  0 WWC   12345678.  999 ZZ
14
15 DECIMAL
```

```
SCREEN #54
 0 ." TEST DATA STACK --2" CR
 1 HEX
 2 : WWE  ( Ddata addr -> )       0 :: DEST=DP ;SET  0 X!
 3        0 :: DEST=DS  ;SET  X! ;
 4 : WWF  ( addr -> Ddata )       0 :: DEST=DP ;SET  0 X!
 5       0 :: SOURCE=DS ;SET X@ ;
 6 : WWG 1000 0 DO  I OFFFF XOR  4KWD  I WWE
 7                               I OFF AND 0= IF ." !" THEN  LOOP
 8    CR 1000 0 DO  I WWF       I OFFFF XOR 4KWD I  ZZ
 9                               I OFF AND 0= IF ." @" THEN  LOOP ;
10
11 0 DISP-PASS !  WWG   1 DISP-PASS !
12 CR
13 456789AB.  0 WWE  0 WWF   456789AB.  999 ZZ
14
15 DECIMAL

SCREEN #55
 0 ." TEST RETURN STACK --1" CR
 1 HEX
 2 : WWH  ( Ddata addr -> )       0 :: DEST=RP ;SET  0 X!
 3        0 :: DEST=RS  ;SET  X! ;
 4 : WWJ  ( addr -> Ddata )       0 :: DEST=RP ;SET  0 X!
 5       0 :: SOURCE=RS ;SET X@ ;
 6 : WWK 1000 0 DO  I 4KWD  I WWH
 7                               I OFF AND 0= IF ." !" THEN  LOOP
 8    CR 1000 0 DO  I WWJ       I 4KWD I  ZZ
 9                               I OFF AND 0= IF ." @" THEN  LOOP ;
10
11 0 DISP-PASS !  WWK   1 DISP-PASS !
12 CR
13 12345678.  0 WWH  0 WWJ   12345678.  999 ZZ
14
15 DECIMAL

SCREEN #56
 0 ." TEST RETURN STACK --2" CR
 1 HEX
 2 : WWL  ( Ddata addr -> )       0 :: DEST=RP ;SET  0 X!
 3        0 :: DEST=RS  ;SET  X! ;
 4 : WWM  ( addr -> Ddata )       0 :: DEST=RP ;SET  0 X!
 5       0 :: SOURCE=RS ;SET X@ ;
 6 : WWN 1000 0 DO  I OFFFF XOR  4KWD  I WWL
 7                               I OFF AND 0= IF ." !" THEN  LOOP
 8    CR 1000 0 DO  I WWM       I OFFFF XOR 4KWD I  ZZ
 9                               I OFF AND 0= IF ." @" THEN  LOOP ;
10
11 0 DISP-PASS !  WWN   1 DISP-PASS !
12 CR
13 456789AB.  0 WWL  0 WWM   456789AB.  999 ZZ
14
15 DECIMAL

SCREEN #57
 0 ." TEST EXIT MECHANISM" CR
 1 HEX            MATH
 2 0 :: DEST=FLAGS ;SET -1. X!  ( Mask interrupts )
 3 0 :: DEST=RP ;SET  201. X!
 4 0 :: DEST=RS ;SET  12345678. X!
```

```
 5 0 :: DEST=RP ;SET  200. X!
 6 0 :: DEST=RS ;SET  67893453. X!
 7 0 :: DEST=DLO ;SET   0075A5AD. X!
 8 0 ::         SOURCE=DLO DEST=DECODE   ;DO
 9 0 :: DECODE SOURCE=DLO DEST=RAM  ;DO
10 0 ::         SOURCE=DLO DEST=RAM  ;DO   CYCLE
11 0 :: SOURCE=RP ;SET  X@       OFFF. DAND          201.  1 ZZ
12 0 :: SOURCE=RS ;SET  X@                        12345678.  2 ZZ
13
14 DECIMAL
15

SCREEN #58
 0 ." TEST CALL MECHANISM" CR
 1 HEX                  MATH
 2 0 :: DEST=FLAGS ;SET -1. X!  ( Mask interrupts )
 3 567C. 2. D+    1238. RAM!       7BCDEF80.  567C. RAM!
 4 >> DEST=RP ;SET  200. X!
 5 >> DEST=RS ;SET  12345678. X!
 6 >> DEST=PAGE ;SET   0. X!
 7 >> DEST=DECODE ;SET   0000123A.    X!
 8 >> DEST=ADDRESS-COUNTER ;SET   0000123A.    X!
 9 >> DECODE ;DO
10 >>         ;DO   CYCLE          CYCLE
11 >> SOURCE=RP ;SET  X@             OFFF.  DAND   1FF.  1 ZZ
12 >> SOURCE=RS ;SET  X@                      0000123C.  2 ZZ
13 >> DEST=RP ;SET  200. X!
14 >> SOURCE=RS ;SET  X@                       12345678.  3 ZZ
15 DECIMAL

SCREEN #59
 0 ." TEST RAM -- 1ST 32K BYTES " CR
 1 HEX
 2 : WWP 8000 0 DO  I 4KWD  I 0 RAM!
 3                          I 3FF AND 0= IF ." !" THEN  4 /LOOP
 4    CR 8000 0 DO  I 0 RAM@    I 4KWD I  ZZ
 5                          I 3FF AND 0= IF ." @" THEN  4 /LOOP
 6    CR 8000 0 DO  I OFFFF XOR  4KWD I 0 RAM!
 7                          I 3FF AND 0= IF ." !" THEN  4 /LOOP
 8    CR 8000 0 DO  I 0 RAM@    I OFFFF XOR 4KWD I  ZZ
 9                          I 3FF AND 0= IF ." @" THEN  4 /LOOP ;
10
11 0 DISP-PASS !  WWP   1 DISP-PASS !
12 CR
13 12345678. 0. RAM!   0. RAM@   12345678. 999 ZZ
14
15 DECIMAL

SCREEN #60
 0 ." TEST RAM BYTE ALIGNMENT FOR READ" CR
 1 HEX
 2 : WWQ  RAM!
 3       0 :: SOURCE=RAM-BYTE ;SET  X@ ;
 4
 5 12345678.  0.  WWQ   00000078.  1 ZZ
 6 12345678.  1.  WWQ   00000056.  2 ZZ
 7 12345678.  2.  WWQ   00000034.  3 ZZ
 8 12345678.  3.  WWQ   00000012.  4 ZZ
 9
10 EDCBA987.  4.  WWQ   00000087.  5 ZZ
```

```
11 EDCBA987.   5.   WWQ    000000A9.   6 ZZ
12 EDCBA987.   6.   WWQ    000000CB.   7 ZZ
13 EDCBA987.   7.   WWQ    000000ED.   8 ZZ
14
15 DECIMAL

SCREEN #61
 0 ." TEST RAM BYTE ALIGNMENT FOR WRITE" CR
 1 HEX           MATH
 2 : WWR     ECDBAFCB.  DOVER RAM!    ( Set testing background )
 3        0 :: DEST=RAM-BYTE ;SET  DSWAP  X!
 4        RAM@  ;
 5
 6 43212435.   0.   WWR    ECDBAF35.   1 ZZ
 7 43212435.   1.   WWR    ECDB35CB.   2 ZZ
 8 43212435.   2.   WWR    EC35AFCB.   3 ZZ
 9 43212435.   3.   WWR    35DBAFCB.   4 ZZ
10
11 432124CA.   4.   WWR    ECDBAFCA.   5 ZZ
12 432124CA.   5.   WWR    ECDBCACB.   6 ZZ
13 432124CA.   6.   WWR    ECCAAFCB.   7 ZZ
14 432124CA.   7.   WWR    CADBAFCB.   8 ZZ
15 DECIMAL

SCREEN #62
 0 \ TESTING 32-BIT FETCH AND STORE IN PROPER BOARD BYTE ORDER
 1 HEX
 2 CODE XD!  ( D ADDR -> )     \ Stores in lo..hi byte order
 3    BX POP   DX POP   AX POP
 4      [BX] , AL MOV      1 [BX] , AH MOV
 5    2 [BX] , DL MOV      3 [BX] , DH MOV
 6    NEXT JMP  END-CODE
 7
 8 CODE XD@  ( ADDR -> D )     \ Fetches in lo..hi byte order
 9    BX POP
10    AL , 0 [BX] MOV      AH , 1 [BX] MOV
11    DL , 2 [BX] MOV      DH , 3 [BX] MOV
12    AX PUSH  DX PUSH
13    NEXT JMP  END-CODE
14 DECIMAL
15

SCREEN #63
 0 ." TEST RAM DMA WRITE " CR
 1 HEX
 2 : WWS RESET-BOARD    0 DISP-PASS !
 3       500 0 DO  12345678. I 0 1230. D+  RAM!     4 /LOOP
 4    CR 500 4 DO  I 4KWD  I PAD  + XD!     4 /LOOP
 5    PAD 4 +  1234.  100 W->BOARD
 6    CR 404 4 DO  I 0 1230. D+  RAM@     I 4KWD   I  ZZ
 7                              4 /LOOP
 8    1 DISP-PASS !
 9     -4. 1234. D+ RAM@  12345678.       0  ZZ
10    404. 1234. D+ RAM@  12345678.     404  ZZ
11    408. 1234. D+ RAM@  12345678.     408  ZZ ;
12 WWS
13 : WRITE-LOOP  BEGIN WWS ?TERMINAL UNTIL ;
14 DECIMAL
15
```

```
SCREEN #64
 0 ." TEST RAM DMA READ " CR
 1 HEX
 2 : WWT RESET-BOARD    0 DISP-PASS !
 3       500 0 DO  12345678.  PAD I +    XD!    4 /LOOP
 4    CR 400 0 DO  I 4KWD  I 0 2350. D+  RAM!   4 /LOOP
 5    2350.  PAD  4 +   100 BOARD->W
 6    CR 400 0 DO  PAD I + 4 +  XD@    I 4KWD   I  ZZ  4 /LOOP
 7    1 DISP-PASS !
 8    PAD  0  +  XD@  12345678.   -1  ZZ
 9    PAD 404 +  XD@  12345678.  404  ZZ
10    PAD 408 +  XD@  12345678.  408  ZZ  ;
11 WWT
12 : READ-LOOP  BEGIN WWT ?TERMINAL UNTIL ;
13 : ##R  BEGIN  RESET-BOARD  2350. PAD 4 + 100 BOARD->W
14    ?TERMINAL UNTIL ;   DECIMAL
15

SCREEN #65
 0 : ##W  BEGIN  RESET-BOARD  PAD 4 + 2350. 100 W->BOARD
 1    ?TERMINAL UNTIL ;   DECIMAL
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15

SCREEN #66
 0 ." TEST RAM DMA WRITE/READ " CR
 1 HEX
 2 : WWW RESET-BOARD    0 DISP-PASS !
 3    500 0 DO  I 4KWD  I PAD  + XD!    4 /LOOP
 4    PAD 4 +  3320.  400 W->BOARD
 5    3320.  PAD  18 +   400 BOARD->W
 6    400 4 DO  PAD I + 14 +  XD@    I 4KWD   I  ZZ  4 /LOOP
 7    1 DISP-PASS !          ;
 8 WWW   CR ." PASS"
 9 DECIMAL
10 : R/W-LOOP  BEGIN WWW ." ." ?TERMINAL UNTIL ;
11
12
13
14
15

SCREEN #67
 0
 1
 2
 3
 4
 5
```

```
 6
 7
 8
 9
10
11
12
13
14
15

SCREEN #68
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15

SCREEN #69
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15

SCREEN #70
 0 \ Word to zero RAM in anticipation of a ram test
 1 : HI-ADDR  FIRST 100 - ;
 2
 3 : FLUSH-RAM
 4    WORD-ALIGN
 5    HI-ADDR  PAD 50 +
 6    DO  0   I !
 7       4 +LOOP ;
 8
 9
10
11
12
```

```
13
14
15

SCREEN #71
 0 \ ADDRESS WRITE/READ RAM TEST
 1 DECIMAL  VARIABLE FX
 2 : ADDR-RW-TEST  ( -> )  SAVE-BUFFERS    0 FX !
 3    CR ." ADDRESS STORAGE RAM TEST"
 4    HI-ADDR 3 OR  3 XOR   PAD    3 OR  3 XOR
 5    DO  I NEGATE  I !
 6        I @ NEGATE  DUP  I  = NOT
 7       IF  CR ." FAILURE A  ADDR=" I . ."  VALUE=" U.  1 FX !
 8         KEY 13 =   ABORT" ABORTED"  ELSE DROP THEN
 9    4 /LOOP
10    HI-ADDR 3 OR  3 XOR   PAD    3 OR  3 XOR    .
11    DO  I @ NEGATE  DUP I  = NOT
12       IF  CR ." FAILURE B  ADDR=" I . ." VALUE=" U.   1 FX !
13         KEY 13 =   ABORT" ABORTED"  ELSE DROP THEN
14    4 /LOOP EMPTY-BUFFERS  ;
15 : ZZ BEGIN FLUSH-RAM ADDR-RW-TEST ?TERMINAL FX @ OR UNTIL ;

SCREEN #72
 0 \ ADDRESS WRITE/READ RAM TEST
 1 HEX
 2 : VAL-RW-TEST  ( VALUE -> )            SAVE-BUFFERS
 3    CR ." VALUE STORAGE RAM TEST   VALUE=" DUP U.
 4    HI-ADDR 3 OR  3 XOR   PAD    3 OR  3 XOR    .
 5    DO  DUP I !  ( I 70000 = IF ." 70000" THEN  )
 6       4 +LOOP
 7    HI-ADDR 3 OR  3 XOR   PAD    3 OR  3 XOR
 8    DO  I NOP @ NOP DDUP = NOT
 9       IF  CR ." FAILURE B  ADDR=" I U.  ."  VALUE=" DUP U.
10         KEY OD = ABORT" ABORTED"   ELSE DROP THEN
11    4 +LOOP     DROP  EMPTY-BUFFERS  ;
12
13 DECIMAL
14
15

SCREEN #73
 0 \ MASTER WRITE/READ RAM TEST
 1 HEX
 2 : CHECK    ?TERMINAL ABORT" BREAK" ;
 3 : RAM-TESTA  ( -> )     HEX
 4    ADDR-RW-TEST      CHECK
 5     1      VAL-RW-TEST  CHECK        2      VAL-RW-TEST  CHECK
 6     4      VAL-RW-TEST  CHECK        8      VAL-RW-TEST  CHECK
 7     10     VAL-RW-TEST  CHECK        20     VAL-RW-TEST  CHECK
 8     40     VAL-RW-TEST  CHECK        80     VAL-RW-TEST  CHECK
 9     100    VAL-RW-TEST  CHECK        200    VAL-RW-TEST  CHECK
10     400    VAL-RW-TEST  CHECK        800    VAL-RW-TEST  CHECK
11     1000   VAL-RW-TEST  CHECK        2000   VAL-RW-TEST  CHECK
12     4000   VAL-RW-TEST  CHECK        8000   VAL-RW-TEST  CHECK
13     0000   VAL-RW-TEST  CHECK    FFFFFFFF   VAL-RW-TEST  CHECK
14 55555555   VAL-RW-TEST  CHECK    AAAAAAAA   VAL-RW-TEST  CHECK
15    DECIMAL ;        DECIMAL

SCREEN #74
 0 \ MASTER WRITE/READ RAM TEST
 1 HEX
```

```
 2 : RAM-TEST    ( -> )  RAM-TESTA    HEX
 3     ADDR-RW-TEST       CHECK
 4 10000      VAL-RW-TEST    CHECK      20000      VAL-RW-TEST   CHECK
 5 40000      VAL-RW-TEST    CHECK      80000      VAL-RW-TEST   CHECK
 6 100000     VAL-RW-TEST    CHECK      200000     VAL-RW-TEST   CHECK
 7 400000     VAL-RW-TEST    CHECK      800000     VAL-RW-TEST   CHECK
 8 1000000    VAL-RW-TEST    CHECK      2000000    VAL-RW-TEST   CHECK
 9 4000000    VAL-RW-TEST    CHECK      8000000    VAL-RW-TEST   CHECK
10 10000000   VAL-RW-TEST    CHECK      20000000   VAL-RW-TEST   CHECK
11 40000000   VAL-RW-TEST    CHECK      80000000   VAL-RW-TEST   CHECK
12    DECIMAL ;
13 DECIMAL
14
15

SCREEN #75
 0 \ REPEAT RAM TEST
 1 DECIMAL
 2 : REPEAT RAM-TEST
 3    REPS 0 DO CR CR ." RAM TEST #" I .
 4                                    RAM-TEST LOOP ;
 5
 6
 7
 8
 9
10
11
12
13
14
15

SCREEN #76
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15

SCREEN #77
 0
 1
 2
 3
 4
 5
 6
 7
```

```
 8
 9
10
11
12
13
14
15

SCREEN #78
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15

SCREEN #79
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15

SCREEN #80
 0 \ MICROCODE TESTING      DO  LOOP
 1 DECIMAL
 2 : XLOOP
 3    1 2 3 4 5
 4     REPS 0 DO  0   200000  -100000  DO
 5          1+     LOOP
 6          DUP  300000 = NOT IF . 1 ABORT" LOOPA"
 7          ELSE DROP THEN
 8            ." ."  ?TERMINAL IF  ABORT." BREAK"  THEN LOOP
 9    CR . . . . .  CR ;
10
11 XLOOP
12
13
14
15
```

```
SCREEN #81
 0 \ MICROCODE TESTING       DOVAR  DOCON
 1 DECIMAL
 2 12345678 CONSTANT CONA
 3 VARIABLE VARA
 4
 5 : XVARCON
 6   1 2 3 4 5
 7     REPS  0 DO      #TIMES 0 DO
 8       CONA I + VARA !    VARA @ I -    CONA = NOT
 9          IF 1 ABORT" DOVAR/DOCON"    THEN
10       0 VARA !
11    LOOP  ." ."  ?TERMINAL IF  1 ABORT" BREAK"  THEN  LOOP
12    CR  . . . . .  CR ;
13
14 XVARCON
15

SCREEN #82
 0 \ MICROCODE TESTING        ABORT"
 1 DECIMAL
 2 : XABORT
 3    1 2 3 4 5
 4     REPS  0 DO                #TIMES 0  DO
 5        1 NOT  ABORT" ABORTA"
 6        0      ABORT" ABORTB"
 7    LOOP  ." ."  ?TERMINAL ABORT" BREAK"     LOOP
 8    CR . . . . .  CR ;
 9
10 XABORT
11
12
13
14
15

SCREEN #83
 0 \ MICROCODE TESTING        =
 1 DECIMAL
 2 : X=
 3    1 2 3 4 5
 4     REPS  0 DO                   #TIMES 0 DO
 5        I     I          =  NOT  ABORT" =A"
 6        I     I 1+       =       ABORT" =C"
 7    LOOP  ." ."  ?TERMINAL ABORT" BREAK"     LOOP
 8    CR . . . . .  CR ;
 9
10 X=
11
12
13
14
15

SCREEN #84
 0 \ MICROCODE TESTING       +  -
 1 DECIMAL
 2 : X+-
 3    1 2 3 4 5
 4     REPS  0 DO   #TIMES 0 DO
 5       12343456 56787654 +  69131110 = NOT  ABORT" +A"
```

```
 6             -1          1 +       0 = NOT  ABORT" +B"
 7       12346545 56782345 - -44435800 = NOT  ABORT" -A"
 8              0          1 -      -1 = NOT  ABORT" -B"
 9    LOOP  ." ."  ?TERMINAL ABORT" BREAK"     LOOP
10    CR . . . . .  CR ;
11
12 X+-
13
14
15

SCREEN #85
 0 \ MICROCODE TESTING       +!
 1 DECIMAL
 2 VARIABLE VARB
 3 : X+!
 4    1 2 3 4 5
 5     REPS 0 DO  12345678 DUP VARB !   #TIMES 0 DO
 6       I +        I  VARB  +!
 7       DUP   VARB @ = NOT  ABORT" +!"
 8    LOOP  DROP ." ."  ?TERMINAL ABORT" BREAK"     LOOP
 9    CR . . . . .  CR ;
10
11 X+!
12
13
14
15

SCREEN #86
 0 \ MICROCODE TESTING       -ROT
 1 DECIMAL
 2 : X-ROT
 3    1 2 3 4 5
 4     REPS 0 DO                #TIMES  0 DO
 5       I  I'  J      -ROT
 6       I'         = NOT  ABORT" -ROTA"
 7       I          = NOT  ABORT" -ROTB"
 8       J          = NOT  ABORT" -ROTC"
 9    LOOP  ." ."  ?TERMINAL ABORT" BREAK"     LOOP
10    CR . . . . .  CR ;
11
12 X-ROT
13
14
15

SCREEN #87
 0 \ MICROCODE TESTING       ROT
 1 DECIMAL
 2 : XROT
 3    1 2 3 4 5
 4     REPS 0 DO                #TIMES  0 DO
 5       J   I     12345678                ROT
 6       J          = NOT  ABORT" ROTA"
 7       12345678   = NOT  ABORT" ROTB"
 8       I          = NOT  ABORT" ROTC"
 9    LOOP  ." ."  ?TERMINAL ABORT" BREAK"   LOOP
10    CR . . . . .  CR ;
11
12 XROT
```

```
13
14
15

SCREEN #88
 0 \ MICROCODE TESTING        0     OBRANCH  BRANCH
 1 DECIMAL
 2 : XO
 3     1 2 3 4 5
 4      REPS 0 DO                #TIMES  0 DO
 5        0  IF   1 ABORT" OBRANCH"   THEN
 6        1  IF    ELSE    1 ABORT" BRANCH"   THEN
 7     LOOP   ." ."  ?TERMINAL ABORT" BREAK"    LOOP
 8     CR . . . . .  CR ;
 9
10 XO
11
12
13
14
15

SCREEN #89
 0 \ MICROCODE TESTING        1+  1-  2*
 1 HEX
 2 : X1
 3     1 2 3 4 5
 4      REPS 0 DO         #TIMES  0 DO
 5          7FFFFFFF  1+  80000000 =    NOT  ABORT" 1+"
 6          FFFFFFFF  1+                     ABORT" 1+"
 7          00000000  1-  FFFFFFFF =    NOT  ABORT" 1-"
 8          3FFFFFFF  2*  7FFFFFFE =    NOT  ABORT" 2*"
 9     LOOP   ." ."  ?TERMINAL ABORT" BREAK"     LOOP
10     CR . . . . .  CR ;
11 DECIMAL
12 X1
13
14
15

SCREEN #90
 0 \ MICROCODE TESTING        <
 1 DECIMAL
 2 : X<
 3     1 2 3 4 5
 4      REPS 0 DO                    #TIMES 0 DO
 5        I I 1+ <  NOT  ABORT" <A"
 6        I I    <       ABORT" <B"
 7     LOOP   ." ."  ?TERMINAL ABORT" BREAK"     LOOP
 8     CR . . . . .  CR ;
 9
10 X<
11
12
13
14
15

SCREEN #91
 0 \ MICROCODE TESTING       DO +LOOP
 1 DECIMAL
```

```
 2 : X+LOOP
 3    1 2 3 4 5
 4     REPS 0 DO  0   200000 -100000   DO
 5         1+  3 +LOOP
 6         100000 = NOT ABORT" +LOOPA"
 7           ." ."  ?TERMINAL ABORT" BREAK"    LOOP
 8    CR . . . . .  CR ;
 9
10 X+LOOP
11
12
13
14
15

SCREEN #92
 0 \ MICROCODE TESTING   I  I'  J
 1 HEX
 2 : XIIJ
 3    1 2 3 4 5
 4     REPS 0 DO  #TIMES 0 DO
 5       11111111  22222222   33333333  >R >R >R      J  I'  I
 6       R>  = NOT ABORT" I"
 7       R>  = NOT ABORT" I'"
 8       R>  = NOT ABORT" J"
 9    LOOP  ." ."  ?TERMINAL ABORT" BREAK"    LOOP
10    CR . . . . .  CR ;
11
12 DECIMAL
13 XIIJ
14
15

SCREEN #93
 0 \ MICROCODE TESTING      0<
 1 DECIMAL
 2 : X0<
 3    1 2 3 4 5
 4     REPS 0 DO  #TIMES 1+  1 DO
 5           I        0<      ABORT" 0<A"
 6           I NEGATE 0< NOT  ABORT" 0<B"
 7    LOOP  ." ."  ?TERMINAL ABORT" BREAK"    LOOP
 8    CR . . . . .  CR ;
 9
10 X0<
11
12
13
14
15

SCREEN #94
 0 \ MICROCODE TESTING      AND  OR  XOR
 1 HEX
 2 100 ALLOT
 3 : XAOX
 4    1 2 3 4 5
 5     REPS 0 DO  #TIMES 0 DO
 6       05AF05AF 5AF05AF0  AND    00A000A0 = NOT  ABORT" AND"
 7       05AF05AF 5AF05AF0  OR     5FFF5FFF = NOT  ABORT" OR"
 8       05AF05AF 5AF05AF0  XOR    5F5F5F5F = NOT  ABORT" XOR"
```

```
 9    LOOP  ." ."  ?TERMINAL ABORT" BREAK"    LOOP
10    CR . . . . .  CR ;
11 DECIMAL
12 XAOX
13
14
15

SCREEN #95
 0 \ MICROCODE TESTING       D@  D!
 1 DECIMAL
 2 VARIABLE VARC     4 ALLOT
 3 : XD@!
 4    1 2 3 4 5
 5     REPS 0 DO  #TIMES 0 DO
 6       I  J   VARC  D!
 7       VARC     D@   J = NOT  ABORT" D@!A"
 8                     I = NOT  ABORT" D@!B"
 9    LOOP  ." ."  ?TERMINAL ABORT" BREAK"    LOOP
10    CR . . . . .  CR ;
11
12  XD@!
13
14
15

SCREEN #96
 0 \ MICROCODE TESTING       D+
 1 DECIMAL
 2 : XD+
 3    1 2 3 4 5
 4     REPS 0 DO  #TIMES 1000 -     -1000 DO
 5       I I'  J I   D+   J I   I I'  D+     \ Test for consistency
 6       ROT = NOT  ABORT" D+A"
 7           = NOT  ABORT" D+B"
 8    LOOP  ." ."  ?TERMINAL ABORT" BREAK"    LOOP
 9    CR . . . . .  CR ;
10
11 XD+
12
13
14
15

SCREEN #97
 0 \ MICROCODE TESTING     DSWAP DDROP DDUP
 1 DECIMAL
 2 : XDD
 3    1 2 3 4 5
 4     REPS 0 DO  #TIMES 0 DO
 5       I I 100 +     1111 2222 DSWAP   DDUP
 6       I 100 +  = NOT  ABORT" DDA"    I = NOT ABORT" DDB"
 7       DSWAP DDROP
 8       I 100 + = NOT  ABORT" DDC"    I = NOT ABORT" DDD"
 9    LOOP  ." ."  ?TERMINAL ABORT" BREAK"    LOOP
10    CR . . . . .  CR ;
11
12 XDD
13
14
15
```

```
SCREEN #98
 0 \ MICROCODE TESTING       DUP  DROP   SWAP  OVER
 1 HEX
 2 : XDDSO
 3     1 2 3 4 5
 4      REPS 0 DO  #TIMES 0 DO
 5        I   -1234  OVER  SWAP  DROP
 6        = NOT  ABORT" DDSO"
 7     LOOP  ." ."  ?TERMINAL ABORT" BREAK"     LOOP
 8     CR . . . . .  CR ;
 9 DECIMAL
10 XDDSO
11
12
13
14
15

SCREEN #99
 0 \ MICROCODE TESTING      NEGATE  NOT
 1 HEX
 2 : XNN
 3     1 2 3 4 5
 4      REPS 0 DO  #TIMES 0 DO
 5        0 NEGATE  ABORT" NEGATEA"
 6     -56789876 NEGATE 56789876 = NOT  ABORT" NEGATEB"
 7      12341267 NEGATE -12341267 = NOT  ABORT" NEGATEC"
 8      I 1+ NOT NOT            NOT  ABORT" NOTA"
 9      0        NOT            NOT  ABORT" NOTB"
10    LOOP  ." ."  ?TERMINAL ABORT" BREAK"     LOOP
11     CR . . . . .  CR ;
12 DECIMAL
13 XNN
14
15

SCREEN #100
 0 \ MICROCODE TESTING       TOGGLE
 1 VARIABLE VARF
 2 HEX
 3 : XTOGGLE
 4     1 2 3 4 5
 5      REPS 0 DO  56781234 DUP VARF !   #TIMES 0 DO
 6        I OFF AND XOR      VARF  I TOGGLE
 7        DUP   OFF AND VARF C@ = NOT  ABORT" TOGGLE"
 8     LOOP  DROP ." ."  ?TERMINAL ABORT" BREAK"     LOOP
 9     CR . . . . .  CR ;
10 DECIMAL
11
12 XTOGGLE
13
14
15

SCREEN #101
 0 \ MICROCODE TESTING       U*
 1 HEX
 2 : XU*
 3     1 2 3 4 5
 4      REPS 0 DO  #TIMES 1000 -   -1000 DO
 5        I J U*   J  I  U*      \ Test for consistency
```

```
 6        RG1 = NOT  ABORT" U*A"
 7            = NOT  ABORT" U*B"
 8    LOOP  ." ."  ?TERMINAL ABORT" BREAK"    LOOP
 9    CR . . . . .  CR ;
10 DECIMAL
11 XU*
12
13
14
15

SCREEN #102
 0 \ MICROCODE TESTING      RRC
 1 HEX
 2 : XRRC
 3   . 1 2 3 4 5
 4     REPS 0 DO  #TIMES 0 DO
 5  0 1   RRC  0  = NOT ABORT" RRCA"  80000000 = NOT ABORT" RRCB"
 6  1 0   RRC  -1 = NOT ABORT" RRCC"   0    = NOT ABORT" RRCD"
 7 100 0  RRC  0  = NOT ABORT" RRCE"   80   = NOT ABORT" RRCF"
 8    LOOP  ." ."  ?TERMINAL ABORT" BREAK"    LOOP
 9    CR . . . . .  CR ;
10 DECIMAL
11 XRRC
12
13
14
15

SCREEN #103
 0 \ MICROCODE TESTING      RLC
 1 HEX
 2 : XRLC
 3    1 2 3 4 5
 4     REPS 0 DO  #TIMES 0 DO
 5      0 1 RLC    0  = NOT ABORT" RLCA"     1 = NOT ABORT" RLCB"
 6   80000000 0 RLC  -1  = NOT ABORT" RLCC"   0 = NOT ABORT" RLCD"
 7     80 0 RLC    0  = NOT ABORT" RLCE"   100 = NOT ABORT" RLCF"
 8    LOOP  ." ."  ?TERMINAL ABORT" BREAK"    LOOP
 9    CR . . . . .  CR ;
10 DECIMAL
11 XRLC
12
13
14
15

SCREEN #104
 0 \ MICROCODE TESTING      ADC                JMP=xxCA
 1 HEX
 2 : XADC
 3    1 2 3 4 5
 4     REPS 0 DO  #TIMES 0 DO
 5  0 0 1      ADC  0  = NOT ABORT" ADCA"     1 = NOT ABORT" ADCB"
 6     1234 5678 0 ADC  0  = NOT ABORT" ADCC"
 7                  68AC = NOT ABORT" ADCD"
 8 FFFFE234 5678 1 ADC  -1 = NOT ABORT" ADCE"
 9                  38AD = NOT ABORT" ADCF"
10 1235 FFFFEDCB 0 ADC  -1 = NOT ABORT" ADCG"
11                         = NOT ABORT" ADCH"
```

```
12    LOOP  ." ."  ?TERMINAL ABORT" BREAK"    LOOP
13    CR . . . . .  CR ;
14 DECIMAL
15 XADC

SCREEN #105
 0 \ MICROCODE TESTING       WORDSWAP    & BYTE-ROLL
 1 HEX
 2 : XBS
 3    1 2 3 4 5
 4     REPS 0 DO  #TIMES 0 DO
 5       12345678   WORDSWAP  56781234 = NOT  ABORT" WORDSWAP"
 6       12345678   BYTEROLL  78123456 = NOT  ABORT" BYTEROLL"
 7    LOOP  ." ."  ?TERMINAL ABORT" BREAK"    LOOP
 8    CR . . . . .  CR ;
 9 DECIMAL
10 XBS
11
12
13
14
15

SCREEN #106
 0 \ MICROCODE TESTING       >R R@  R>
 1 HEX
 2 : XRRR
 3    1 2 3 4 5
 4     REPS 0 DO  #TIMES 0 DO
 5      12345678  9876ABCD  >R   12345678 = NOT  ABORT" >R"
 6      R@                     9876ABCD = NOT  ABORT" R@"
 7      42345678   R>          9876ABCD = NOT  ABORT" R>A"
 8                             42345678 = NOT  ABORT" R>B"
 9    LOOP  ." ."  ?TERMINAL ABORT" BREAK"    LOOP
10    CR . . . . .  CR ;
11 DECIMAL
12 XRRR
13
14
15

SCREEN #107
 0 \ MICROCODE TESTING       U/MOD
 1 HEX
 2 : X-UM
 3    1 2 3 4 5
 4     REPS 0 DO  #TIMES 1000 -  -1000 DO
 5         I' I   I J +   U/MOD          I' I   I J +  U/MOD
 6         ROT      = NOT ABORT" U/MODA"
 7                  = NOT ABORT" U/MODB"
 8    LOOP  ." ."  ?TERMINAL ABORT" BREAK"    LOOP
 9    CR . . . . .  CR ;
10 DECIMAL
11 X-UM
12
13
14
15
```

```
SCREEN #108
 0 \ MICROCODE TESTING   LIT
 1 HEX
 2 : XLIT
 3    1 2 3 4 5
 4     REPS 0 DO  #TIMES 0 DO
 5       111111  111111  2222222 2222222  33333 33333  44444 44444
 6       = NOT ABORT" LITA"
 7       = NOT ABORT" LITB"
 8       = NOT ABORT" LITC"
 9       = NOT ABORT" LITD"
10    LOOP  ." ."  ?TERMINAL ABORT" BREAK"    LOOP
11    CR . . . . .  CR ;
12
13 DECIMAL
14 XLIT
15

SCREEN #109
 0 \ MICROCODE TESTING     <PICK>
 1 DECIMAL
 2 : X-PICK
 3    1 2 3 4 5
 4    1 REPS  DO   I NEGATE  -1 +LOOP   \ Leave stuff on stack
 5     REPS 1+ 1 DO  #TIMES 0 DO
 6         J PICK  J NEGATE  = NOT   ABORT" <PICK>"
 7    LOOP  ." ."  ?TERMINAL ABORT" BREAK"    LOOP
 8    REPS 0 DO DROP LOOP   \ Take stuff from stack
 9    CR . . . . .  CR ;
10
11 X-PICK
12
13
14
15

SCREEN #110
 0 \ MICROCODE TESTING     ROLL
 1 DECIMAL
 2 : X-ROLL
 3    1 2 3 4 5
 4     REPS 0 DO  #TIMES 0 DO
 5         I  0  J  -1   4 ROLL
 6          I = NOT   ABORT" <ROLL>A"
 7         -1 = NOT   ABORT" <ROLL>D"
 8          J = NOT   ABORT" <ROLL>C"
 9          0 = NOT   ABORT" <ROLL>B"
10    LOOP  ." ."  ?TERMINAL ABORT" BREAK"    LOOP
11    CR . . . . .  CR ;
12
13 X-ROLL
14
15

SCREEN #111
 0 \ MICROCODE TESTING     ?DUP
 1 DECIMAL
 2 : X-QDUP
 3    1 2 3 4 5
 4     REPS 0 DO  #TIMES 0 DO
 5         I  0 ?DUP   ABORT" ?DUPA"   I = NO  ABORT" ?DUPB"
```

```
 6         13131   I 1+ ?DUP
 7          I 1+ = NOT  ABORT" ?DUPC"   I 1+ = NOT ABORT" ?DUPD"
 8         13131 = NOT  ABORT" ?DUPE"
 9     LOOP  ." ."  ?TERMINAL ABORT" BREAK"     LOOP
10     CR . . . . .  CR ;
11
12 X-QDUP
13
14
15

SCREEN #112
 0 \ MICROCODE TESTING      S->D
 1 DECIMAL
 2 : X-SD
 3     1 2 3 4 5
 4      REPS 0 DO  #TIMES 0 DO
 5        134  S->D   ABORT" S->DA"  134 = NOT ABORT" S->DB"
 6       -134  S->D -1 = NOT ABORT" S->DC" -134 = NOT ABORT" S->DD"
 7     LOOP  ." ."  ?TERMINAL ABORT" BREAK"     LOOP
 8     CR . . . . .  CR ;
 9
10 X-SD
11
12
13
14
15

SCREEN #113
 0 \ MICROCODE TESTING      ABS
 1 DECIMAL
 2 : X-ABS
 3     1 2 3 4 5
 4      REPS 0 DO  #TIMES 0 DO
 5        I          ABS  I = NOT ABORT" ABSA"
 6        I NEGATE   ABS  I = NOT ABORT" ABSB"
 7     LOOP  ." ."  ?TERMINAL ABORT" BREAK"     LOOP
 8     CR . . . . .  CR ;
 9
10 X-ABS
11
12
13
14
15

SCREEN #114
 0 \ MICROCODE TESTING  CMOVE
 1 HEX
 2 CREATE TEXTD  10 ALLOT    CREATE TEXTE 10 ALLOT
 3 : X-CMOVE
 4     1 2 3 4 5
 5     11 TEXTE C!  22 TEXTE 1+ C!
 6      REPS 0 DO  #TIMES 0 DO
 7         0 TEXTD !
 8         TEXTE  TEXTD 1+  2 CMOVE
 9         TEXTD  @  00221100 = NOT ABORT" CMOVE"
10     LOOP  ." ."  ?TERMINAL ABORT" BREAK"     LOOP
11     CR . . . . .  CR ;
12 DECIMAL
```

```
13 X-CMOVE
14
15

SCREEN #115
 0 \ MICROCODE TESTING  2/
 1 DECIMAL
 2 : X-2/
 3    1 2 3 4 5
 4     REPS 0 DO  #TIMES 0 DO
 5        5  2/  2 = NOT ABORT" 2/A"
 6       -5  2/  -2 = NOT ABORT" 2/B"
 7       -6  2/  -3 = NOT ABORT" 2/C"
 8       -1  2/           ABORT" 2/D"
 9        I  2/  I 2/ = NOT ABORT" 2/D"
10    LOOP  ." ."  ?TERMINAL ABORT" BREAK"    LOOP
11    CR . . . . .  CR ;
12
13 X-2/
14
15

SCREEN #116
 0 \ MICROCODE TESTING   @  !
 1 DECIMAL
 2 VARIABLE VARXA
 3 : X-@!
 4    1 2 3 4 5
 5     REPS 0 DO  #TIMES 0 DO
 6         I VARXA !   VARXA @   I = NOT ABORT" !@A"
 7         J VARXA !   VARXA @   J = NOT ABORT" !@B"
 8    LOOP  ." ."  ?TERMINAL ABORT" BREAK"    LOOP
 9    CR . . . . .  CR ;
10
11 X-@!
12
13
14
15

SCREEN #117
 0 \ MICROCODE TESTING      DROT
 1 DECIMAL
 2 : XDROT
 3    1 2 3 4 5
 4     REPS 0 DO  #TIMES 0 DO
 5       1111111111111111. 2222222222222222. 3333333333333. DROT
 6       1111111111111111. D= NOT  ABORT" DROTA"
 7          3333333333333. D= NOT  ABORT" DROTB"
 8       2222222222222222. D= NOT  ABORT" DROTC"
 9     LOOP  ." ."  ?TERMINAL ABORT" BREAK"     LOOP
10    CR . . . . .  CR ;
11 XDROT
12
13
14
15

SCREEN #118
 0 \ MICROCODE TESTING   LSLN
 1 HEX
 2 : XLSLN
```

```
 3     1 2 3 4 5
 4      REPS 0 DO  #TIMES 0 DO
 5        1          01F     LSLN
 6        80000000       =     NOT     ABORT" LSLN"
 7     LOOP  ." ."  ?TERMINAL ABORT" BREAK"    LOOP
 8     CR . . . . .  CR ;
 9
10 DECIMAL
11 XLSLN
12
13
14
15

SCREEN #119
 0 \ MICROCODE TESTING   LSRN
 1 HEX
 2 : XLSRN
 3     1 2 3 4 5
 4      REPS 0 DO  #TIMES 0 DO
 5        8FFF0F0F 1F LSRN  1 = NOT ABORT" LSRN"
 6     LOOP  ." ."  ?TERMINAL ABORT" BREAK"    LOOP
 7     CR . . . . .  CR ;
 8
 9 DECIMAL
10 XLSRN
11
12
13
14
15

SCREEN #120
 0 \ MICROCODE TESTING      DLSL
 1 HEX
 2 : XDLSL
 3     1 2 3 4 5
 4      REPS 0 DO  #TIMES 0 DO
 5        80028001 90080004   DLSL
 6        20100009      = NOT  ABORT" DLSLA"
 7        00050002      = NOT  ABORT" DLSLB"
 8     LOOP  ." ."  ?TERMINAL ABORT" BREAK"    LOOP
 9     CR . . . . .  CR ;
10 DECIMAL
11 XDLSL
12
13
14
15

SCREEN #121
 0 \ MICROCODE TESTING   LSR
 1 HEX
 2 : XLSR
 3     1 2 3 4 5
 4      REPS 0 DO  #TIMES 0 DO
 5        84123456 LSR  42091A2B  = NOT ABORT" LSR"
 6     LOOP  ." ."  ?TERMINAL ABORT" BREAK"    LOOP
 7     CR . . . . .  CR ;
 8
```

```
 9 DECIMAL
10 XLSR
11
12
13
14
15

SCREEN #122
 0 \ MICROCODE TESTING    ASR
 1 HEX
 2 : XASR
 3    1 2 3 4 5
 4     REPS 0 DO  #TIMES 0 DO
 5       84122443 ASR  C2091221  = NOT ABORT" ASRA"
 6       44125434 ASR  22092A1A  = NOT ABORT" ASRB"
 7    LOOP  ." ."  ?TERMINAL ABORT" BREAK"    LOOP
 8    CR . . . . .  CR ;
 9
10 DECIMAL
11 XASR
12
13
14
15

SCREEN #123
 0 \ MICROCODE TESTING      <UNORM>
 1 HEX
 2 : XUNORM
 3    1 2 3 4 5
 4     REPS 0 DO   #TIMES 0 DO
 5       1   10   <UNORM>
 6          -0E    = NOT  ABORT" <UNORMA>"
 7        40000000 = NOT  ABORT" <UNORMB>"
 8    LOOP  ." ."  ?TERMINAL ABORT" BREAK"    LOOP
 9    CR . . . . .  CR ;
10 DECIMAL
11 XUNORM
12
13
14
15

SCREEN #124
 0 \ MICROCODE TESTING      <UDNORM>
 1 HEX
 2 : XUDNORM
 3    1 2 3 4 5
 4     REPS 0 DO   #TIMES 0 DO
 5       1 0  10   <UDNORM>
 6          -2E     = NOT  ABORT" <UDNORMA>"
 7     0  40000000 D= NOT  ABORT" <UDNORMB>"
 8    LOOP  ." ."  ?TERMINAL ABORT" BREAK"    LOOP
 9    CR . . . . .  CR ;
10 DECIMAL
11 XUDNORM
12
13
14
15
```

emonic

ass ALU unchanged) (default)
[ALU]
[ALU]
L[ALU]

Mode=1)
=notA
=AnorB
sed)
0
AnandB
notB
xorB
sed)
ed)
xnorB

andB
l
d)
rB
(default)

ode=0, CIN=0)

)
notB+1
)

Summary

SELECT
TION SELECT
NTROL
NTROL
LEXER SHIFT CONTROL

ION RAW INPUT

CARRY-IN
/DEST CONTROL

CODE SELECT BITS
RO-ADDRESS CONSTANT BITS 1-2
T-MPC CONTROL
DE BIT
T ADDRESS COUNTER BIT

Micro-operation field	Value	Mnemonic
(arithmetic ALU functions Mode=0, CIN=1)		
ALUFx	0	ALU=A+0
	1	(unused)
	2	ALU=A-B-1
	3	(unused)
	4	(unused)
	5	(unused)
	6	ALU=A-B
	7	(unused)
	8	(unused)
	9	ALU=A+B+1
	10	(unused)
	11	(unused)
	12	ALU=A+A+1
	13	(unused)
	14	(unused)
	15	(unused)
CIN	0	CIN=0 (default for logical ALU operations)
	1	CIN=1
DSHx	0	(no-op) (default
	1	SR[DLO]
	2	SL[DLO]
	3	DEST=DLO

CONDx	0	JMP=xx0
	1	JMP=xxC
	2	JMP=xxZ
	3	JMP=xxS
	4	JMP=xxL
	5	(unused)
	6	(unused)
	7	JMP=xx1
MADx	0	JMP=00x
	1	JMP=01x
	2	JMP=10x
	3	JMP=11x
INMPC	0	(no-op) (default)
	1	INC[MPC]

Micro-operation field	Value	Mnemonic
NDEC0	0	DECODE
	1	(no-op) (default)
INAD	0	(no-op) (default)
	1	INC[ADC]

APPENDIX C

WISC TECHNOLOGIES CPU/32 USER MANUAL

W I S C C P U / 3 2

P R E L I M I N A R Y

D O C U M E N T A T I O N

WISC Technologies, Inc.

La Honda, CA 94020

Preliminary: 2 June 1987

PREFACE

This is a preliminary document in which we have endeavored to provide you with the necessary and sufficient information to make full use of the CPU/32 product. At this time, it will be necessary for program developers to understand the Forth language to follow the source code provided. We look forward to having other languages implemented.

Hopefully, you will find this new experience as stimulating and rewarding as we have in developing it. You will find Phil Koopman's January 1987 BYTE article "Microcoded versus Hard-wired Control" and his April 1987 BYTE article "The WISC Concept" useful as additional support material. We recommend reading this document twice. On the first pass, do all the examples and don't worry about details that are not clear. On the second pass, concentrate on putting all the pieces together to form a clear over-all picture.

CPU/32 developers are licensed to include any or all of the run-time MVP-FORTH/32 program as a part of their products. Reasonable support for developers will be provided.

P.K. and G.B.H.

June 1987

CONTENTS

contained in patent application

Part One
OVERVIEW

Introduction

Most of today's common microprocessors fall into the category of "complex instruction set computers" (CISC). Efforts to improve the technology have lead, on one front, to "reduced instruction set computers" (RISC). The CPU/32 represents a third alternative, the stack-oriented "writable instruction set computer" (WISC).

The most significant potential offered by the WISC CPU/32 is the new way of addressing a problem found in classical CISC and RISC architectures: that of semantic mismatch between the computational needs of an application program and the CPU's instruction set. This mismatch limits the ultimate throughput of such systems. The WISC CPU/32 allows an opcode's semantic content to be harmonious with the hardware structure and application program's needs, vastly improving throughput.

The concept of the writable instruction set computer, incorporated in the WISC CPU/32, will be a new experience for most users. Complete access to the processor's logic control lines is available. Custom opcodes can be implemented easily and integrated into applications. In contrast to writable "control store" options available on some computers, the CPU/32's instruction set has a simple format for easy modification. The writable instruction set can be employed to make an application's fundamental operations the native code of the WISC processor.

Another significant feature of the CPU/32 is the use of independent hardware stacks for handling operands and subroutine return addresses. The use of these stacks simplifies the CPU/32 hardware by eliminating the need for decoding operand location information - all operands are implicit in the op-code. In addition, hardware stacks greatly increase the speed of the frequent subroutine calls demanded by modern structured programming. A unique feature of the CPU/32 is the combination of an opcode with a subroutine call, subroutine return, or unconditional branch within each instruction. This allows processing subroutines and branches in parallel with other operations, further increasing the speed of subroutine calls. In many cases, subroutine calls actually take no time at all.

The documentation includes everything that was used to produce the hardware and software systems: complete circuit drawings, details of the microcode format, and source code for all the software. This may be more than you need, but it is our intention to be as open with the product as possible. Note that the boards, documentation, and software are copyrighted. No restriction is placed on the use of this software with applications using the CPU/32. A patent has been applied for.

Not all users of the WISC CPU/32 need to master the most detailed information that is provided. Executable code and complete source listings are included for a micro-assembler, a cross-compiler, and the completely functional development language. All are ready to run, permitting productive use of the system from the beginning. Eventually, you will want to study and understand all facets of the WISC CPU/32 system. Our experience teaches that understanding all parts of a problem and its solution is the ultimate approach to problem-solving. For tutorial purposes, you may skip to the parts of this documentation which interest you most.

Getting Started

Install the Hardware

To install the CPU/32, first thread the two ribbon cables coming out of the CPU/32 box through the back of the IBM PC/AT or compatible's box. Then attach them to the headers marked "CN1" and "CN2" on the HOST card. The color-marked end of the cable sockets should go nearest the "CN1" and "CN2" markings on each cable. If you have to pull the cable back over the connector to get the cable out of the PC box, the connector is on backwards.

********** WARNING **********
Placing the ribbon cables on incorrectly will damage the HOST card. It will not, however, damage the PC or the CPU/32 box.
****

Next, place the HOST card into any slot on the IBM PC/AT or compatible. The HOST card will work equally well in an 8-bit slot or a 16-bit slot. The HOST card will also work in some IBM PC's and IBM PC XT's, but operation is not guaranteed. This is because many of the earlier PC's have a hardware bug that produces a significant amount of "noise" on the bus address lines which violates published IBM PC design specifications.

The CPU/32 comes configured to address the PC's I/O port numbers 300 through 307 (hex). If this causes a conflict in your system, see the section on reconfiguring the HOST card I/O ports.

Distribution Software

Several diskettes are provided with the CPU/32. We suggest that you make backups of them using the DOS DISKCOPY program before proceeding any further.

The disk labelled "WISC System" is the run-time disk that brings up MVP-FORTH on the CPU/32. To run MVP-FORTH on the CPU/32, boot DOS in the normal manner, then insert the "WISC System" disk in the A drive. From the DOS prompt, type:

```
A:CPU32
```

This places you in an MVP-FORTH kernel that has the microcode assembler, cross-compiler, and other utilities pre-loaded. To load the microcode memory and the first 32k of program memory (which contains the kernel and floating-point package for the CPU/32), turn on the CPU/32 box's power and type in:

```
DR0  LOAD-ALL
```

You will see a series of "happy faces" indicating that the loading process is progressing. Note that the CPU/32 memory image is stored as Forth screens at the high end of the "WISC System" disk, so no additional DOS files should be added to this disk to avoid overwriting the Forth screens.

The CPU/32 is now ready for operation. To transfer system control to the CPU/32, type in:

```
CPU32
```

You will see a "(CPU/32) OK" prompt. You are now running a 32-bit version of MVP-FORTH directly from the CPU32. Try typing in VLIST or any other function desired. To return control to the PC, type in BYE and notice the return to the normal Forth "OK" prompt.

A note about CPU/32 Forth: DR0, DR1, etc. are not defined in MVP-FORTH/32. In order to change the default disk drive, return to the PC Forth with a BYE, perform the disk assignment (e.g. DR1), then return to the CPU32. This decision was made to isolate the CPU32's Forth from the many possible effects of enhanced disk interfaces for DOS files, etc.

Troubleshooting

If everything did not go smoothly, first double-check that the ribbon cables are on snugly and in the proper orientation. If the PC shows a "START.." and then dies, the CPU/32 is not responding properly to the "CPU32" command. If this happens, or anytime that you want to verify correct operation of the system, run the single-stepping diagnostics.

The "Diagnostics" disk contains two sets of diagnostics. The first set is the single-stepping diagnostics that uses the PC host to exercise all the resources of the CPU/32 in a slave mode. To run these diagnostics, run the CPU32.COM file from the "WISC System" disk as described above, but do not do a LOAD-ALL. Next, place the diagnostics disk in the A drive, and from the PC's Forth, type in:

```
DR0  2 LOAD
```

The diagnostics will then exercise each portion of the system, displaying messages as the tests progress. If the tests complete without a failure message, the CPU/32 has passed the single-stepping diagnostics. (If a failure is generated, see the section on problem resolution below.)

The second set of diagnostics on this disk is the stress test. To run the stress test, run MVP-FORTH/32 using the CPU32--LOAD-ALL--CPU32 sequence as described previously. From the "(CPU/32) OK" prompt, type in:

```
DRO 3 LOAD
```

This test first does a memory addressability and stuck-bit failure test, then repetitively exercises the critical microcoded definitions. The test takes a while to run. The CPU/32 will run an infinite number of floating point stress tests at the end of the sequence. Pressing any key on the keyboard will terminate testing. If your system passes this test successfully, then everything should be operating correctly.

Problem Resolution

Your system was completely tested before shipping using the same floppy disks you have. If any problems arise during installation or operation of your system, please complete the following steps, in order, to resolve the problem.

1) Please check for obvious problems with the hardware: unplugged power cables, connectors, etc.

2) Please re-read the applicable sections of the documentation.

3) If the hardware does not pass diagnostics, or seems unreliable in operation, ensure that all connecting ribbon cables inside the CPU/32 box are securely plugged in, and that all cards are properly seated in the motherboard.

4) If you still can not resolve the problem after re-reading the documentation, please call us any time at (415) 747-0760.

Host PC Software

The PC host communicates with the WISC CPU/32 system with up to 8 I/O ports. Any program or language that can address these ports can run the CPU/32. For example, the IN and OUT functions of BASIC, or user-written procedures in C, Pascal, etc. may be used to control the system, even when the CPU/32 is running Forth. Currently, we are providing all development and support software in MVP-FORTH.

The Host PC software is organized as a micro-assembler, a cross-compiler, and the source code to cross-compile the MVP-FORTH/32 kernel. Complete source code for this software is provided in Forth screen format on the disks marked "Cross-Compiler/Microcode" and "CPU/32 Kernel".

Cross-Compiler/Microcode Disk

The "Cross-Compiler/Microcode" disk contains the source code for the cross-compiler, microcode assembler, and the microcoded primitives used by MVP-FORTH/32. To reload these screens, run the FORTH.COM file from the "WISC System" disk under DOS by typing:

```
FORTH
```

This is the MVP-FORTH kernel which has the assembler and double precision word set from the MVP-FORTH math package (MVP-FORTH series volume 3) pre-loaded. Place the "Cross-Compiler/Microcode" disk in the A drive. Turn on the CPU/32, then type in:

```
DRO  2 LOAD
```

Screen 2 loads all the source code, compiles the microcode directly into the CPU/32 microcode memory, and creates the basic Forth dictionary structure with headers for the microcode words on the CPU/32. When the loading is completed, screen 2 does a SAVE-ALL, which saves the new memory images onto the highest screens of the A disk.

When the loading is completed, you may wish to do a SAVE-FORTH to a DOS files disk to save the revised version of the cross-compiler/micro-assembler. File XCOMP.COM on the "WISC System" disk corresponds to the host PC Forth system you have at this point.

The most straightforward way to add or modify microcoded primitives is by following the above procedure, then recompiling the MVP-FORTH/32 kernel as described in the next section.

MVP-FORTH/32 Kernel Disk

The "MVP-FORTH/32 Kernel" disk contains the source code for the cross-compiled MVP-FORTH/32 kernel. To reload the kernel, continue on from the previous section using the XCOMP.COM MVP-FORTH kernel used to load the cross-compiler and microcode from the host PC. Place the "MVP-FORTH/32 Kernel" disk in the A drive, then type in:

```
2 LOAD
```

This will start the kernel cross-compilation process. The cross compiler actually builds the target image in the CPU/32's program memory. The messages "START.." followed by ." END" indicate a transfer of control to the CPU/32 to execute functions during the compilation process (see the section on the usage of "" and "" for more information about this.)

When the cross-compilation is completed, transfer control to the CPU/32 and complete compilation of the kernel by typing in:

```
CPU32
3 LOAD
```

Do a VLIST to satisfy yourself that the dictionary is complete, then return control to the host and save the new CPU/32 memory image by typing in:

```
BYE
SAVE-ALL
```

If desired, you may do a SAVE-FORTH to save a copy of the MVP-FORTH kernel for later use. The host Forth kernel you are using now corresponds to the file CPU32.COM on the "WISC System" disk, and is well-suited for experimenting with single-stepping CPU/32 functions from the host PC.

Tests and Demonstrations

The CPU/32 comes with some demonstration disks in addition to the disks already talked about. While the function and the content of the demonstration disks may change, they are all set up with the same format for consistency.

Each demonstration disk should be placed in the A drive after running the CPU32 command from DOS. Each disk has a Forth memory image that should be transferred to the CPU/32 using the LOAD-ALL command. Each disk has screen 2 reserved as a host loading screen. From the host PC, type in:

```
DRO 2 LOAD    ( Automatically does a LOAD-ALL)
CPU32
```

Next, load the application program from the CPU/32 by typing in:

```
3 LOAD
```

When the load is completed, instructions for running the demonstration will be displayed. Typing in HELP will re-display the instructions.

Most of the graphics demonstrations require an Enhanced Graphics Adapter (EGA). The other demonstrations assume the use of a Color Graphics Adapter (CGA) with 80 columns of text. Changing the segment number from B800 hex to B000 hex in the host PC's source code will make the non-graphics demonstrations work on a monochrome display.

General Notes on MVP-FORTH/32

The differences between MVP-FORTH/32 and the functional descriptions in ALL ABOUT FORTH are minor. To make the most of the software tools available, you should become familiar with MVP-FORTH. In most cases, the high-level and low-level functions are the same. The kernel's implementation is different in that microcoded definitions replace the machine language used in the original MVP-FORTH.

The following notes cover the major differences between MVP-FORTH and MVP-FORTH/32. Further details will come from study of the source code.

* The arrangement of the header fields is different.

* The link field comes before the name field.

* There is no code field. In most cases, the PFA is the same as the CFA.

* The value for variables and constants is stored 4 bytes after the PFA.

* The name field count is a 4-byte value with flag bits in the highest byte.

* All strings including the name fields are blank-padded to fill up an evenly divisible by 4 number of bytes. The name field counts still reflect the actual name length.

* For colon definitions, the array of pointers to lower level subroutines is automatically executed in sequence. A DOCOL operation is not needed.

* Each instruction consists of an opcode value in the highest 9 bits and an address for subroutine call or jump (or a subroutine exit bit) in the low 23 bits. Forth sequences of opcodes followed by subroutine calls (and/or subroutine exits) are automatically compacted into a single instruction by the compiler.

* All operations that are 16 bit operations in MVP-FORTH work on 32 bits in MVP-FORTH/32

* All 32-bit memory operations (including @ and !) and instructions must be aligned on word boundaries. This means that the memory address used must be evenly divisible by 4. The compiler automatically performs this alignment in virtually all cases.

* A single screen buffer is used (little, if any, difference will be noted).

* The user variables are at low memory addresses and do not change addresses with disk buffer changes.

* The stack address space is not accessible with @ and !. Therefore, the MVP-FORTH implementation for .S has been modified.

The elemental editor command FP is available from the CPU/32. The standard Forth line editor, however, is available from the host. Identical screens can be read by either system. Change the editor to suit your own needs and desires.

MVP-FORTH/32 has no disk utilities. There are no direct disk functions. Only system function calls to the currently active drive on the host are required. This approach eliminates confusion that results when using a DOS interface or other enhancement to the host PC's Forth.

There is no assembler - with the supplied microcode the CPU/32's native language is Forth. High speed words may be written in microcode if desired, corresponding to the assembly language used in other Forth systems.

In addition to the included standard wordset from MVP-FORTH, the MVP-FORTH/32 system has a number of additional words in microcode. Some of these are simple primitives for common combinations of words and math functions. Others are completely new words that are useful in the demonstration programs.

Other Forth Dialects

We realize the limitations of the standard distrubuted version of MVP-FORTH, especially when using a hard-disk based system under DOS. We have used an optimized version of MVP-FORTH known as LIB-FORTH for our development efforts. Among other things, LIB-FORTH has support for DOS screen files, faster execution speed, and a full-screen editor. LIB-FORTH is in the

public domain and is available from Mountain View Press. LIB-FORTH functions correctly, but is not a fully polished, fully supported product. If you wish to use LIB-FORTH, contact us for a copy of the CPU/32 files in the DOS screens format.

We plan to develop a more powerful Forth that allows the use of one or more CPU/32's on a single host PC within an integrated software environment. We are also pursuing porting other languages to the CPU/32. Until these products are ready, however, MVP-FORTH remains the only the fully supported host language.

Part Two
HARDWARE DESIGN

Introduction
Specifications
Capabilities
- Operating Speed
- General Architecture

The Host Interface
- Master/Slave Modes
- Write Ports
- Read Ports
- The Status Register - STATUS
- The Host Request Register - HOST

The Data Stack
- The Data Stack Pointer - DP
- The Data Stack - DS
- Restrictions

The Arithmetic Logic Unit Complex
- The Arithmetic Logic Unit and Multiplexer - ALU
- The Data Hi Register - DHI
- The Bus Latch - LATCH
- The Data Low Register - DLO
- Multiplication and Division
- Restrictions

The Return Stack
- The Return Stack Pointer - RP
- The Return Stack - RS
- Restrictions

Memory Addressing and Subroutine Control
- The RAM Address Latch - ADDRESS-LATCH
- The Address counter - ADDRESS-COUNTER
- The Next Address Register - NAR
- The Page Register - PAGE
- Restrictions

Program Memory
- Program Memory - RAM
- Byte Addressing Program Memory
- Restrictions

Instruction Latch
- The Instruction Latch - IL

The Micro-program Counter, Condition Codes & Interrupts
- The Micro-Program Counter - MPC
- The Condition Code Register - CCR
- The Interrupt Register - FLAGS

Introduction

This section describes the operation of the hard[illegible] detail. Appendix A contains a summary of the host [illegible] interface. The tables in Appendix B contain the a[illegible]ary of the micro-instruction format and micro-assembler mnemonics. Appendix C contains a summary of some of the CPU/32's important characteristics. Appendix D contains complete schematics for the CPU/32. The reader should immediately look at Appendix D, Figures 1 and 2 to see the CPU/32 block diagrams before proceeding.

Specifications

* Clock cycle: 150 ns (jumper and oscillator selectable) micro-instruction cycle.

* Throughput: Approximately 3.5 million Forth operations per second (varies depending on instruction mix)

* Architecture: 32-bit data paths. Writable 32-bit horizontal microcode memory. Two independent 32-bit hardware stack memories. Overlapped micro-instruction execution and fetch. Overlapped macro-instruction execution and fetch, with direct execution of subroutines calls.

* Host Computer Requirements: One full-size expansion slot, with two ribbon cables to the CPU/32 box. All CPU/32 cards may be relocated inside the host PC/AT case if desired. Requires an IBM PC/AT or compatible. May work on some IBM PC compatibles if the address bus meets published PC design specifications.

* Technology: Most logic is 74ALS series TTL chips. The ALU is implemented with 74F181 chips using carry-lookahead logic. Stack and microcode memories use 4k x 4, 35 ns static RAM chips. Program memory uses 32k x 8, 120 ns static RAM chips.

* Hardware stacks: Independent 4k-element, 32-bit data and return stacks with hardware stack pointers. Stack pointers can be incremented or decremented in parallel with other operations.

* Microcode Memory: 4k x 32-bit words divided into 512 opcode pages of up to 8 instructions each. Microcode memory is changeable by the host PC.

* Program Memory: 1 megabyte of 120 ns static program memory standard. Expandible to up to 8 megabytes of static memory. Capability to address up to 2 gigabytes of dynamic program memory in 8 megabyte pages. Memory is organized into 32-bit words, accessible as either bytes or words.

* Interrupts: CPU/32 interrupts are available for data stack underflow/overflow, return stack underflow/overflow and host attention request.

* DMA Memory Transfer: CPU/32 program memory may be loaded from or to host PC memory (including video memory) using the host PC DMA controller.

Capabilities

Operating Speed

The WISC CPU/32 runs at a 33% duty cycle micro-instruction clock of 150 ns (6.67 MHz.) as derived from a 20 MHz crystal. Since most primitives are only 2 clock cycles long, this gives a best-case operating speed of 3.33 MOPS million high level operations per second (MOPS). In addition, subroutine calls, exits, and unconditional branches that can be combined with opcodes are processed at zero time cost. In actual programs, the average primitive will be somewhat more than 2 clock cycles, but the availability of zero-cost subroutines will probably yield an effective instruction rate of better than 3 MOPS including subroutine call instructions. This figure will be adversely affected in programs that perform many multiplications, divisions, or other complicated operations.

Various benchmarks have shown speed increases of 5 to 10 times over LIB-FORTH (a speed-optimized version of MVP-FORTH) running on an 80286 at 8 MHz, with zero-wait-state memory. Programs that perform a lot of 32-bit arithmetic or have a higher than average number of subroutine calls will perform even better than this 5 to 10 times improvement. Programs with a lot of floating point math will not perform as well, but will still be faster than the 80286.

General Architecture

The WISC CPU/32's general architecture is shown in Appendix D, figures 1 and 2. The CPU/32 uses 32-bit horizontal microcode to drive a 32-bit, stack oriented processor. The CPU/32 is capable of overlapped micro-instruction fetching and execution, and overlapped macro-instruction fetching and execution. Several functional units are implemented in hardware and connected by a single 32-bit, 3-state data bus. Additionally, the instruction addressing and decoding data paths are connected 32-bit memory address and memory data paths.

Since the micro-assembler is the primary means of controlling the hardware resources, control signals will be described in terms of their micro-assembler mnemonics. The function of each micro-operation will be described in the corresponding hardware section.

The Host Interface

The host interface is shown at the top of figure 2. The host adapter card plugged into the PC host contains logic to convert 8-bit PC data to 32-bit CPU/32 data, as well as logic to control the operation of the CPU/32.

Master/Slave Modes

The CPU/32 has two modes: Master and Slave. In Master mode, the CPU/32 runs programs with a continuously cycling clock. Typically, the host PC monitors the CPU/32 status register, waiting for a non-zero value, then returns the CPU/32 to Slave mode for servicing.

To enter Slave mode, use the STOP command (better yet, use the RESET-BOARD command, in case the CPU/32's internal state is unknown). In the Slave mode, the CPU/32 waits for IN and OUT commands from the host to receive or send information. It will also respond to DMA transfers on DMA channel 3.

While in slave mode, the CPU/32 responds to the port addresses shown in Appendix A. In the normal configuration, the port addresses are in the range 300 to 307 hex. Jumpers J1 through J14 on the host interface card may be changed to modify this address to any otherwise unused bank of 8 ports as shown in Appendix A.

Write Ports

Ports 300 through 307 are used as "write" ports with the assembly language OUT command. Each port accepts 8 bits of data, and may be written to using the WRITE0...WRITE7 words from the micro-assembler. Each WRITE word accepts an 8-bit value from the stack and writes it to the appropriate port.

Port 0 is used to write data to the 32-bit data bus. Four bytes are written successively (lowest order byte first, highest last) to this port. As the fourth byte is written, the CPU/32's clock is automatically cycled once to clock the results in to the destination determined by the contents of the micro-instruction register. The X! micro-assembler word takes a 32-bit number from the data stack and writes it to the CPU/32 data bus. In DMA operations, a WRITE0 is simulated by the host interface card hardware for each data transfer, automatically arranging each quartet of DMA'ed bytes into 32-bit words for memory storage.

Port 1 is used to write data to the 32-bit micro-instruction register (MIR). It requires 4 successive WRITE1 operations, just like port 0, but always places the results in the MIR. The micro-assembler word MIR! takes a 32-bit value and places it in the MIR. Typically, the MIR value is set using a micro-assembler construction (accessed from the normal FORTH vocabulary within the CPU32 version of the MVP-FORTH kernel on the host PC) such as:

```
>> DEST=DLO ;SET
```

which automatically determines bit values for the micro-instruction. The ">>" is a prefix that indicates the start of a micro-instruction definition, and ";SET" indicates the end of the definition, and directs the micro-assembler to load the assembled micro-instruction into the micro-instruction register using the MIR! primitive. Subsequently typing in:

```
12345678. X!
```

will store the 32-bit quantity 12345678 in the DLO register. To read back the value, enter the following commands:

```
>> SOURCE=DLO ;SET   X@   D.
```

Note that all values used are 32-bit integers, since the CPU/32 is a 32-bit machine.

Port 2 single-steps the CPU/32's clock. WRITE2 has the same effect as the single-step that automatically occurs with X!, except the host does not drive the bus data lines. The data written to the port is ignored. The micro-assembler word CYCLE writes a dummy value to port 2.

Port 3 starts the board in Master mode. The data written to the port is ignored. The micro-assembler word GO writes a dummy value to port 3.

Port 4 stops the board and places it in Slave mode. The data written to the port is ignored. The micro-assembler word STOP writes a dummy value to port 4.

Port 5 sets the board in DMA mode. The data written to the port is ignored. The micro-assembler word SET-DMA writes a dummy value to port 6. Typically this port is not directly used by application programs.

Port 6 resets the data bus sequencer used by WRITE0, WRITE1, READ0, and READ1. It also takes the board out of DMA mode. As a matter of good programming practice, use RESET-SEQ command, which writes a dummy value to port 6, before each series of reads or writes. The words X@, X!, MIR@, and MIR! do this automatically

Port 7 writes to the 8-bit service register on the CPU/32. This register may be written even when the CPU/32 is in Master mode. If interrupts are enabled, writing to this register will generate a CPU/32 interrupt. The micro-assembler word PCREQ takes an 8-bit value from the stack and writes it to port 7.

Read Ports

Ports 300 through 302 are used as "read" ports with the assembly language IN command. Each port returns 8 bits of data, and may be read using the READ0...READ2 words from the micro-assembler. Each READ word returns a 16-bit value with the high 8 bits zeroed to the stack.

Port 0 is used to read data to the 32-bit data bus. Four bytes are read successively (lowest order byte first, highest last) from this port. As the fourth byte is read, the CPU/32's clock is automatically cycled once to adjust any counters that may be affected by the MIR contents. The bus source is

determined by the contents of the MIR. The X@ micro-assembler word reads the contents of the data bus and returns a 32-bit value to the data stack. In DMA operations, a READ0 is simulated by the host interface card hardware for each data transfer, automatically retrieving successive 32-bit bus bytes to form quartets of 8-bit values.

Port 1 is used to read data from the 32-bit micro-instruction register (MIR). It requires 4 successive READ1 operations, just like port 0, but always returns the result from the MIR. The micro-assembler MIR@ returns a 32-bit value from the MIR. Typically, MIR@ is only useful for diagnostic purposes.

Port 2 reads the CPU/32's 8-bit status register. STATUS is the micro- assembler word that returns a 16-bit value with the high 8 bits zeroed. The status register typically has a zero value while the CPU/32 is in Master mode. The CPU/32 sets the status register to a non-zero value to indicate a request to be returned to Slave mode for service.

The Status Register - STATUS

The STATUS register is an 8 bit register that allows the CPU/32 to communicate requests for service to the host while in master mode. Typically, the STATUS register is 0, with a non-zero value indicating a request for one of 255 possible user-defined services from the host. The STATUS register is set using the DEST=STATUS micro-operation, and is queried from the host using the STATUS word. There is no way to read the STATUS register from the CPU/32. An example of setting a STATUS value is:

```
>> DEST=STATUS ;SET   12. X!
STATUS .    ( Returned value is 12)
```

The Host Request Register - HOST

The 8-bit HOST register allows the host PC to make requests for attention to the CPU/32. The host PC may write to the host register using the PCREQ word. The CPU/32 may read the HOST register value using the SOURCE=FLAGS micro-operation:

```
HEX
45 PCREQ
>> SOURCE=FLAGS ;SET  X@  DROP OFF AND .
     ( Display lowest 8 bits of value, giving 45)
DECIMAL
```

The upper bits of the FLAGS register contain interrupt status information described in a later section.

The Data Stack

The data stack is one of two hardware stacks in the CPU/32. In the MVP-FORTH/32 implementation, the data stack contains the Forth data stack elements (except for the top-most stack element, which is contained in the DHI register described in the next section.) The data stack consists of the data stack memory and the data stack pointer.

The Data Stack Pointer - DP

The data stack pointer (DP - not to be confused with the MVP-FORTH user variable dictionary pointer DP) is a 12-bit up/down counter that may be incremented, decremented, or loaded on each micro-instruction. The following micro-assembler commands load the DP, read it back, and show results of increment and decrement operations:

```
>> DEST=DP ;SET  200. X!
>> SOURCE=DP ;SET  X@ D.       ( Display 200)
>> INC[DP] ;SET    CYCLE
>> SOURCE=DP ;SET  X@ D.       ( Display 201)
>> SOURCE=DP  DEC[DP] ;SET  X@ D. X@ D. X@ D.
          ( Display 201, 200, 199)
```

Notice that X@ cycles the clock on each read operation. The DP increment or decrement operations take place at the end of each clock cycle. This means that the data stack itself may be read, then the DP incremented or decremented at the end of the very same cycle.

The Data Stack - DS

The data stack (DS) is a 4k by 32-bit fast memory. The DS value may be read or written on each cycle, with the DP value changed at the end of each cycle. In MVP-FORTH/32, the DP points to the second to topmost stack element, with an INC[DP] acting as a "pop" operation, and a DEC[DP] followed on the next clock cycle with a DEST=DS operation acting as a "push" operation. The following sequence of commands pushes two items on the DS, then pops them off again:

```
>> DEC[DP]  ;SET  CYCLE
>> DEST=DS  DEC[DP] ;SET  11111. X!  ( Push 11111)
>> DEST=DS  ;SET          22222. X!  ( Push 22222)
>> SOURCE=DS  INC[DP] ;SET  X@ D. X@ D.
          ( Pop 22222 then 11111)
```

Restrictions

Any time the DP value is between 0 and 1FF hex, a data stack underflow/overflow interrupt is generated. The DS positions between 0 and 1FF are still usable if interrupts are masked, however. This provides run-time stack size checking with no speed penalty. In a sophisticated software environment, the DP interrupt could be used to page portions of a larger than 4k-word data stack in and out of the DS memory. The starting value for DP with an empty stack must be initialized to FFF hex.

The Arithmetic Logic Unit Complex

The Arithmetic Logic Unit complex consists of the Data Hi register (DHI), bus data latch (LATCH), arithmetic and logic unit (ALU) with associated multiplexer, and Data Low register (DLO). All these have 32-bit data paths. The ALU multiplexer and DLO registers may be connected to form a 64-bit shifting mechanism.

The Arithmetic and Logic Unit and Multiplexer - ALU

The ALU performs 32-bit addition, subtraction, and Boolean logic operations on one or both of two inputs, denoted the A-side and B-side inputs. The result of the arithmetic or logical operation are passed through the multiplexer and may be shifted right or left one bit, rolled right 8 bits (with the lowest 8 bits moving to the highest 8 bit position), or passed through unchanged.

ALU operations are broken up into logical operations and arithmetic operations (corresponding to the 2 mode selects of the 74F181 chip). Logical ALU micro-operations leave the carry in bit CIN uncommitted, and result in an indeterminate carry-out bit. Allowable logical operations are:

FUNCTION	DESCRIPTION
ALU=A	Pass A side through unchanged
ALU=B	Pass B side through unchanged
ALU=notA	One's complement of A side
ALU=notB	One's complement of B side
ALU=AorB	A logical "or" B
ALU=AandB	A logical "and"
ALU=AxorB	A logical "xor" B
ALU=AnorB	A logical "nor" B
ALU=AnandB	A logical "nand" B
ALU=AnxorB	A logical "xnor" B
ALU=0	Force ALU outputs to all 0
ALU=-1	Force ALU outputs to all 1 (two's complement negative 1)

All arithmetic operations are performed with two's complement integers. These operations automatically set the carry-in bit value as required to generate functions, and result in a valid carry-out value at the end of the clock cycle in which they are used. Allowable arithmetic operations are:

FUNCTION	DESCRIPTION
ALU=A+0	Pass A side through unchanged (set carry-out to 0)
ALU=A+1	Add 1 to A side
ALU=A-1	Subtract 1 from A side
ALU=A+A	Multiply A side by 2
ALU=A+A+1	Multiply A by 2 and add 1
ALU=A+B	Add A side to B side
ALU=A+B+1	Add A side to B side plus 1
ALU=A-B	Subtract B side from A side
ALU=A-B-1	Subtract B side from A side, then subtract 1
ALU=AornotB+1	(Special purpose function) If the A side is 0, this takes the two's complement of the B side

Some examples of ALU functions are given below (note that the word ;DO is equivalent to the sequence ;SET CYCLE):

```
>> ALU=B  DEST=DHI  ;SET  5. X!  ( Set DHI to 5)
>> DEST=DLO         ;SET  3. X!  ( Set DLO to 3)
>> SOURCE=DLO ALU=A+B DEST=DHI ;DO  ( Add DLO to DHI)
>> SOURCE=DHI       ;SET  X@ D.  ( New DHI value is 8)
>> ALU=A+1 DEST=DHI ;DO          ( Add 1 to DHI)
>> SOURCE=DHI       ;SET  X@ D.  ( New DHI value is 9)
```

The output of the ALU function is passed through the multiplexer. The SL[ALU] micro-operation shifts the ALU output one bit left, discarding the highest bit, and inserting the highest bit from the DLO register into the lowest bit position of the multiplexer output. The SR[ALU] micro-operation shifts the ALU output right one bit, discarding the lowest bit (or shifting it into the DLO register if SR[DLO] is used) and shifting in the carry-in bit described in the next paragraph. The ROLL[ALU] operation does a right-rotate by 8 bits. Some examples are:

```
>> DEST=DLO          ;SET  0. X!   ( Zero highest bit )
>> ALU=B DEST=DHI ;SET  400. X! ( Store 400 in DHI)
>> ALU=A  SL[ALU] DEST=DHI ;DO
>> SOURCE=DHI      ;SET  X@  D.     ( Result is 800 )
>> ALU=A  CIN=0 SR[ALU] ;SET CYCLE  CYCLE
>> SOURCE=DHI      ;SET  X@  D.     ( Result is 200 )
HEX
>> ALU=B DEST=DHI ;SET  12345678. X!
>> ALU=A  ROLL[ALU] ;DO
>> SOURCE=DHI ;SET X@ D.      ( Result is 78123456 )
DECIMAL
```

The carry-in bit is used for both arithmetic and shifting operations. CIN=0 or CIN=1 generate a 0 or 1 value for the carry-in bit respectively for shifting operations. ALU arithmetic functions automatically generate their own carry bit values.

The output of the ALU (NOT the multiplexer) serves as the source of the sign bit and zero bit for conditional branching. Since the default operation for the ALU is ALU=A, and since the DHI register value is clocked on each cycle, the zero and sign conditions always apply to the value in the DHI register

The Data Hi Register - DHI

The DHI register is the 32-bit register that is attached to the ALU A side and is the receiver of the ALU output on each clock cycle. The default micro-assembler ALU function is ALU=A, meaning that if no ALU function is specified, the DHI contents simply flow through the ALU on each clock cycle. Since DHI is clocked each cycle, the micro-operation DEST=DHI is only used for clarity, and may be used freely even when another bus destination is specified in the micro-instruction.

The DHI register may be driven onto the bus using the SOURCE=DHI micro-operation. Note that the "old" value of DHI may be placed on the bus even while the ALU is computing a new value to be placed in DHI on the same clock cycle:

```
>> ALU=B DEST=DHI ;SET  123. X!  ( Set DHI to 123)
>> SOURCE=DHI  ALU=0 DEST=DHI ;SET
X@ D.          ( Fetch 123 & clock 0 into DHI )
X@ D.          ( Fetch the 0 from DHI )
```

The DHI register may hold any quantity during the course of a microcoded primitive's execution. By convention, the DHI register always contains the top stack element at the start or end of a word when executing MVP-FORTH/32.

The Bus Latch - LATCH

The LATCH is a 32-bit transparent latch that connects the data bus to the B side of the ALU. In normal operation, the LATCH is transparent: it simply allows the data bus to flow directly into the ALU. Thus, in the previous examples that stored data in DHI by using the ALU=B construct, the latch just passed data through.

However, there is another function for the LATCH. At the end of each clock cycle, the LATCH captures whatever data was on the bus (whether it was used by the ALU or not). Then, if the subsequent bus source is SOURCE=DHI on the next cycle, the LATCH retains and supplies to the ALU B side the data from the previous clock cycle. This is extremely useful for exchanging the contents of the DHI register with the contents of any other system resource. For example, exchanging the current DS value and DHI without using the LATCH would be accomplished by:

```
>> SOURCE=DHI DEST=DLO ;DO
>> SOURCE=DS  ALU=B  DEST=DHI ;DO
>> SOURCE=DLO  DEST=DS ;DO
```

However, using the LATCH, this exchange can be accomplished in only two cycles:

```
>> SOURCE=DS  DEST=LATCH ;DO
>> SOURCE=DHI DEST=DS  SOURCE=LATCH ALU=B DEST=DHI ;DO
```

On the first cycle, the DS value is simply placed on the bus for retention in the LATCH. If desired, the DS value could also be written to any other bus destination, or even used by the ALU to compute a new DHI value before the exchange occurs. On the second clock cycle, DHI is written to the DS, and the retained LATCH value is written to DHI, accomplishing the exchange. Note that the micro-operation SOURCE=LATCH is just for readability (just like DEST=DHI). The micro-operation SOURCE=DHI is what causes the LATCH to retain its data. The LATCH will retain its data indefinitely, as long as SOURCE=DHI is used in consecutive clock cycles.

The Data Low Register - DLO

The DLO register is a 32-bit shift register that is used both as a temporary holding register and as a low half of a 64-bit shifting mechanism. As a holding register, SOURCE=DLO and DEST=DLO allow storing any 32-bit value in DLO and reading it back:

```
>> DEST=DLO ;SET  123456. X!
>> SOURCE=DLO ;SET X@ D.  ( Display 123456)
```

When used as a shift register, DLO may be shifted independently using SR[DLO] or SL[DLO], and may be combined with the ALU multiplexer to form a 64-bit shift register if desired. DLO may not be set and shifted in the same clock cycle, but it may be read and then shifted (after reading) in the same micro-cycle. When shifting left, the lowest bit of DLO is set by the CIN bit. When shifting right, the highest bit of DLO is set by the output of the lowest bit of the ALU (not the ALU multiplexer):

```
>> DEST=DLO ;SET   4. X!   ( Set DLO to 4)
>> CIN=1 SL[DLO] ;DO       ( Shift in a 1)
>> SOURCE=DLO ;SET X@ D.   ( Result is 9)
>> DEST=DLO ;SET  -8. X!   ( Set DLO to -8)
>> ALU=-1 SR[DLO] ;DO      ( Shift right, high bit -1)
>> SOURCE=DLO ;SET  X@ D. ( Result is -4)
```

The single-stepping diagnostics have good examples of shifting a single bit across all 64 bits of DLO and the ALU/multiplexer/DHI circle in both directions.

Multiplication and Division

Two very special modes of operation of the LATCH and ALU are provided for multiplication and division. When the special bus source designation MULTIPLY is used, the ALU multiplexer performs an arithmetic shift right operation on the partial sum (which works by feeding in the arithmetic carry-out bit into the multiplexer.) In addition, the LATCH retains previous data. This capability allows performing a 32-bit multiply in one cycle per bit, plus a few cycles of overhead for setup. See the microcode definition of U* for an example of a shift and conditional add algorithm using this capability.

The special bus source designation DIVIDE not only shifts the logical complement of the ALU sign bit into the DLO shift left input, but also manipulates the ALU function inputs to provide either an ALU=A+B or an ALU=A-B operation as required. This results in a one-clock cycle per bit (plus setup overhead) non-restoring division algorithm for unsigned numbers as demonstrated by the microcoded definition for U/MOD.

Restrictions

Due to time constraints, the zero condition value is not valid after an arithmetic operation (but is valid after logical operations). If necessary, allow the DHI value to cycle through the ALU using the default ALU=A operation, then do a microcoded branch based on the Z bit. Alternately, when comparing two numbers for equality, use the ALU=AxorB function to avoid this limitation. In MVP-FORTH/32, since OBRANCH and other words may test the Z bit on the very first micro-instruction, this means that only logical operations may be performed on the ALU on the ENDing micro-instruction of any word.

Since arithmetic operations set the CIN bit, do not specify an explicit CIN value for shifting while doing an arithmetic operation. Clever micro-programmers may be able to use the CIN value set by the arithmetic operation as the shifting input.

Be sure to specify a SOURCE=DHI micro-operation when using the LATCH for a retained value. The micro-operations SOURCE=LATCH and DEST=LATCH are used only for readability; they do not enforce this constraint.

The Return Stack

The return address stack is the second of two hardware stacks in the CPU/32. In the MVP-FORTH/32 implementation, the

return stack contains the Forth return addresses and loop variables. The return stack implementation consists of the data stack memory and the data stack pointer.

The Return Stack Pointer - RP

The return stack pointer (RP) is a 12-bit up/down counter that may be incremented, decremented, or loaded on each micro-instruction. The following micro-assembler commands load the RP, read it back, and show results of increment and decrement operations:

```
>> DEST=RP ;SET  200. X!
>> SOURCE=RP ;SET  X@ D.    ( Read back 200)
>> INC[RP] ;SET    CYCLE
>> SOURCE=RP  DEC[RP] ;SET  X@ D. X@ D. X@ D.
            ( Read back 201, 200, 199)
```

Notice that X@ cycles the clock on each read operation. The RP increment or decrement operations take place at the end of each clock cycle. This means that the return stack itself may be read, then the RP incremented or decremented at the end of the very same cycle. The astute reader will notice that the RP works exactly the same as the DP.

The Return Stack - RS

The data stack (RS) is a 4k by 32-bit fast memory. The RS value may be read or written on each cycle, with the RP value changed at the end of each cycle. In MVP-FORTH/32, the RP points to the topmost stack element, with an INC[RP] acting as a "pop" operation, and a DEC[RP] followed on the next clock cycle with a DEST=RS operation acting as a "push" operation. The following sequence of commands pushes two items on the RS, then pops them off again:

```
>> DEC[RP]  ;DO
>> DEST=RS  DEC[RP] ;SET  11111. X!   ( Push 11111)
>> DEST=RS  ;SET          22222. X!   ( Push 22222)
>> SOURCE=RS  INC[RP] ;SET  X@ D. X@ D.
            ( Pop 22222 then 11111)
```

The return stack is very similar in operation to the data stack, except that the data to and from the return stack may either be routed from/to the system data bus, or may routed from the address counter or to the address latch and address counter. This added routing capability is crucial in that it supports parallel subroutine processing in conjunction with other operations. It does have a slight drawback in that due to extra chip enabling logic delays, the return stack is too slow to be written through the ALU on a single clock cycle.

Restrictions

Any time the RP value is between 0 and 1FF hex, a return stack underflow/overflow interrupt is generated. The RS positions between 0 and 1FF are still usable if interrupts are masked, however. This provides run-time stack size checking with no speed penalty. In a sophisticated software environment, the RP interrupt could be used to page portions of a larger than 4k-word return stack in and out of the RS memory. The starting value for RP with an empty stack must be initialized to FFF hex.

Since the RS is used for subroutine call processing, in general SOURCE=RS and DEST=RS should never be used on the very first or very last (ENDing) micro-instruction within a microcoded word definition. Also, DEST=RP and SOURCE=RP should never be used during these micro-instructions since they can produce results that are difficult to predict. The discussion on instruction decoding contains some special cases when these rules do not apply.

Due to timing constraints, the RS value may not be written through the ALU to the DHI register directly, nor may it be written to RAM. The RS value may be captured by the LATCH for use in the subsequent clock cycle, may be written to the DS, and may be written to DLO (or to DP or RP for that matter). RS may be written to the address latch or the address counter on a single clock cycle, but the address latch value will be written too late for RAM to be available for use on the very next clock cycle - wait one clock cycle before accessing RAM. The hardware subroutine return function that copies the RS contents to the address latch does allow RAM to be read on the subsequent clock cycle, however, since it uses different control circuitry.

Memory Addressing and Subroutine Control

The data paths used for memory addressing and subroutine control are shown in the bottom half of figure 1. The memory addressing data paths connect to the return stack (RS), the memory address counter (ADDRESS-COUNTER), the memory address latch (ADDRESS-LATCH), the next address register (NAR), and the memory page register (PAGE). The ADDRESS-COUNTER, ADDRESS-LATCH, and NAR all manipulate the lowest 23 bits of program data, while the PAGE register holds the next highest 8 bits of data, giving a 31 bit address space consisting of 256 pages of 8 megabytes. The 32nd memory address bit is ignored by the program memory cards.

There are several important limitations to using the memory address path resources, most of which are designed to avoid interfering with the automatic subroutine processing logic.

The RAM Address Latch - ADDRESS-LATCH

The RAM address latch (ADDRESS-LATCH, not to be confused with LATCH which is attached to the ALU) is a 23-bit transparent latch that is used to address memory for all functions except automatic instruction decoding. Its outputs are enabled to address program memory during all micro-instructions except the first and last within a microcoded word. Data written to ADDRESS-LATCH becomes a valid memory address approximately half-way through the cycle in which it is written with DEST=ADDRESS-LATCH. This leaves enough time for program memory to be addressed and read or written during the very next clock cycle:

```
123456.  100. RAM!    ( Store 123456 at addr 100)
>> DEST=ADDRESS-LATCH ;SET  100. X!  ( addr 100 )
>> SOURCE=RAM ;SET  X@ D.    ( Fetch value 123456)
```

Once set, the ADDRESS-LATCH retains its value until the next instruction is fetched by the DECODE micro-operation, or until a

DEST=ADDRESS-LATCH or DEST=ADDRESS-COUNTER is used again.

The Address Counter - ADDRESS-COUNTER

The ADDRESS-COUNTER is a 23-bit count-by-4 register which has the lowest two bits always forced to zero. Its primary purpose is to increment by one word address the values to be stored on the return address stack, thus generating a return address that points to the instruction after the subroutine call. The ADDRESS-COUNTER may be set using the DEST=ADDRESS-COUNTER micro-operation. This micro-operation causes whatever data is on the bus to flow through the ADDRESS-LATCH and into the ADDRESS-COUNTER. This means that a DEST=ADDRESS-COUNTER micro-operation also implicitly performs a DEST=ADDRESS- LATCH operation.

The ADDRESS-COUNTER may be incremented by 4 during any but the first and last micro-instructions within a microcoded word definition. The increment, which is specified by the INC[ADC] micro-operation, occurs at the end of the clock cycle, and may be combined with a SOURCE=ADDRESS-COUNTER micro-operation to read the value present before incrementing. An interesting use of the ADDRESS-COUNTER is to perform an INC[ADC] followed by the next cycle with a SOURCE=ADDRESS-COUNTER DEST=ADDRESS-LATCH. This provides a capability to step through words of program memory for reading or writing. An example of using the ADDRESS-COUNTER is:

```
RESET-BOARD    ( Among other things, this
                 zeros the page register)
>> DEST=ADDRESS-COUNTER ;SET  100. X!
                ( Also sets ADDRESS-LATCH to same value)
>> INC[ADC] ;DO
>> SOURCE=ADDRESS-COUNTER ;SET X@ D.  ( Value of 104)
```

The Next Address Register - NAR

The next address register (NAR) is the closest thing there is to a program counter in the CPU/32. The NAR is a 23-bit holding register for the address of the next instruction. Each CPU/32 instruction specifies the next instruction to be executed with a subroutine call, subroutine exit, or unconditional branch. This means that the NAR does not increment to determine instruction addresses, but rather is reloaded at the end of the instruction fetching process. The section on instruction decoding will clarify this point.

The NAR is loaded using the DEST=DECODE instruction, which is only useful for purposes discussed in the section on instruction decoding. The NAR may not be read back directly, but its contents may be accessed in single-stepping mode by performing a dummy instruction decode that copies its contents into the ADDRESS-COUNTER. The NAR's outputs are enabled for program memory addressing only during the first and last micro-instructions within a microcoded word definition.

The Page Register - PAGE

The PAGE register is used to extend the memory address range beyond the 23 bits provided by ADDRESS-COUNTER, ADDRESS-LATCH, and NAR. The PAGE register is set to zero by the RESET-BOARD command, and may only be changed with the DEST=PAGE micro-

operation. The PAGE register contents are always appended to the high end of the ADDRESS-COUNTER value, and may be examined by reading the ADDRESS-COUNTER:

```
HEX
>> DEST=ADDRESS-COUNTER ;SET 0045678C. X!
          ( Set ADDRESS-COUNTER 23-bit value)
>> DEST=PAGE ;SET  74800000. X!
          ( 8-bit PAGE register value is E9 hex,
            starting with bit 23 of value)
>> SOURCE=ADDRESS-COUNTER ;SET X@ D.
          ( Returned value is 74C5678C. )
DECIMAL RESET-BOARD
```

Since the page register contents are used for generating bits 23-31 of the memory address, the page register selects which of 256 non-overlapping banks of 8 megabytes is in use at any given time. Inter-segment memory accesses may be accomplished by saving the PAGE register value for the program, setting PAGE to accessed the desired memory location, then restoring the PAGE register value before the next instruction is decoded.

Restrictions

As discussed in the return stack section, the sequences SOURCE=RS DEST=ADDRESS-LATCH and SOURCE=RS DEST=ADDRESS-COUNTER are valid, but result in a memory address which is set up too slowly to provide valid RAM accessing during the next clock cycle. Wait one clock cycle before reading or writing RAM when using these constructions.

Never perform a SOURCE=RS, DEST=RS, DEST=PAGE, DEST=DECODE, SOURCE=ADDRESS-COUNTER, DEST=ADDRESS-COUNTER, DEST=ADDRESS-LATCH or INC[ADC] during the first or last clock cycles of a microcoded instruction. Doing so will interfere with instruction decoding and crash whatever program is executing. Exceptions: on a SPECIAL instruction (which is guaranteed not to contain a subroutine call or return) SOURCE=RS, DEST=RS, and SOURCE=ADDRESS-COUNTER may be used in the first micro-instruction; and on an instruction that is known to be a subroutine call (such as the Forth LIT word implementation in MVP-FORTH/32) SOURCE=ADDRESS-COUNTER may be used to access the value being written to the return stack on the first micro-instruction. The section on instruction decoding explains the reason for these limitations.

The ADDRESS-COUNTER must be restored to its original value before the last micro-instruction of a word if it has been used for memory addressing, otherwise subroutine calling on the subsequent word will not work properly. If a DEST=DECODE micro-operation is used, the ADDRESS-COUNTER must be set to the memory address used to fetch the macro-instruction. A typical two micro-instruction sequence to accomplish this is:

```
... DEST=ADDRESS-COUNTER ...  ( Set address to decode)
                              ( set ADDRESS-LATCH too)
... SOURCE=RAM DEST=DECODE ...( Decode instruction)
```

Program Memory

The CPU/32 comes with a standard 1 megabyte of program memory occupying two memory expansion boards of 512k bytes each. Each board takes 16 pieces of Hitachi HM62256 32k by 8 bit 120 ns static memory chips (or equivalent), arranged as 128k words of 32-bits. Up to a total of sixteen static memory expansion boards may be used with the CPU/32 by setting jumpers on each board as shown in Appendix A. In the future, dynamic memory boards will allow addressability of up to 2 gigabytes of program memory.

Program Memory = RAM

The program memory is organized as up to 2 gigabytes of 32-bit words, and is addressable as either bytes or words. Words must always be aligned on word boundaries (evenly divisible by 4 memory addresses). For example, memory addresses 0, 4, 8, 12, and 16 are aligned on word boundaries, while addresses 1, 2, 3, 5, 6, and 7 are not. Program memory is accessed by first setting the memory address in the ADDRESS-LATCH, then performing a SOURCE=RAM or DEST=RAM operation. In order to write and then read a 32-bit RAM value:

```
>> DEST=ADDRESS-LATCH ;SET  100. X!
>> DEST=RAM ;SET 12345678. X!
          ( Store 12345678 at address 100)
>> SOURCE=RAM ;SET X@ D.  ( Fetch 12345678)
```

Program memory data takes one cycle to become valid after an address is established. However, since the ADDRESS-LATCH outputs are valid approximately halfway through the cycle in which they are set, RAM may be read or written on the very next micro-instruction.

In word access mode, the lowest two bits of the ADDRESS-LATCH value are ignored treated as if zero, forcing access on even word boundaries.

Byte Addressing Program Memory

Program memory may be accessed as an array of bytes instead of full-words. If SOURCE=RAM-BYTE or DEST=RAM-BYTE are used, the lowest 8 bits of the data bus are mapped onto the appropriate RAM byte in the word being addressed. If writing to RAM, the highest 24 bits of the data bus are ignored. If reading from RAM, the highest 24 bits of the data bus are set to zero. Byte accessing to RAM is controlled at the memory chip level, so a read/modify/write operation is not required to change a single memory byte:

```
HEX
>> DEST=ADDRESS-LATCH ;SET  102. X!  ( Address 102)
>> DEST=RAM ;SET  12345678. X!
          ( Set 32-bit word 100 to 12345678 )
>> SOURCE=RAM-BYTE ;SET X@ D.
          ( Byte 102 value is 34)
>> DEST=RAM-BYTE   ;SET  99. X!
>> SOURCE=RAM      ;SET  X@ D.
          ( New 32-bit word 100 value is 12995678)
DECIMAL
```

Restrictions

The output of the RAM is too slow to be fed directly through the ALU on a single clock cycle. It may, however, be captured by the LATCH for use on the cycle subsequent to the READ=RAM or READ=RAM-BYTE micro-operation. As discussed in the section on the return stack, do not read or write RAM directly from the RS.

RAM is available for use on the cycle immediately after a DEST=ADDRESS-LATCH or DEST=ADDRESS-COUNTER micro-operation, except when the source is the RS (in which case a one clock cycle wait is required). Feeding the output of RAM directly to the ADDRESS-LATCH may appear to work in simple test cases, but will lead to unreliable operation since the program memory's data path will be directly feeding back to its address path.

Instruction Latch

The Instruction Latch - IL

The instruction latch (IL) is an 11 bit transparent latch that is used to contain a 9 bit opcode and 2 bits of subroutine control information for an instruction. Opcode information is written to the IL from bits 23-31 of the RAM data bus, and subroutine control information is written from bits 0-1 of the RAM data bus (according to the CPU/32's standard instruction format.) The micro-operation to do this is DEST=DECODE, which also writes bits 2-22 to the NAR. Since the IL is a transparent latch, information written to it is available as an input to the micro-program counter during the cycle in which it is written. This has important implications for instruction decoding as discussed in a later section.

The IL contents may be read to the data bus using the SOURCE=MPC micro-operation. This is only useful for diagnostics and interrupt processing. When reading back the IL, bits 23-31 and bits 0-1 contain the IL information, and bits 8-19 contain micro-program counter information.

The Micro-Program Counter, Condition Codes & Interrupts

The CPU/32 has a micro-program "counter" (MPC) to address micro-program memory. The value that addresses micro-program memory actually consists of an assembly of 12 bits from different sources. Bit 0 (the lowest bit) is the output of the condition code multiplexer. Bits 1 and 2 are constants from the micro-instruction register. Bits 3-6 come from a 4-bit counter within the MPC, and bits 7-11 come from a register in the MPC.

The Micro-Program Counter - MPC

The micro-program counter is a 9 bit register whose lowest 4 bits may be incremented if desired. When an instruction is decoded, the opcode is directly set into the MPC. This means that micro-program memory is accessed as 512 opcodes, with each opcode having a page of 8 micro-instructions available to it.

Within a page of micro-program memory, the MPC value is left undisturbed while the lowest three bits are manipulated by JMP=xxx micro-operations within the page as described below.

If a microcoded operation requires more than 8 words of micro-program memory, the INC[MPC] micro-operation may be used to increment the lowest 4 bits of the MPC to jump to the next page. The INC[MPC] must be used on the next to last micro-instruction executed within a page, and the last micro-instruction must contain a JMP=xxx micro-operation to the micro-instruction of the next page that is to be executed on the succeeding clock cycle.

The MPC may be read with the SOURCE=MPC micro-operation. Bits 11-19 contain the current MPC value. Other bits contain other information about the IL and micro-program memory address. In particular, bits 9-10 contain the two micro-instruction register bits used to form bits 1-2 of the micro-program memory address.

The Condition Code Register - CCR

The condition code register (CCR) is used to hold status information for use in conditional microcode branches. It is updated at the end of every clock cycle, and is not directly accessible to high level language programs. The value of one of the 8 possible condition codes is selected by the 3-bit micro-instruction register conditional branching field to form the lowest bit of the micro-program memory address for the subsequent micro-instruction. The 8 condition code values are:

CCR number	micro-assembler notation	description
0	0	always has a 0 value
1	C	logical inverse of ALU carry-out valid only when an arithmetic ALU operation is used during the preceding clock cycle. A carry-out of 0 sets this condition code to 1.
2	Z	logical inverse of ALU zero test. When the ALU output is zero, this bit is set to 0.
3	S	Sign bit of ALU output. When ALU output bit 31 (before shifting) is 1, this bit is 1.
4	L	Lowest bit of DLO register. When DLO bit 0 is a 1, this bit is a 1. Bit reflects contents of DLO before any shifting or loading operation during a clock cycle.
5		Unused - reserved for future expansion
6		Unused - reserved for future expansion
7	1	Always 1

The CCR value may be read for diagnostic purposes in single-step mode by selecting the CCR addressing number 0-7 with the appropriate bits of the MIR, and using the SOURCE=MPC micro-operation. Bit 8 contains the current CCR value for the condition selected. Other bits contain other information about the IL and micro-program memory address.

The section on micro-instruction processing contains an explanation of how to use condition codes in a microcoded word definition.

The Interrupt Register - FLAGS

The CPU/32 supports interrupts for DS underflow/overflow, RS underflow/overflow, and host service request, as well as a software interrupt capability. The interrupt status register forms the highest 8 bits of the FLAGS register (the lowest 8 bits are formed by the host service register). The FLAGS register may be set using the DEST=FLAGS micro-operation and may be read using the SOURCE=FLAGS micro-operation. Each bit of the FLAGS register has a specific meaning:

FLAGS

BIT	MEANING
24	Data stack pointer underflow/overflow
25	Return stack pointer underflow/overflow
26	Host service request
27	Dynamic RAM parity error (currently unused)
28	Software interrupt #2
29	Software interrupt #1
30	Software interrupt #0
31	Interrupt mask bit 1=masked, 0=unmasked

Interrupts are inactive when their corresponding bits are 0. These bits must be set to 0 using a DEST=FLAGS micro-operation when clearing an interrupt. The effect of any interrupts is masked whenever bit 31 of the FLAGS register is 1, but the values may still be read if desired using the SOURCE=FLAGS micro-operation. Bits 24 through 30 may be set to any value desired using the DEST=FLAGS micro-operation to reset interrupt bits, simulate interrupts or generate software interrupts. Care should be taken not to over-write pending interrupt bit when setting the FLAGS register.

If interrupts are unmasked, any interrupt bit that is non-0 will force an opcode of 1 to be executed at the next instruction decoding cycle, and will disable any instruction fetching, subroutine calling, exiting, and unconditional branching operations that may be in progress. Opcode 1 is defined in MVP-FORTH/32 to mask interrupts and invoke an interrupt handler.

Restrictions

Since only the lowest 4 bits of the MPC form a counter, the INC[MPC] micro-operation must not be used from an MRAM page number that is 1 less than evenly divisible by 16 (i.e. don't use INC[MPC] from MRAM pages 15, 31, 47, 63, etc.)

Any microcoded word that masks the interrupts by setting bit 31 of the FLAGS register must wait for a clock cycle before executing a DECODE micro- operation to avoid generating another interrupt.

Micro-Program Memory & Micro-Instruction Register

Micro-Program Memory - MRAM

The MRAM is a 4k word by 32-bit memory used to store all microcode in the system. As discussed previously, MRAM is conceptually broken up into 512 pages of 8 words, with each page corresponding to an opcode. MRAM may be accessed in Slave mode by using the SOURCE=MRAM and DEST=MRAM micro-operations. However, setting up the correct micro-program address involves getting the appropriate page number into the MPC and the appropriate offset value and condition code selection value into the micro-instruction register. Therefore, use of the micro-assembler words MRAM! and MRAM@ is highly recommended:

```
12345678. 503 MRAM!   ( Store value in location 503)
503 MRAM@ D.   ( Retrieve value 12345678)
```

The Micro-Instruction Register - MIR

The MIR holds the micro-instructions that actually make the CPU/32 perform all other functions. The MIR is a 32-bit register that sends control signals throughout the CPU/32 hardware. The MIR is broken up into fields as shown in Appendix B. Each group of micro-assembler directives controls one or more fields within the MIR.

The MIR may be written using the MIR! micro-assembler word, and read for diagnostic purposes using the MIR@ word:

```
HEX
12345678. MIR!    ( Store value in MIR)
MIR@ D.           ( Returned value is 12345678)
>> DEST=DLO ;SET
MIR@ D.           ( Microcoded bit pattern from ;SET )
              ( is fetched, value is 40D00200)
DECIMAL
```

The MIR is loaded on each clock cycle while in Master mode from the contents of MRAM. However, during slave mode, it is only loaded by the MIR! instruction, not by clock cycling from CYCLE, X@, or X!. This allows retention of a valid micro-instruction during repeated Slave mode operations.

Restrictions

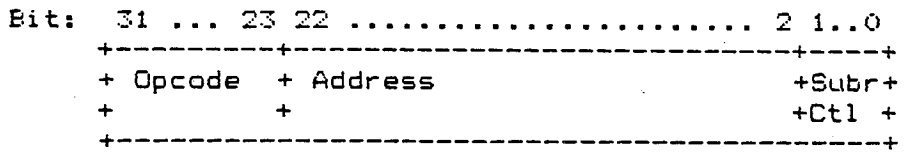
MRAM may not be modified by the CPU/32 in Master mode, since the MRAM supplies instructions to maintain Master mode operation. If the CPU/32 needs to modify MRAM while executing a program, set up a service request to the host to make the desired change in Slave mode, then return the CPU/32 to Master mode.

Program & Micro-Program Execution

Up until this point we have examined only the functions of each resource of the CPU/32 in Slave mode. This section describes how instructions and micro-instructions are executed in Master mode.

Instruction Format

The CPU/32 instruction format consists of three fields: two bits of subroutine control information in bits 0-1 (lowest two bits), a program address field in bits 2-23, and an opcode in bits 24-31 (highest nine bits).



The Subroutine control bits allow each instruction to be either an unconditional jump (value of 0), a subroutine return (value of 1), or a subroutine call (value of 2). The value 3 is illegal for these 2 bits.

The 21-bit address field is the word address that has an implicit lowest 2 bit value of 0 appended to form a word-aligned 23-bit byte address for addressing the next instruction to be executed. This field is used for subroutine calls and unconditional branches. The field is ignored during subroutine return operations.

The opcode is a 9-bit value that directly addresses a page of MRAM. Opcode 0 is reserved for use as a No-op, and opcode 1 is reserved for use as the interrupt service opcode. The opcode is executed as the subroutine call, subroutine return, or unconditional branch is processed, making subroutine processing "free".

Instruction Processing

High level instructions, called macro-instructions, consist of the 32-bit opcode plus address and control format just described. The macro-instructions are fetched and decoded during a three-clock-cycle sequence, called the decoding sequence. The sequence is started by the DECODE micro-operation, which is referred to as the DEC0 cycle. The END micro-assembler word is used to denote the cycle after DECODE (referred to as the DEC1 cycle). The first clock cycle of the subsequent instruction is the last cycle of the decoding sequence, known as the DEC2 cycle.

The DEC0 cycle, initiated by the DECODE micro-operation, must always occur in the next-to-last micro-instruction executed within a microcoded word. During the DEC0 cycle, the interrupt flag registers are examined for pending interrupts, and the MPC is clocked with the value of the Instruction Latch (IL). If a non-masked interrupt is pending, the MPC value is forced to 1 instead of the IL value and the call and exit control bits are set to 0, causing an interrupt service word to start execution.

The DEC1 cycle, denoted by the END micro-assembler word, must always occur in the last micro-instruction executed within a microcoded word. During the DEC1 cycle, the ADDRESS-COUNTER is incremented to form a subroutine return address pointing to the next sequential word. If a subroutine call or unconditional branch is specified by the instruction in the IL, the NAR outputs are enabled to drive the RAM address bus. If a subroutine call is being processed, then an INC[RP] is automatically performed.

If a subroutine exit is being processed, then the ADDRESS-LATCH is loaded from the RS, and the ADDRESS-LATCH outputs are used to drive the RAM address bus. The END micro-assembler word forces a JMP=000 micro-operation to ensure that the 0th offset micro-instruction is the first micro-instruction executed by the next opcode.

The DEC2 cycle occurs during execution of the first micro-instruction of the word that was held in the IL during the DEC0 cycle. During the DEC2 cycle, if an interrupt is not being processed, the ADDRESS-COUNTER is loaded from whatever value is present on the RAM address bus, and the IL and NAR are loaded with whatever value is on the program RAM data bus. All these loads occur at the end of the clock cycle. Additionally, if an unconditional branch is being processed, the NAR is used to drive the RAM address bus. If a subroutine call is being processed, the NAR is used to drive the RAM address bus and the RS is written with the contents of the ADDRESS-COUNTER. If a subroutine exit is being processed, then the contents of the ADDRESS-LATCH register are used to drive the RAM address bus and the RP is incremented.

Each microcoded instruction must be at least two clock cycles long. Since the IL is a transparent latch that contains valid data before the end of a clock cycle, the opcode may be read from RAM during the DEC2 cycle and clocked into the MPC in the same clock cycle if desired. This means that the DEC2 cycle of one instruction may occur simultaneously with the DEC0 cycle of the next instruction.

In the instruction decoding process, the next instruction to be executed is being loaded into NAR and IL as the first micro-instruction of the current instruction is being executed. Use of the micro-operation DEST=DECODE allows changing the contents of the NAR, IL, and subroutine control bits during the middle of a microcoded word, just as if those registers had been loaded with the normal decoding sequence. The MVP-FORTH/32 word OBRANCH and other words exploit this fact.

There are some micro-operations that, while prohibited under most circumstances, are essential for efficient program execution for special cases. Be very careful when exploiting these special cases, and do not rely on a single test to ensure correct operations, since some violations of usage rules manifest themselves as relatively infrequent random failures:

1) SOURCE=RAM will place the contents being fetched to the NAR and IL during the DEC2 cycle onto the data bus. This can be used to fetch a data cell instead of an instruction by using a subroutine call to the data cell. A manual subroutine exit function using a DEST=DECODE must then be performed.

2) SOURCE=ADDRESS-COUNTER may be used during the DEC2 cycle to fetch the contents about to be saved on the RS if the instruction being executed is guaranteed to be a subroutine call. This is accomplished by having a special compiling word for the microcoded word in the Forth kernel.

3) SOURCE=RS, DEST=RS, SOURCE=RP, DEST=RP may all be used during the DEC2 cycle if the instruction being executed is guaranteed to be an unconditional jump. This is accomplished by denoting the microcoded word as SPECIAL with the micro-assembler.

Micro-Instruction Processing

Each CPU/32 opcode is implemented as a series of two or more micro-instructions. Each opcode starts on the MRAM page number corresponding to the opcode number 0-511. Each opcode may take one to sixteen consecutive MRAM pages as required.

Within each page, the order of the micro-instructions is unimportant, except that the first micro-instruction on the opcode's first page must be located at offset 0. The micro-assembler assumes that instructions are to be processed in sequential order (i.e. 0,1,2,3,4,5,6,7) unless instructed otherwise with the JMP=xxx micro-instruction.

The JMP=xxx micro-instruction, where the "xxx" may be replaced with a large variety of binary bit patterns, allows non-sequential conditional branching within and between micro-program memory pages. As an example, consider the microcode sequence:

```
3 ::  ALU=A+1 DEST=DHI ;;
4 ::  ALU=A+1 DEST=DHI JMP=110 ;;
6 ::  ALU=A+1 DEST=DHI ;;
```

In this sequence, the characters "3 ::" replace the familiar ">>" which we have been using to load the MIR. Any number between 0 and 7 followed by a "::" begins a micro-instruction definition and sets the offset within the current micro-program memory page (set by the OPCODE: command described in part 3). The command ";;" replaces ";SET" and actually writes the micro-instruction to MRAM in the current page instead of to the MIR. In this example the DHI register is incremented three times, with non-sequential flow of control to MRAM offsets 3, 4, and 6. The "JMP=110" micro-operation specifies the lowest three bits of the next micro-instruction to be 110, which is 6 decimal. Other unconditional jump micro-operations are: JMP=000, JMP=001, ... , JMP=111.

Conditional jump micro-operations allow the lowest bit of the subsequent micro-instruction's address to be determined by a condition code, resulting in a two-way branch. For example:

```
0 :: ALU=A  DEST=DHI ;;
1 :: JMP=11S      DECODE ;;
6 :: ALU=0  DEST=DHI   END ;;
7 :: ALU=-1 DEST=DHI   END ;;
```

jumps from location 1 to location 6 if the ALU sign bit is 0 (binary offset 110) and location 7 if the ALU sign bit is 1 (binary offset 111). This microcoded word tests the DHI register and sets it to 0 if the DHI value is non-negative, but sets DHI to -1 if the initial DHI value is negative. Notice that there are two possible last micro-instructions for execution, both of which have an END micro-assembler directive.

During micro-program execution, the MPC/MRAM/MIR arrangement forms a two-stage micro-program pipeline. The micro-instruction to be executed next is being fetched while the MIR controls the system with the current micro-instruction. This means that conditional branching is accomplished using the condition code

that was set at the end of the cycle before the branch is executed. This means that, in the above example, the JMP=11S acted on the value at the ALU output at the end of the previous micro-instruction (in this case, the micro-instruction at offset 0.)

Allowable microcode branches are:

JMP=000	JMP=001	JMP=010	JMP=011
JMP=100	JMP=101	JMP=110	JMP=111
JMP=00Z	JMP=01Z	JMP=10Z	JMP=11Z
JMP=00C	JMP=01C	JMP=10C	JMP=11C
JMP=00S	JMP=01S	JMP=10S	JMP=11S
JMP=00L	JMP=01L	JMP=10L	JMP=11L

see the section on condition codes for an explanation of the codes "Z", "C", "S", and "L".

In MVP-FORTH/32, each microcoded definition is defined to place the top of stack element in the DHI register (without using an arithmetic ALU operation) at the END micro-instruction of each word. That means that the very first micro-instruction of each word may execute a conditional branch using the Z or S condition on its very first micro-instruction.

Bus Source and Destination

In the preceding sections, we have been taking for granted the availability of micro-operations to designate bus sources and bus destinations. A maximum of one bus source (specified by a SOURCE=... micro-operation) may be enabled within any given micro-instruction. If no bus source is specified, the X! micro-assembler command may be used in single-step mode to drive the data bus. The SOURCE=HOST micro-operation explicitly specifies this same micro-operation. The micro-assembler word SOURCE=LATCH is a dummy operation to be used to explicitly give a source for the B side of the ALU, and must be used in the same micro-instruction as a SOURCE=DHI micro-operation.

One bus destination may also be specified within a micro-operation. As a special case, DEST=DLO may be used simultaneously with any other DEST=.... micro-operations due to the way DLO is implemented in hardware. The micro-assembler words DEST=LATCH and DEST=DHI are dummy operations for documentation purposes. This means that whatever data is on the bus may be used for the ALU B side input and be written to DHI as well as any other desired bus destination.

Part Three
DEVELOPMENT SOFTWARE

Introduction

All software used to develop the WISC CPU/32 was written in MVP-FORTH. Any programming language with access to the host adapter card's 8 ports could have been used, but Forth's power and degree of control made it ideal for the task.

Interactive Development

The unique interactivity of Forth made possible the step-by-step development of the system software along with the CPU/32 hardware. Each part of the hardware was immediately tested with simple Forth functions in a single-stepping mode from the host computer. When the hardware implementation was complete, the necessary software also was essentially complete. The fully functional system could be immediately tested with this boot-strapped software.

The cross-compiler and micro-assembler are closely interrelated to each other as well as to the hardware. We have described the hardware first since understanding the hardware is essential to understanding the software beyond the high level Forth source code level of detail.

As was demonstrated in the hardware portion of this documentation, interactive testing using the single-step Slave mode of the CPU/32 is the best way to learn the system. In the case of other software, the best way to learn the use of the software is by experimenting with the functions of various words from the Forth interpreter.

The Cross Compiler

The cross-compiler is used by the host to generate a run-time, object code kernel on the CPU/32. It links primitives created by the micro-assembler with any required program structures and functions. When finished, an image of the first 32k bytes of RAM and all micro-program memory MRAM residing in the CPU/32 is saved by the host to Forth blocks on disk using the SAVE-ALL function. When performing a cold start, the LOAD-ALL function must be used to initialize the CPU/32's memory.

Sealed Vocabulary

The cross-compiler maintains a sealed vocabulary containing all the macro functions currently defined for the board. At the base of this dictionary are special cross-compiler words such as IF, ELSE, THEN, : (colon), and ; (semi-colon). After cross-compilation has started, words are added to this sealed vocabulary and are cross-compiled on the the CPU/32. Whenever the keyword CROSS-COMPILER is used, any word definitions, constants, variables, etc. will be compiled to the board.

Exercising New Commands

By entering the Forth word { (left brace,) the cross-compiler enters immediate-CPU/32-execution mode from the host. Any words subsequently typed in are searched for in the sealed vocabulary and, if found, are executed by the CPU/32. Numbers must be entered with a decimal point (32-bit integer in MVP-FORTH) to be correctly transferred to the CPU/32's data stack. As an example, from the host PC type in:

```
CROSS-COMPILER
{ 3. 6. + }     ( Displays START..END)
D.              ( Displays 9)
FORTH
```

"START.." and "END" are displayed for each word executed on the CPU/32 in this mode of operation. If execution freezes between these two displays, the execution of a cross-compiled word from the host has failed and the CPU/32 is hung. A soft reboot of the host PC is recommended.

The immediate execution mode works by building a special Forth word on the CPU/32 to execute the desired function, copying the data stack to the CPU/32, transferring control to the CPU/32 in Master mode, then stopping the CPU/32 and transferring back the resulting data stack. Entering the Forth word } (right brace) will leave the immediate execution mode and return to the cross-compiler. No colon definitions or other creation of dictionary entries should be attempted while between { and }.

Current Limitations

The current cross-compiler cannot keep track of the dictionary pointer, etc. on the CPU/32 if any changes are made to the dictionary by the CPU/32's kernel. This means that no cross-compiling or micro-assembly may be done after MVP-FORTH/32 has altered the dictionary in any way. This could be changed by modifying the cross-compiler to query key variables from the CPU/32 after each BYE command from MVP-FORTH/32.

Cross-compiled code should be kept to a minimum, because it is time-consuming and tricky to write. After a minimal kernel has been generated, the CPU/32 should do all further Forth compilation.

The Micro-Assembler

The micro-assembler is a tool to save the programmer from having to set all the bits for microcode by hand. It allows the

use of mnemonics for setting the micro-operation fields in a micro-instruction and, for the most part, automatically handles the micro-instruction addressing scheme. The micro-assembler and cross-compiler are co-resident. They share routines for accessing the CPU/32's resources and the sealed host dictionary.

Caution: writing microcode is not for the faint of heart! When using microcode, you are directly controlling signals on the wires of the printed circuit boards. If your microcode does not work as expected, you should re-read the applicable portions of Part Two of this documentation. It may, in some exceptionally tricky cases, be necessary to consult with the hardware schematics to resolve a tough question about what the hardware will and will not do. Pay particular attention to the "limitation" sections in applicable documentation. Some timing constraint violations will not show up with complete consistency. We recommend that you test each new microcoded definition with a DO LOOP that executes at least ten million times, using worst-case (and/or random) data and checking for correct and consistent results.

Defining Microcoded Definitions

Part Two of this document goes into detail about the meanings of the various micro-operations that may be combined to make up each micro-instruction. This section covers how to actually run the micro-assembler to define microcoded words on the CPU/32.

The word OPCODE: starts all microcoded definitions. The input parameter is the page number (from 0 to 511 decimal) in which the opcode resides. For example, the word + is opcode 8. This means that whenever an 8 appears in an opcode field (which would look like 040xxxxx hex in a 32-bit word,) the word + will be executed while any concurrent subroutine call, subroutine exit, or unconditional branch is being processed. The character string after OPCODE: is the name of the opcode that will be added to the cross-compiler and CPU/32 dictionaries. It is the programmer's responsibility to ensure that two opcodes are not assigned to the same MRAM page.

The word ;;END signifies the end of the definition of a microcoded primitive. Its main purpose is to return to the cross-compiler vocabulary.

The variable CURRENT-OPCODE contains the MRAM page currently assigned by OP-CODE: . It may be changed using the Forth ! (store) word to facilitate multi-page definitions in conjunction with the INC[MPC] micro-operation.

The word :: (double colon) marks the start of the definition of a micro-instruction. The number before :: must be from 0 to 7, and signifies the offset from 0 to 7 within the current MRAM page for that micro-instruction. Micro-instructions may be defined within the page in any order desired. the word ";;" (double semi-colon) is used to end each micro-instruction definition and copy the assembled bit pattern to MRAM.

When single-stepping microcode via the MIR instead of loading MRAM, the word ">>" may be used instead of "0 ::" since the offset within a page is irrelevant. Additionally, the word ";SET" must be used instead of ";;" to direct the micro-instruction to the MIR instead of MRAM. The word ";DO" may be used instead of the sequence ";SET CYCLE".

Examples

At this point, the next step towards learning the CPU/32 is to read and understand the examples provided in the source code given in Appendices E through G. The single-stepping test code in Appendix G provides examples of about every conceivable way to route data around in the CPU/32.

Most custom microcoded words that are needed in applications can be created by simply copying one of the pre-defined words to a new source screen, changing the opcode number, and modifying the definition. As an example, the word ORC! differs from +! only in that the ALU=A+B function was changed to ALU=AorB, and that the RAM word accesses were changed to RAM-BYTE accesses.

APPENDIX A

Host Data Bus Interface

WRITE PORT ADDRESSES

300 WRITE0 - DATA BUS (AUTOMATICALLY SEQUENCED)
301 WRITE1 - MIR (WRITE 4 TIMES JUST LIKE WRITE0)
302 WRITE2 - SINGLE STEP BOARD CLOCK
303 WRITE3 - START BOARD
304 WRITE4 - STOP BOARD
305 WRITE5 - SET DMA MODE
306 WRITE6 - RESET DATA BUS SEQUENCER & DMA MODE
307 WRITE7 - SERVICE REQUEST REG & INTERRUPT

READ PORT ADDRESSES
300 READ0 - DATA BUS (AUTOMATICALLY SEQUENCED)
301 READ1 - MIR (READ 4 TIMES JUST LIKE READ0)
302 READ2 - STATUS REGISTER (8 BITS)
303 READ3 -
304 READ4 -
305 READ5 -
306 READ6 -
307 READ7 -

Example Host Card Jumper Positions
("x" denotes jumper ir place)

Hex Starting Port Address

	300	308	310	318	320	328	330	338	340	348	etc.
J1	x	.	x	.	x	.	x	.	x	.	
J2	.	x	.	x	.	x	.	x	.	x	
J3	x	x	.	.	x	x	.	.	x	x	
J4	.	.	x	x	.	.	x	x	.	.	
J5	x	x	x	x	.	.	.	.	x	x	
J6	.	.	.	.	x	x	x	x	.	.	
J7	x	x	x	x	x	x	x	x	.	.	
J8	.	.	.	.	.	.	.	.	x	x	
J9	x	x	x	x	x	x	x	x	x	x	
J10	.	.	.	.	.	.	.	.	.	.	
J11	.	.	.	.	.	.	.	.	.	.	
J12	x	x	x	x	x	x	x	x	x	x	
J13	.	.	.	.	.	.	.	.	.	.	
J14	x	x	x	x	x	x	x	x	x	x	

Example Memory Card Jumper Positions
("x" denotes jumper in place)

Memory card number

	0	1	2	3	4	5	6	7	8	9	etc.
J0	.	x	.	x	.	x	.	x	.	x	
J1	x	.	x	.	x	.	x	.	x	.	
J2	.	.	x	x	.	.	x	x	.	.	
J3	x	x	.	.	x	x	.	.	x	x	
J4	.	.	.	.	x	x	x	x	.	.	
J5	x	x	x	x	.	.	.	.	x	x	
J6	.	.	.	.	.	.	.	.	x	x	
J7	x	x	x	x	x	x	x	x	.	.	

APPENDIX B

Micro-Instruction Format Summary

Microcode Bit Assignments

BOARD	#BITS	BITS	NAME	FUNCTION
INT	4	0-3	SRCx	BUS SOURCE SELECT
INT	4	4-7	DSTx	BUS DESTINATION SELECT
INT	2	8-9	DCTLx	DP COUNT CONTROL
INT	2	10-11	RCTLx	RP COUNT CONTROL
INT	2	12-13	SSELx	ALU MULTIPLEXER SHIFT CONTROL
INT	1	14	unused	-- MIR14
INT	1	15	unused	-- MIR15
DATA	4	16-19	ALUFx	ALU FUNCTION RAW INPUT
DATA	1	20	MODE	ALU MODE
DATA	1	21	CIN	ALU/SHIFT CARRY-IN
DATA	2	22-23	DSHx	DLO SHIFT/DEST CONTROL
ADDR	3	24-26	CONDx	CONDITION CODE SELECT BITS
ADDR	2	27-28	MADx	NEXT MICRO-ADDRESS CONSTANT BITS 1-2
ADDR	1	29	INMPC	INCREMENT-MPC CONTROL
ADDR	1	30	NDECO	NOT-DECODE BIT
ADDR	1	31	INAD	INCREMENT ADDRESS COUNTER BIT

Micro-Assembler Mnemonic Summary

Micro-operation field	Value	Mnemonic
SRCx	0	SOURCE=HOST (default)
	1	SOURCE=DP
	2	SOURCE=DS
	3	SOURCE=DLO
	4	SOURCE=DHI
	5	MULTIPLY
	6	DIVIDE-SELECT
	7	SOURCE=FLAGS
	8	SOURCE=RP
	9	SOURCE=RS
	10	SOURCE=ADDRESS-COUNTER
	11	SOURCE=RAM
	12	SOURCE=RAM-BYTE
	13	SOURCE=MPC
	14	SOURCE=MRAM
	15	SOURCE=FPU (unimplemented)
DSTx	0	(default) no bus destination
	1	DEST=DP
	2	DEST=DS
	3	(unused)
	4	DEST=PAGE
	5	DEST=ADDRESS-LATCH
	6	DEST=STATUS

Micro-operation field	Value	Mnemonic
	7	DEST=FLAGS
	8	DEST=RP
	9	DEST=RS
	10	DEST=ADDRESS-COUNTER
	11	DEST=RAM
	12	DEST=RAM-BYTE
	13	DEST=DECODE
	14	DEST=MRAM
	15	DEST=FPU (unimplemented)
DCTLx	0	DEC[DP]
	1	INC[DP]
	2	no-op (default)
	3	unused
RCTLx	0	no-op (default)
	1	(unused)
	2	INC[RP]
	3	DEC[RP]
SSELx	0	(pass ALU unchanged) (default)
	1	SR[ALU]
	2	SL[ALU]
	3	ROLL[ALU]
(logical ALU functions Mode=1)		
ALUFx	0	ALU=notA
	1	ALU=AnorB
	2	(unused)
	3	ALU=0
	4	ALU=AnandB
	5	ALU=notB
	6	ALU=AxorB
	7	(unused)
	8	(unused)
	9	ALU=AxnorB
	10	ALU=B
	11	ALU=AandB
	12	ALU=-1
	13	(unused)
	14	ALU=AorB
	15	ALU=A (default)
(arithmetic ALU functions Mode=0, CIN=0)		
ALUFx	0	ALU=A+1
	1	(unused)
	2	ALU=AornotB+1
	3	(unused)
	4	(unused)
	5	(unused)
	6	ALU=A-B
	7	(unused)
	8	(unused)
	9	ALU=A+B+1

Micro-operation field	Value	Mnemonic
	10	(unused)
	11	(unused)
	12	ALU=A+A+1
	13	(unused)
	14	(unused)
	15	(unused)
(arithmetic ALU functions Mode=0, CIN=1)		
ALUFx	0	ALU=A+0
	1	(unused)
	2	ALU=A-B-1
	3	(unused)
	4	(unused)
	5	(unused)
	6	ALU=A-B
	7	(unused)
	8	(unused)
	9	ALU=A+B+1
	10	(unused)
	11	(unused)
	12	ALU=A+A+1
	13	(unused)
	14	(unused)
	15	(unused)
CIN	0	CIN=0 (default for logical ALU operations)
	1	CIN=1
DSHx	0	(no-op) (default
	1	SR[DLO]
	2	SL[DLO]
	3	DEST=DLO
CONDx	0	JMP=xx0
	1	JMP=xxC
	2	JMP=xxZ
	3	JMP=xxS
	4	JMP=xxL
	5	(unused)
	6	(unused)
	7	JMP=xx1
MADx	0	JMP=00x
	1	JMP=01x
	2	JMP=10x
	3	JMP=11x
INMPC	0	(no-op) (default)
	1	INC[MPC]
NDECO	0	DECODE
	1	(no-op) (default)
INAD	0	(no-op) (default)
	1	INC[ADC]

APPENDIX C

Summary of Hardware Characteristics:

INSTRUCTION FORMAT:

Each instruction performs a simultaneous micro-coded primitive execution and a jump. The microcoded primitive starts at micro-code word offset 0 of the microcode page held in the op-code field. The address field contains the word address of the next instruction to be executed. The jump mode field controls whether the address field is a jump target, or whether a subroutine call (DOCOL) or subroutine return (EXIT) is to be performed.

In the case where the next instruction is at the next consecutive memory address, the JUMP mode is used with the address field of the instruction set appropriately. All instructions must be aligned on word boundaries

JUMP MODE FIELD:
BITS 0-1: 00 = JUMP (eliminates tail-end recursion)
(also for jump to next sequential address)
01 = Subroutine exit
10 = Subroutine call
11 = ILLEGAL

ADDRESS FIELD:
BITS 2-22: High 21 bits of the 23-bit address of jump destination. The low two bits of the address are assumed to be zero (i.e. aligned on a word boundary.)

OP CODE FIELD:
BITS 23-31: These 9 bits set the page register of the micro-program counter for the next op-code.

DECODING SEQUENCE

CYCLE	ACTION
DEC0	Clock interrupt register near falling clock edge Clock micro-program counter at end of clock cycle
DEC1	Increment address counter on rising clock edge Enable next address register outputs to RADxx IF EXIT, then clock Ram Address Latch from RS and enable RAL outputs instead of Next Addr Reg ENDIF IF CALL then decrement RP ENDIF
DEC2	(Note: DEC2 from one instruction may occur simultaneously with DEC0 of the next instruction) Enable Next Addr Reg outputs to RADxx IF EXIT, then enable RAL outputs and increment RP ENDIF IF CALL, then write address counter to RS on rising clock edge ENDIF IF no interrupt, then clock next address register, next address register, and instruction latch ENDIF Clock address counter from address bus

INTERRUPT HANDLING ACTIONS

If an interrupt occurs, then the MPC is loaded with a reference to instruction # 1 (interrupt handler) on DEC0. Also, EXIT and CALL are inhibited during this clock cycle. The interrupt handling micro-instruction is expected to save the contents of the next address register, the page register, and the instruction latch, and set the interrupt mask bit. IMPORTANT: The interrupt handling word must wait one clock cycle after setting the interrupt mask before doing its own DECODE!!!

Since the next address register is not clocked if an interrupt is in progress, the next address register, instruction latch, and page register all contain the data of the instruction that was about to be executed when the interrupt occurred.

INTERRUPT FLAGS

BIT	CONTENTS
24	DATA STACK POINTER ERROR
25	RETURN STACK POINTER ERROR
26	HOST SERVICE REQUEST (HOST WROTE TO SERVICE REQ REG)
27	DYNAMIC RAM PARITY ERROR (UNIMPLEMENTED)
28	Software interrupt #2
29	Software interrupt #1
30	Software interrupt #0
31	INTERRUPT MASK (1=ENABLE 0=MASK)

SUMMARY OF IMPORTANT MICROCODING RESTRICTIONS

1) In general, never use the RS or RP during the first or last micro-instruction of a word.

2) Never route the RS or RAM through the ALU in a single clock cycle.

3) Never route RS to or from RAM data directly.

4) Wait one clock cycle before using RAM after setting ADDRESS-LATCH from the RS.

5) Always use a DECODE on the next to last micro-instruction executed and an END on the last micro-instruction executed in each definition.

6) Never use an arithmetic ALU operation on the last micro-operation of a word.

7) Never use a JMP=xxZ micro-operation immediately following an ALU arithmetic operation.

APPENDIX D

~~DRAWINGS~~

~~WITH SIGNAL NAME DOCUMENTATION~~

P.K. 11/8/90

Signal Descriptions for HOST Board
("*" indicates inter-board connection)

NOTATION	SIGNAL DESCRIPTION
AEN	PC-Bus address enable (active low)
ALE	PC-Bus address latch enable -- used to latch PCAx
DIR	INTxx transceiver direction HI=host->CPU/32
DMA	DMA mode
INT0 *	Inter-box data bus (from host interface
..... *	.. card to CPU/32) bits 0-31
INT31 *	..
MASTR	Master mode
NCYCL	NOT-clock cycle request to CPU/32
NDMA	NOT-DMA mode
NDMC	NOT-DMA cycle clock request
NDMIR	NOT-Destination MIR signal to CPU/32
NIOR	PC-Bus NOT-I/O read signal
NIOW	PC-Bus NOT-I/O write signal
NMAST	NOT-master mode
NPCDK	PC-Bus NOT-DMA Acknowledge (DMA channel 3)
NRD0	NOT-read from I/O port offset 0
NRD1	NOT-read from I/O port offset 1
NRD2	NOT-read from I/O port offset 2
NREF	PC-Bus NOT-refresh signal (DACK0 for RAM refresh)
NRES	NOT-power on reset
NSEL	NOT-board select
NSEL0	NOT-Select INTxx data bits (0-7) to/from PC
NSEL1	NOT-Select INTxx data bits (8-15) to/from PC
NSEL2	NOT-Select INTxx data bits (16-23) to/from PC
NSEL3	NOT-Select INTxx data bits (24-31) to/from PC

NSMIR	NOT-Source MIR signal to CPU/32
NWR0	NOT-write to I/O port offset 0-7
.....	..
NWR7	..
NXENB	NOT-enable transceiver to PC data bus
NXRD0	NOT-intermediate read 0 (conditioned to form NRD0)
NXWR0	NOT-intermediate write 0 (conditioned to form NWR0)
PC0	Internally buffered 8-bit pc data bus
.....	..
PC7	..
PCA0	PC-Bus address bits -- used for I/O port selection
.....	.. (Direct from PC-bus)
PCA9	..
PCAD0	PC address for read/write port offset decoding
.....	..
PCAD2	..
PCD0	PC-Bus data bits
.....	..
PCD7	..
PCDRQ	PC-Bus DMA Request from board (DMA channel 3)
PCIOR	I/O Read in progress
PCIOW	I/O Write in progress
PCRES	PC-Bus reset signal (active high)
PCTC	PC-Bus terminal count
PULLX	4.7K Ohm pull-up resistor
XNCYC *	Inter-box signal for NCYCL
XDIR *	Inter-box signal for DIR
XNDMR *	Inter-box signal for NDMIR
XNDSV *	Inter-box signal for NDSRV
XNMAS *	Inter-box signal for NMAST
XNREF *	Inter-box signal for NREF
XNSMR *	Inter-box signal for NSMIR
XNSST *	Inter-box signal for NSSTA

Signal Descriptions for INT Board
("*" indicates inter-board connection)

NOTATION	SIGNAL DESCRIPTION
ALU31 *	ALU31 highest bit for sign conditional branching From ALU board
BUS0 *	System data bus (32 bits)
...	
BUS31 *	...

```
CLOCK *   SYSTEM CLOCK

DCTL0     Data stack pointer control bit 0 from MIR
DCTL1     Data stack pointer control bit 1 from MIR

DLOLO *   Lowest bit from DLO register from ALU board

DP0       Data stack pointer
.....     ..
DP11      ..

DST0      Bus destination control select from MIR
.....     ..
DST3

DVOSC     Divided crystal frequency

FASTC *   System clock that is a few nano-seconds ahead of CLOCK
          for use with RAM writes and latch clocking

INT0  *   Inter-box data bus (from host interface
..... *   .. card to CPU/32) bits 0-31
INT31 *   ..

MAD0  *   MICROADDR0     MICROCODE RAM ADDRESS
..... *   ...
MAD11 *   MICROADDR11

MASTR     Master mode

MIRCK *   MIR register clock signal

MRXDR *   Micro-instruction to bus xceiver direction

NACOT *   NOT-ALU caryy out from ALU board

NALU0 *   NOT-ALU=0 from DATA to ADDR boards

NBYTE *   NOT-byte access control

NCKFP *   NOT-cycle FPU clock

NCSMR *   NOT-chip select for MRAM

NCYCL     NOT-clock cycle request from host

NDADC *   NOT-DEST ADDRESS-COUNTER
NDDEC *   NOT-DEST decode
NDDP      NOT-DEST DP
NDDS      NOT-DEST DS
NDFLG *   NOT-DEST INTERUPT FLAGS
NDFPU *   NOT-bus destination FPU & cycle FPU clock

NDIV  *   NOT DIVIDE-SELECT

NDMA  *   NOT-DMA transfer mode

NDMIR     NOT-Destination MIR signal from host to CPU/32
```

```
NDMRA      NOT-DEST MRAM
NDPAG *    NOT-DEST RAM ADDRESS PAGE REGISTER

NDPER *    NOT-Data stack pointer overflow/underflow

NDRAL *    NOT-DEST RAM-ADDRESS-LATCH
NDRB  *    NOT-DEST RAM-BYTE
NDRP       NOT-DEST RP
NDRS  *    NOT-DEST RS
NDRW  *    NOT-DEST RAM-WORD

NDSRV      NOT-Destination service request register
           (loading signal from host)

NDSTA      NOT-DEST STATUS REGISTER

NDSWE      NOT-DS RAM chip write enable

NDSXE      NOT-DS to bus transceiver enable

NHOST *    NOT-host interrupt

NINTR      NOT-interrupt to PC HOST when dest=host request

NMAST      NOT-master mode

NMRXE *    NOT-micro-instruction to data bus xceiver enable

NMULT *    NOT MULTIPLY-SELECT

NPCEN *    NOT-enable PC to bus xceiver from host

NREF  *    NOT-refresh command from host for DRAMs

NRPEN *    NOT-Return stack count up/down enable
NRPER *    NOT-Return stack pointer over/underflow

NRSWE *    NOT-RS write enable for stack RAM chips
NRSCS *    NOT-RS chip select for return stack RAM chips
NRSXE *    NOT-RS to bus xceiver enable

NSADC *    NOT-SOURCE ADDRESS-COUNTER
NSDHI *    NOT-SOURCE DHI & HOLD B-SIDE ALU LATCH
NSDLO *    NOT-SOURCE DLO
NSDP       NOT-SOURCE DP
NSDS       NOT-SOURCE DS

NSFLG *    NOT-SOURCE SERVICE REQUEST (0-7)
                      /INTERRUPT FLAGS(24-31)
NSFPU *    NOT-bus source FPU
NSMIR *    NOT-bus source MIR
NSMPC *    NOT-SOURCE MPC
NSMRA      NOT-SOURCE MRAM
NSPC       NOT-SOURCE PC -- host data buffer to bus
NSRB  *    NOT-SOURCE RAM-BYTE
NSRP  *    NOT-SOURCE RP
NSRS  *    NOT-SOURCE RS
NSRW  *    NOT-SOURCE RAM-WORD

NSSTA      NOT-Source Status register from CPU/32 to host
```

```
NWAIT *   NOT-Open collector clock cycle wait request
                for use by slow memories

NWMRA *   NOT-write MRAM conditioned
            write-enable signal to MRAM memory CHIPS

OSC   *   Clock frequency without cycling/waiting inhibits

PULLG     4.7K Ohm pullup resistor

RALCK *   RAM address latch clock
          -- conditioned for DMA transfers.

RCTL0 *   RS control bit from MIR
RCTL1 *   RS control bit from MIR

RP0       RETURN STACK POINTER BITS 0-11
.....     ..  Up to 4K bytes of addressability
RP11      ..

RPUP  *   Return pointer up/not-down control bit

RS0   *   Return stack data bus
..... *   ..
RS31  *   ..

RSXC  *   RS bus transceiver direction control

SPL0  *   GND
SPL1  *   +5v

SRC0      Bus source control from MIR
.....     ..
SRC3      ..


SSEL0 *   ALU shift selection control from MIR to ALU board
SSEL1 *   ...

XCLK  *   system clock bus version (inverted)

XNCYC *   Inter-box signal for NCYCL
XNDMA *   Inter-box signal for NDMA
XNDMR *   Inter-box signal for NDMIR
XNDSV *   Inter-box signal for NDSRV
XNINT *   Inter-box signal for NINTR
XNMAS *   Inter-box signal for NMAST
XNPCE *   Inter-box signal for NPCEN
XNREF *   Inter-box signal for NREF
XNSMR *   Inter-box signal for NSMIR
XNSST *   Inter-box signal for NSSTA
```

Signal Descriptions for DATA Board
("*" indicates inter-board connection)

NOTATION	SIGNAL DESCRIPTION
ALU0	ALU outputs (from ALU into multiplexor)
.....	...
ALU30	...
ALU31 *	...
ALUF0	Raw ALU function inputs from MIR
.....	.. (inverted & conditioned to form FUNCx)
ALUF3	
ALUX0	X carry-lookahead signal from ALU(0 - 3)
ALUX1	X carry-lookahead signal from ALU(4 - 7)
ALUX2	X carry-lookahead signal from ALU(8 -11)
ALUX3	X carry-lookahead signal from ALU(12-15)
ALUX4	X carry-lookahead signal from ALU(16-19)
ALUX5	X carry-lookahead signal from ALU(20-23)
ALUX6	X carry-lookahead signal from ALU(24-27)
ALUX7	X carry-lookahead signal from ALU(28-31)
ALUY0	Y carry-lookahead signal from ALU(0 - 3)
ALUY1	Y carry-lookahead signal from ALU(4 - 7)
ALUY2	Y carry-lookahead signal from ALU(8 -11)
ALUY3	Y carry-lookahead signal from ALU(12-15)
ALUY4	Y carry-lookahead signal from ALU(16-19)
ALUY5	Y carry-lookahead signal from ALU(20-23)
ALUY6	Y carry-lookahead signal from ALU(24-27)
ALUY7	Y carry-lookahead signal from ALU(28-31)
BUS0 *	System data bus (32 bits)
...	...
BUS31 *	...
CIN	ALU Carry-in bit for ALU from MIR
CKDHI	Clock DHI register
CLOCK *	system clock
DLOHI	High shift out bit of DLO register
DLOLO *	Low shift out bit of DLO register
DSH0	DLO Shift control bit 0 from MIR
DSH1	DLO Shift control bit 1 from MIR
FASTC *	FASTCLOCK
FUNC0	ALU 74181 function & mode inputs
.....	... (conditioned for division as required)
FUNC3	...
LATEN	Enable transparent latch on B-side ALU input (pass bus data through to ALU)
MAD0 *	MICROADDR0 MICROCODE RAM ADDRESS
.....	...
MAD11 *	MICROADDR11

MIRCK * MIR clock

MODE ALU mode select from MIR

MRXDR * Micro-instruction to data bus xceiver direction

NALU0 * NOT-ALU = zero for condition code

NACOT * NOT-ALU carry out for condition code

NCIN0 NOT-ALU Carry-in bit for ALU (0-3)
NCIN1 NOT-ALU Carry-in bit for ALU (4-7)
NCIN2 NOT-ALU Carry-in bit for ALU (8-11)
NCIN3 NOT-ALU Carry-in bit for ALU (12-15)
NCIN4 NOT-ALU Carry-in bit for ALU (16-19)
NCIN5 NOT-ALU Carry-in bit for ALU (20-23)
NCIN6 NOT-ALU Carry-in bit for ALU (24-27)
NCIN7 NOT-ALU Carry-in bit for ALU (28-31)

NCSMR * NOT-chip select for MRAM

NDIV * NOT DIVIDE-SELECT

NMRXE * NOT-micro-instruction to data bus xceiver enable

NMULT * NOT MULTIPLY-SELECT

NSDHI * NOT-SOURCE DHI & HOLD B-SIDE ALU LATCH
NSDLO * NOT-SOURCE DLO
NSMIR * NOT-bus source MIR

NWMRA * NOT-write MRAM conditioned
write-enable signal to MRAM memory CHIPS

PULLA 4.7 K-ohm Pull-up resistor
PULLB 4.7 K-ohm Pull-up resistor

SLIN DLO shift left input bit

SPL0 * GND
SPL1 * +5v

SRIN Multiplexor shift right input for ALU multiplexor.

SSEL0 * Multiplexor selection bit 0 from MIR
SSEL1 * Multiplexor selection bit 1 from MIR

XCLK * system clock bus version (inverted)

Signal Descriptions for ADDR Board
("*" indicates inter-board connection)

NOTATION	SIGNAL DESCRIPTION
ACIN1	Address counter carry-in for bits (10-17)
ACIN2	Address counter carry-in for bits (18-22)
ADCEN	Address counter count enable from MIR
ALU31 *	Sign bit from ALU output
BUS0 *	System data bus (32 bits)
...	
BUS31 *	...
BYTE	Signifies that byte RAM accessing is being used
CALL	Subroutine Call being processed during decode
CKLRD	Clock low 2 bits of RAM data to form CALL/EXIT bits
CLOCK *	System clock
CLOKB *	System clock -- Second copy for fanout control
COND0	Condition code multiplexor control bits
.....	.. from micro-instruction
COND2	..
DC12	Decode cycle 1 or 2 -- debounced DEC1 OR DEC2
DEC0	Decode cycle 0 -- "DECODE" microcycle
DEC1	Decode cycle 1 -- "END" microcycle
DEC2	Decode cycle 2 -- 1st cycle of next opcode
DLOLO *	Low shift out bit of DLO register
EXIT	Subroutine exit being processed during decode
FASTC *	FASTCLOCK
INAD	Address counter increment control bit from MIR
INMPC	Increment MPC microcode control bit from MIR
INTR	Interrupt service required (used during Decode 0)
MAD0 *	Condition code select next-microaddress bit
MAD1 *	Next micro-address bit 1 from MIR
MAD2 *	Next micro-address bit 2 from MIR
MAD3 *	Next micro-address
..... *	
MAD11 *	..
MIRCK *	MIR clock
MPD3	Micro-program counter data input bus
.....	..
MPD11	..

MRXDR * Micro-instruction to data bus xceiver direction

NACOT * NOT-ALU carry out condition code bit

NADOE NOT-address counter output enable

NALU0 * NOT-ALU equal to zero condition code bit

NBYTE NOT-Signifies byte RAM accessing is being used

NCAL2 NOT CALL activated only during DEC2 cycle
NCALL NOT-currently processing a CALL command during decode (subroutine return address stored to return stack.)

NCKFP * NOT-cycle FPU clock

NCSMR * NOT-chip select for MRAM

ND2CK NOT-decode 2 clock for IL and NAR

NDADC * NOT-DEST ADDRESS-COUNTER
NDADD NOT-DEST address latch (activated either for direct write to address latch or for pass-thru to address counter).
NDDEC * NOT-DEST DECODE (simulates decoding of instruction and loads next address register & instruction latch.)

NDEC0 NOT-zeroth cycle of decode from MIR
NDEC1 NOT-1st cycle of decode
NDEC2 NOT-2nd cycle of decode (may correspond with the next NDEC0).

NDFLG * NOT-DEST INTERUPT FLAGS
NDFPU * NOT-bus destination FPU & cycle FPU clock
NDMA * NOT-DMA transfer mode
NDPAG * NOT-dest page register

NDPER * NOT-Data stack pointer over/underflow

NDRAL * NOT-DEST RAM-ADDRESS-LATCH
NDRB * NOT-DEST RAM-BYTE
NDRS * NOT-DEST RS
NDRW * NOT-DEST RAM-WORD

NEX1 NOT-EXIT activated during DEC1 cycle
NEXIT NOT-currently processing an EXIT command during decode (subroutine return address fetched from return stack)

NHOST * NOT-host service request interrupt

NINTR NOT-interrupt pending (for Decode 0)

NLDAD NOT-load address counter

NMRXE * NOT-micro-instruction to data bus xceiver enable

NPRTY * NOT-RAM parity error (open collector)

NRAM * NOT-activate ram data xceivers to ram data bus

```
NRLOE      NOT-RAM address latch output enable

NRPER *    NOT-Return stack pointer over/underflow
NRPEN *    NOT-Return stack count up/down enable

NRSCS *    NOT-RS chip select for return stack RAM chips
NRSWE *    NOT-RS write enable for stack RAM chips
NRSXE *    NOT-RS to bus xceiver enable

NRXEH      NOT-RAM to data bus xceiver enable for bits (8-31)

NSADC *    NOT-SOURCE ADDRESS-COUNTER
NSFLG *    NOT-SOURCE SERVICE REQUEST (0-7)
                    /INTERRUPT FLAGS(24-31)
NSFPU *    NOT-bus source FPU
NSMIR *    NOT-bus source MIR
NSMPC *    NOT-SOURCE MPC
NSRB  *    NOT-SOURCE RAM-BYTE
NSRS  *    NOT-SOURCE RS
NSRW  *    NOT-SOURCE RAM-WORD

NWMRA *    NOT-write MRAM conditioned
             write-enable signal to MRAM memory CHIPS

OSC   *    (undivided crystal frequency)

PULLD      4.7K pull-up resistor
PULLE      4.7K pull-up resistor
PULLF      4.7K pull-up resistor
PULLG      4.7K pull-up resistor

RAD0  *    RAM address bus fed to RAM boards
.....  *   ...
RAD31 *    ...

RALCK *    RAM address latch clock -- conditioned for DMA
           addressing and latch loading/flow-through.  Note: latch
           becomes transparent when this signal is high.

RCTL0 *    RS control bit from MIR
RCTL1 *    RS control bit from MIR

RD0   *    Ram data bus to/from ram boards &
.....      ...  to next address register
RD31  *    ...

RPUP  *    Return pointer up/not-down control bit

RS0   *    Return stack data bus
.....  *   ..
RS31  *    ..

RSXC  *    RS bus transceiver direction control

SPL0  *    GND
SPL1  *    +5v

XCLK  *    system clock bus version (inverted)
```

Signal Descriptions for MEM Board
("*" indicates inter-board connection)

NOTATION		SIGNAL DESCRIPTION
AD0		Internal RAM address bus on this board
.....		.. (Buffered from other boards by transmitters)
AD14		..
CLOCK	*	system clock
D0		Internal RAM data bus on this board
.....		.. (isolated from other boards by xceivers)
D31		..
NBDSL		NOT-board select based on address jumpers
NDRB	*	NOT-DEST RAM-BYTE
NDRW	*	NOT-DEST RAM-WORD
NENB0		NOT-enable RAM bank 0
NENB1		NOT-enable RAM bank 1
NENB2		NOT-enable RAM bank 2
NENB3		NOT-enable RAM bank 3
NOE		NOT-output enable for RAM chips
NRAM	*	NOT-RAM data xceiver enable for all RAM boards
NWE0		NOT-write enable bits (0-7) of active RAM bank
NWE1		NOT-write enable bits (8-15) of active RAM bank
NWE2		NOT-write enable bits (16-23) of active RAM bank
NWE3		NOT-write enable bits (24-31) of active RAM bank
RAD0	*	RAM address bus from ADDR board
.....	*	..
RAD30	*	..
RD0	*	Ram data bus to/from ram boards &
.....	*	... to next address register
RD31	*	...
SPL0	*	GND
SPL1	*	+5v
XCLK	*	system clock bus version (inverted)

What we claim is:

1. A writable instruction set computer comprising:

data bus means for transferring data having a predetermined number of bits;

addressable and writable main program memory means coupled to said data bus means for storing macrocode, including instructions having the predetermined number of bits, and for storing data from and loading stored data onto said data bus means;

memory address logic means coupled to said data bus means and said main program memory means for addressing said main program memory means;

addressble and writable micro-program memory means coupled to said main program memory means for storing microcode instructions addressed by the macrocode instructions;

arithmetic logic unit (ALU) means coupled to said data bus means for performign operations on data received from said data bus means as defined by the microcode stored in said micro-program memory means;

data stack memory means coupled to said data bus means for storing data received from said data bus means for use during program execution;

return stack memory means physically separate from said main memory means, and coupled to said data bus means and to said memory address logic means for storing subroutine return address used during program execution, said memory address logic means addressing said main program memory means with the subroutine return address stored in said return stack memory means while said ALU means performs operations on data transferred from said data stack memory means on said data bus means;

clock means for generating a cyclic clock signal; and

execution control logic means coupled to said microprogram memory means, ALU means, data stack memory means, return stack memory means, data bus means, and clock means for executing the microcode instructions, including performing only one data transfer on said data bus means for each clock signal cycle;

said data bus means providing only one communication path for transferring bidirectionally data between said ALU means, said data stack memory means and said main program memory means.

2. A computer according to claim 1 wherein said main program memory means stores each instruction as the combination of an opcode and a main program memory address.

3. A computer according to claim 2 wherein said address included in said instruction comprises the address of the location of the succeeding instruction in said main program memory.

4. A computer according to claim 3 wherein said execution control logic means is further for executing the operation specified by the opcode of a current macrocode instruction while, simultaneously with the operation executing, said memory address logic means fetches the macrocode instruction corresponding to the address included in the current macrocode instruction.

5. A computer according to claim 4 wherein said main program memory means further stores for a machine language program instruction, an indicator indicating whether the succeeding operation is a subroutine return, and said memory address logic means is further responsive to address information received from said stack memory means for executing a subroutine return simultaneously with the executing of the current operation, when the indicator indicates that the next operation is a subroutine return.

6. A computer according to claim 5 wherein said main program memory means stores a condition code having one of a plurality of values including a predetermined value, and a macrocode instruction comprises a conditional branch opcode requiring execution of a subroutine call if the value of the condition code is the predetermined value, and a subroutine call address, said memory address logic means further executing the subroutine call while said execution control logic means executes the conditional branch opcode, said memory address logic means being responsive to said execution control logic means for aborting the execution of the subroutine call if the value of the condition code is not the predetermined value.

7. A computer according to claim 1 wherein said ALU means comprises first and second input ALU ports and an output ALU port, said computer further comprising transparent latch means having an input latch port coupled to said data bus means and an output latch port coupled to said first input ALU port, said latch means being controllable for either transferring data input on said input latch port to said output latch port or retaining data input on said input latch port without it appearing on said output latch poret, and data register means having a register input port coupled to to said output ALU port and a register output port coupled to said second input ALU port and to said data bus means, said transparent latch means being for storing temporarily data received from said data bus means while data stored in said data register means is output to said data bus means.

8. A computer according to claim 1 wherein each macrocode instruction includes an opcode, and further comprising:

data stack pointer means coupled to said data bus means and said data stack memory means for only storing one pointer pointing to an element in said data stack memory mean,s wherein said execution control logic means is further for setting the pointer to point to any element in said data stack memory means without altering the contents of said stack memory means, the one pointer being the only means for accessing an element in said data stack memory means; and

interrupt means coupled to said execution control logic means, and responsive to interrupt signals for generating an interrupt opcode when an interrupt signal indicates that the program execution is to be interrupted;

said execution control logic means being responsive to the interrupt opcode for itnerrupting program execution by isnerting the interrupt opcode in place of the next macrocode opcode, and thereby interrupting the program execution only when a next macrocode opcode is to be executed by said execution control logic means, said execution control logic means further controlling execution of the macrocode such that the pointer stored in said data stack pointer means is set to point to a predetermined data stack element prior to executing each new macrocode opcode, whereby the pointer can be changed to point to different data stack elements during execution of a macrocode opcode

without altering the contents of said data stack memory means.

9. A writable instruction set computer comprising:

bus means;

addressable and writable main program memory means coupled to said bus means for storing macrocode including opcodes, and data, and for loading stored data onto said bus means;

memory address logic means coupled to said bus means and said main program memory means for addressing said main program memory means;

addressable and writable micro-program memory means coupled to said main program memory means for storing microcode addressed by the macrocode opcodes;

arithmetic logic unit (ALU) means coupled to said bus means for performing operations on data from said bus means as defined by microcode stored in said micro-program memory means;

data stack memory means coupled to said bus means for storing data used during opcode execution;

execution control logic means coupled to said main program memory means, said bus means and said micro-program memory means, and responsive to instructions received from said main program memory means for executing the macrocode;

data stack pointer means coupled to said bus means and said data stack memory means for only storing one pointer pointing to an element in said data stack memory means, wherein said execution control logic means is further for setting the pointer to point to any element in said data stack memory means without altering the contents of said data stack memory means, the one poitner being the only means for accessing an element in said data stack memory means; and

interrupt means coupled to said execution control logic means, and responsive to interrupt signals for generating an interrupt opcode when an interrupt signal indicates that the program execution is to be interrupted;

said execution control logic means bieng responsive to the itnerrupt opcode for interrupting program execution by inserting the interrupt opcode in place of the next macrocode opcode, and thereby interrupting the program execution only when a next macrocode opcode is to be executed by said execution control logic means, said execution control logic means further controlling execution of the macrocode such that the pointer stored in said data stack pointer means is set to point to a predetermined data stack element prior to executing each new macrocode opcode, whereby the pointer can be changed to point to different data stack elements during execution of a macrocode opcode without altering the contents of said data stack memory means.

* * * * *

United States Patent [19]
Koopman, Jr. et al.

[11] Patent Number: 5,053,952
[45] Date of Patent: Oct. 1, 1991

[54] STACK-MEMORY-BASED WRITABLE INSTRUCTION SET COMPUTER HAVING A SINGLE DATA BUS

[75] Inventors: **Philip J. Koopman, Jr.**, N. Kingston, R.I.; **Glen B. Haydon**, La Honda, Calif.

[73] Assignee: **WISC Technologies, Inc.**, La Honda, Calif.

[21] Appl. No.: **58,737**

[22] Filed: **Jun. 5, 1987**

[51] Int. Cl.[5] **G06F 9/42**; G06F 9/22; G06F 13/40

[52] U.S. Cl. **364/200**; 364/244.3; 364/262.7; 364/240; 364/247.7; 364/240.1

[58] Field of Search ... 364/200 MS File, 900 MS File

[56] **References Cited**

U.S. PATENT DOCUMENTS

3,215,987	11/1965	Terzian	364/200
3,629,857	12/1971	Faber .	
3,757,306	9/1978	Boone .	
3,771,141	11/1973	Culler	364/200
3,786,432	1/1974	Woods .	
4,045,781	8/1977	Levy et al.	364/200
4,204,252	5/1980	Hitz et al.	364/200
4,210,960	7/1980	Borgerson et al.	364/200
4,415,969	11/1983	Bayliss et al.	364/200
4,447,875	5/1984	Bolton et al.	364/200
4,491,912	1/1985	Kainaga et al.	364/200
4,546,431	10/1985	Horvath	364/200
4,610,003	9/1986	Logsdon et al.	364/200
4,618,925	10/1986	Bratt et al.	364/200
4,654,780	3/1987	Logsdon et al.	364/200
4,674,032	6/1987	Michaelson .	
4,719,565	1/1988	Moller .	
4,791,551	12/1988	Garde	364/200
4,835,738	5/1989	Niehaus et al. .	

OTHER PUBLICATIONS

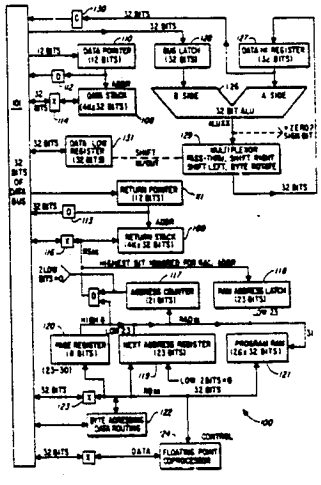
"Stack-Oriented WISC Machine", WISC Technologies, La Honda, Ca., 94020, 2 pages.
BYTE 6/86, Microcoded IBM PC Board, Mtn. Vw. Press Advertisement, Haydon, MVP Microcoded CPU/16, Mountain View Press, 4 pages.
Koopman & Haydon, MVP Microcoded CPU/16 Architecture, Mountain View Press, 4 pages.
Koopman, Microcoded Versus Hard-Wired Control, BYTE, Jan. 1987, pp. 235-242.
Haydon, The Multi-Dimensions of Forth, *Forth Dimensions*, vol. 8, No. 3, pp. 32-34, Sep./Oct., 1986.
Rust, ACTION Processor Forth Right, *Rochester Forth Standards Conference*, pp. 309-315, 3/8/79.
Wada, Software and System Evaluation of a Forth Machine System, *Systems, Computers, Controls*, vol. 13, No. 2, pp. 19-28.
Wada, System Design and hardware Structure of a Forth Machine System, *Systems, Computers, Controls*, vol. 13, No. 2, 1982, pp. 11-18.
(List continued on next page.)

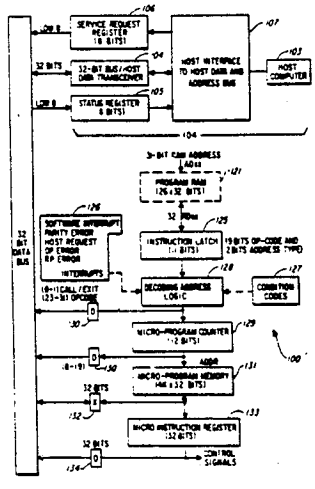
Primary Examiner—Gareth D. Shaw
Assistant Examiner—P. V. Kulik
Attorney, Agent, or Firm—Edward B. Anderson

[57] **ABSTRACT**

A computer is provided as an add-on processor for attachment to a host computer. Included are a single data bus, a 32-bit arithmetic logic unit, a data stack, a return stack, a main program memory, data registers, program memory addressing logic, micro-program memory, and a micro-instruction register. Each machine instruction contains an opcode as well as a next address field and subroutine call/return or unconditional branching information. The return address stack, memory addressing logic, program memory, and microcoded control logic are separated from the data bus to provide simultaneous data operations with program control flow processing and instruction fetching and decoding. Subroutine calls, subroutine returns, and unconditional branches are processed with a zero execution time cost. Program memory may be written as either bytes or full words without read/modify/write operations. The top of data stack ALU register may be exchanged with other registers in two clock cycles instead of the normal three cycles. MVP-FORTH is used for programming a microcode assembler, a cross-compiler, a set of diagnostic programs, and microcode.

9 Claims, 90 Drawing Sheets





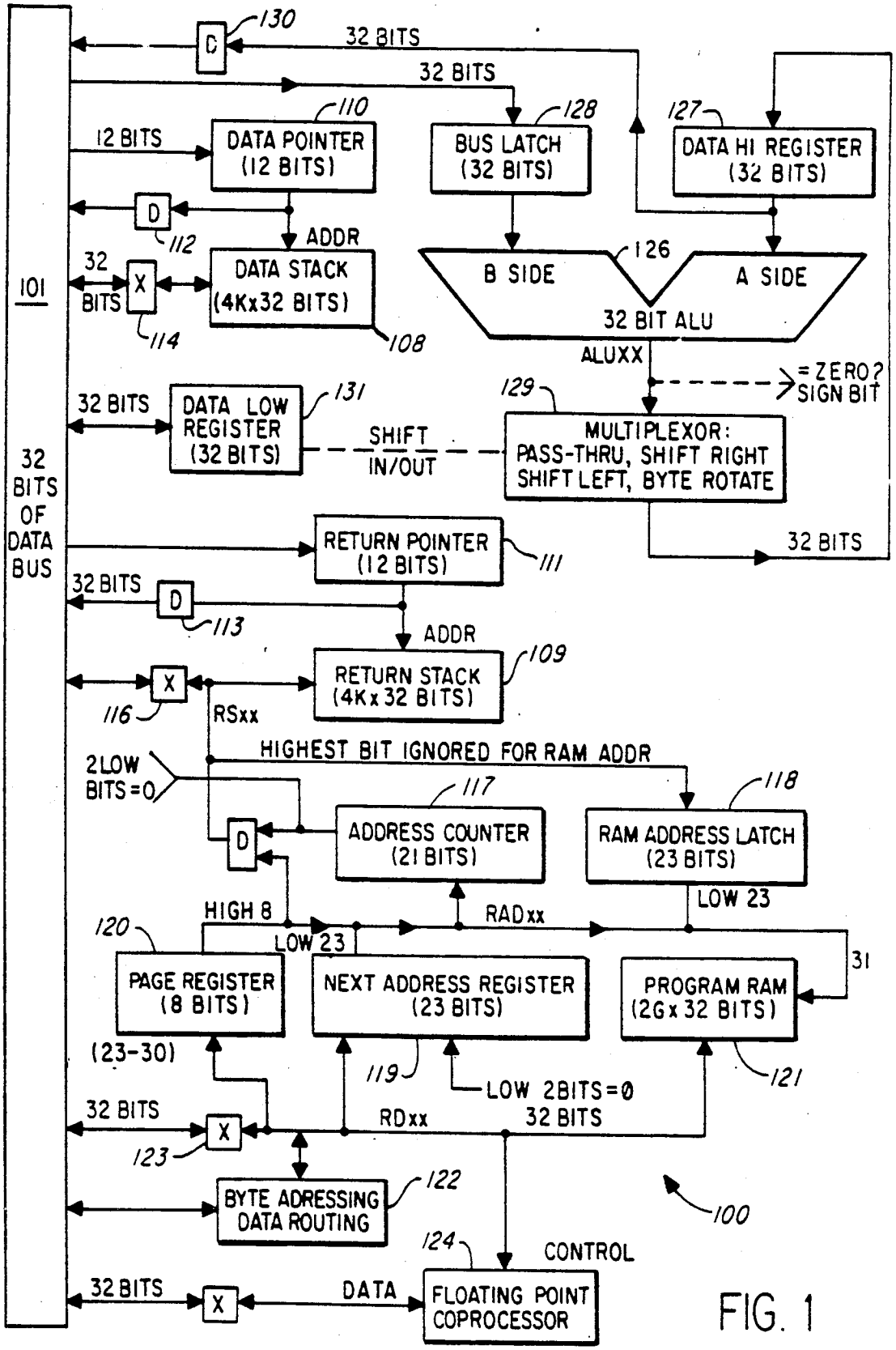

130
C
32 BITS
32 BITS
110
128
127
12 BITS
DATA POINTER (12 BITS)
BUS LATCH (32 BITS)
DATA HI REGISTER (32 BITS)
D
112
ADDR
32 BITS
X
114
DATA STACK (4Kx32 BITS)
108
101
B SIDE
126
A SIDE
32 BIT ALU
ALUXX
= ZERO? SIGN BIT
32 BITS
DATA LOW REGISTER (32 BITS)
131
129
SHIFT IN/OUT
MULTIPLEXOR: PASS-THRU, SHIFT RIGHT SHIFT LEFT, BYTE ROTATE
32 BITS OF DATA BUS
RETURN POINTER (12 BITS)
111
32 BITS
D
113
ADDR
X
116
RETURN STACK (4Kx32 BITS)
109
RSxx
HIGHEST BIT IGNORED FOR RAM ADDR
2LOW BITS=0
117
118
D
ADDRESS COUNTER (21 BITS)
RAM ADDRESS LATCH (23 BITS)
HIGH 8
LOW 23
RADxx
LOW 23
120
31
PAGE REGISTER (8 BITS)
NEXT ADDRESS REGISTER (23 BITS)
PROGRAM RAM (2Gx32 BITS)
(23-30)
119
LOW 2BITS=0
121
32 BITS
X
123
RDxx
32 BITS
BYTE ADRESSING DATA ROUTING
122
100
124
CONTROL
32 BITS
X
DATA
FLOATING POINT COPROCESSOR

FIG. 1

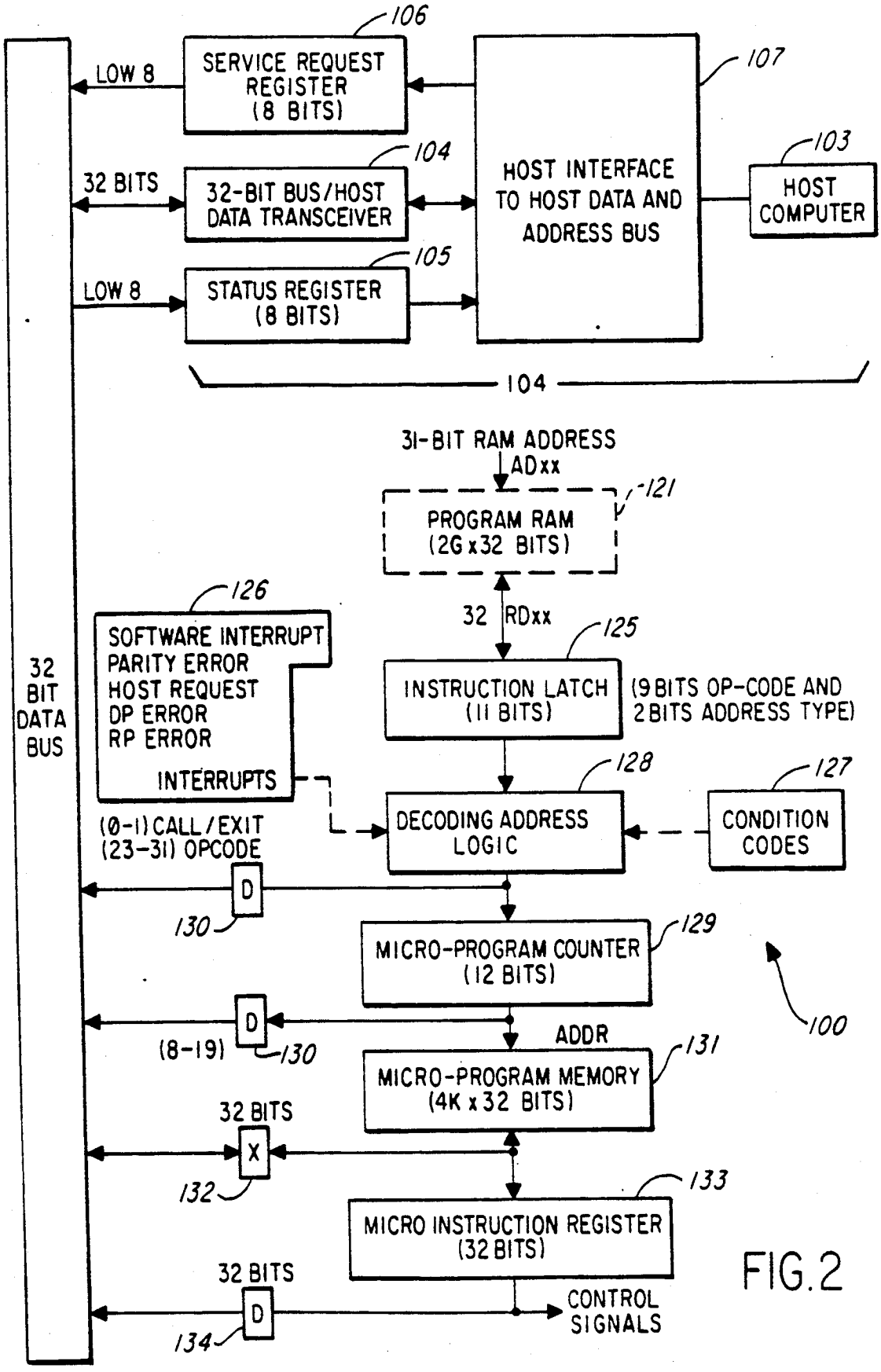

106
SERVICE REQUEST REGISTER (8 BITS)
LOW 8
107
HOST INTERFACE TO HOST DATA AND ADDRESS BUS
103
HOST COMPUTER
104
32 BITS
32-BIT BUS/HOST DATA TRANSCEIVER
105
LOW 8
STATUS REGISTER (8 BITS)
104
31-BIT RAM ADDRESS
ADxx
121
PROGRAM RAM (2G x 32 BITS)
32 RDxx
126
SOFTWARE INTERRUPT
PARITY ERROR
HOST REQUEST
DP ERROR
RP ERROR
INTERRUPTS
125
INSTRUCTION LATCH (11 BITS)
(9 BITS OP-CODE AND 2 BITS ADDRESS TYPE)
32 BIT DATA BUS
128
DECODING ADDRESS LOGIC
127
CONDITION CODES
(0-1) CALL/EXIT
(23-31) OPCODE
D
130
129
MICRO-PROGRAM COUNTER (12 BITS)
100
D
(8-19)
130
ADDR
131
MICRO-PROGRAM MEMORY (4K x 32 BITS)
32 BITS
X
132
133
MICRO INSTRUCTION REGISTER (32 BITS)
32 BITS
D
134
CONTROL SIGNALS

FIG. 2

is a transparent latch (implemented in the preferred embodiment with standard 74ALS373 integrated circuits) that can either pass data through from the data bus, or retain data present on the bus on the previous clock cycle. This retention capability, along with the capability to direct the contents of the ALU register directly to the bus, allows exchanging the data hi register with the data stack or other registers in two clock cycles instead of the three clock cycles which would be required without this innovation. Since exchanging the top two elements of the data stack is a common operation, this results in a substantial increase in processing speed with very little hardware cost over having multiple intermediate storage registers.

In the preferred embodiment of the invention, a four-way decoder is used to control individual 8-bit banks of the 32-bit program memory. This, combined with data flow logic in the interface between the program memory and the data bus, allows individual access to modification of any byte value in program memory with a single write operation. Conventional computers require a full width memory read, 8-bit modification of the data within a temporary holding register, and a full width memory write operation to update a byte in memory, resulting in substantially slower speeds for such operations. While the preferred embodiment employs this new technique to modify 8 bits of a 32 bit word, this technique is generally applicable to accessing any subset of bits within any length of memory word.

The combination of appropriate software shown in Appendix A that exploits the simultaneous processing of conditional branching opcodes with subroutine calls and the use of hardware stacks combine to form an exceptionally efficient expert system inference engine. An expert system rule base typically is formed by a nested list of "rules" which can invoke other rules via subroutine calls that are only activated under certain conditions. The capability of the preferred embodiment to simultaneously process each rule-oriented subroutine call while evaluating the conditions under which the subroutine call will either be allowed to proceed or will be aborted greatly speeds up processing of expert system programs. Expert systems can run at speeds of over 600,000 inferences per second on the preferred embodiment using a 150ns clock cycle, which is a substantial improvement over existing general purpose computers, and in fact over most special purpose computers.

It will be seen that such a computer offers substantial optimization of throughput while maintaining flexibility. It is also predicted that use of such a machine will positively influence programs and programming languages to have improved structure and lower development cost by not penalizing the modern software principle of breaking programs up into small subroutines.

These and other advantages and features of the invention will be more clearly understood from a consideration of the drawings and the following detailed description of the preferred embodiment.

BRIEF DESCRIPTION OF THE DRAWINGS

Referring to the associated sheets of drawings:

FIGS. **1** and **2** are a system block diagram showing a preferred embodiment made according to the present invention;

FIGS. **3** through **89** show the detailed schematics of the embodiment of FIGS. **1** and **2** organized into groups of components placed on five separate printed circuit boards in the preferred embodiment, and;

FIGS. **90** through **95** show a preferred placement of the integrated circuits for FIGS. **3** through **89** on **5** expansion boards for use in conjunction with a host computer.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT SYSTEM HARDWARE

Referring initially to FIG. **1** and FIG. **2**, a system overview of the hardware of a writable instruction set computer **100** made according to the present invention is shown. Computer **100** includes a single 32-bit system data bus **101**. An interface assembly **102** is coupled to bus **101** for interfacing with a host computer **103**, which for the preferred embodiment is an IBM PC/AT, made by International Business Machines, Inc., or equivalent personal computer. Assembly **102** includes a bus interface transceiver **104**, an 8-bit status register **105** for requesting host services, and an 8 bit service request register **106** for the host to request services of computer **100**. In the preferred embodiment, the host interface adapter **107** provides the necessary 8 bit host to 32 bit computer data sizing changes. Hosts in other embodiments would not necessarily be restricted to an 8-bit interface.

Memory stack means are provided in the form of a data stack **108** and a return address stack **109**. Each stack is organized in the preferred embodiment as 4 kilowords of 32 bits per word. Each stack has an associated pointer. Specifically, a data stack pointer **110** is associated with data stack **108**, and a return stack pointer **111** is associated with return stack **109**. As can be seen, each stack pointer receives as input the low 12 bits from bus **101** and has its output connected to the address input of the corresponding stack, as well as through a transmitter **112** or **113** to bus **101**. The data stack data inputs and outputs are buffered through transceiver **114** to provide for better current driving capability. The return stack data may be read from or written to the data bus **101** through the transceiver **116**. In addition, the return stack data may be read from the address counter **117** or written to the address latch **118**.

The RAM address latch **118** and the next address register **119** are the two possible sources for the low 23 bits of address to the program memory (RAM) **121**. The bits **23–30** of program memory address are provided by a page register **120**, allowing up to 2 gigabytes of addressable program memory organized as a group of non-overlapping **8** megabyte pages. When fetching an instruction based on an unconditional branch or subroutine call specified by the address field of the previous instruction, the next address register **119** is used to address memory **121**. For subroutine calls, the contents of the address counter **117** are loaded with the address of the calling program, incremented by 4, and saved in the return stack **109** for use upon subroutine return. The return pointer **111** is decremented before writing to return stack **109**.

Upon subroutine return, return stack **109** provides an address through RAM address latch **118** to address program RAM **121**. RAM address latch **118** retains the address while return stack pointer **111** is incremented to pop the return address off the return stack. In jump, subroutine call, and subroutine return operations, the instruction fetched from program RAM **121** is stored in next address register **119** and the instruction latch **123** at the end of the fetching operation. Thus, each instruc on

directly addresses the next instruction through the next address register **119** and program RAM **121**.

It should be noted that the address counter **117** and next address register **119** are not used as a program counter in the conventional sense. In conventional computers, the program counter is a hardware device used as the primary means of generating addresses for program memory whose normal operation is to increment in some manner while accessing sequential instructions. In computer **100**, the next address register **119** is a simple holding register that is used to hold the address of the next instruction to be fetched from memory. The value of the next address register **119** is determined by an address field contained within the previous instruction executed, NOT from incrementing the previous register value. The address counter **117** is not directly involved in computing instruction addresses; it is only used to generate subroutine return addresses. Thus, computer **100** uses address information in each instruction to determine the address of the next instruction to be executed for high level language programs.

Program RAM **121** is organized as a 32-bit program memory addressable for full-words only on evenly divisible by 4 byte addresses. Computer **100** provides a minimum quantity of 512 kilobytes of program memory, with expansion of up to 8 megabytes of program memory possible. A minor modification of the memory expansion boards, employed to allow for decoding more boards, allows use of up to 2 gigabytes of program memory. Program memory words of 32 bits are read from or written to the data bus **101** through transceiver **123**. Additionally, single byte values with the high 24 bits set to **0** may be read and written to any byte (within each 32-bit word) in memory through the byte addressing and data routing block **122**.

Provisions have been made to incorporate a microcode-controlled floating point math coprocessor **124** into the design, but such a processor has not yet been implemented in the preferred embodiment. The floating point coprocessor **124** would take its instructions not from a separate microcode memory, as is the usual design practice, but rather directly from program memory.

The thirty-two bit arithmetic logic unit (ALU) **126** has its A input connected to a data high register (DHI) **127** and its B input connected to the data bus **101** through a transparent latch **128**. The output of the ALU **126** is connected to a multiplexer **129** that provides for data pass-through, single bit shift left and shift right operations, and a byte rotate right operation. The output of ALU **126** is always fed back into the DHI register **127**. The DHI register **127** is connected to data bus **101** through a data transmitter **130**.

A data low register (DLO) **131** is connected via a bidirectional path to the data bus **101**, and its shift in/out signals are connected to the multiplexer **129** to provide a 64-bit shifting capability.

The opcode portion of program RAM **121** is connected to instruction latch **125** for the purpose of holding the next opcode to be executed by the machine. This instruction latch **125** is decoded according to existing interrupt information from interrupt register **126** and conditional branching information from the condition code register **127** to form the contents of the micro-program counter **129**. The micro-program counter **129** forms a 12 bit address into micro- program memory **131**. The three low bits of the address into micro-program memory **131** are generated from a combination of the micro-address constant inputs and decoding of the condition select field to allow for conditional branching. The contents of the output of the decoding/address logic **128** and the micro-program counter **129** may be read to data bus **101** for diagnostic and interrupt processing purposes through bus driver **130**.

Micro-program memory **131** is a 32-bit high speed memory of 4 kilowords in length. Its data may be read or written to data bus **101** through transceiver **132**, providing a writable instruction set capability. During program execution, its data is fed into the micro-instruction register **133** to provide control signals for operation. Micro-instruction register **133** may be read to data bus **101** through transmitter **134** for diagnostic purposes.

The detailed schematics of the various integrated circuits forming computer **100** are shown in FIGS. **3–89**. Narrative text preceding each group of figures gives descriptions of each signal mnemonic used in the schematics. Other than to identify general features of these circuits, they will not be described in detail, the detail being ascertainable from the hardware themselves. However, some general comments are in order.

Computer **100** in its preferred embodiment is designed for construction on five boards which take five expansion slots in a personal computer. It is addressed with conventional 8088 microprocessor IN and OUT port instructions. It uses 32-bit data paths and 32- bit horizontal microcode (of which bits only **30** are actually used.) It operates on a jumper- and crystal-oscillator controlled micro-instruction cycle period which is preferably set at 150 ns. Most of the logic is the 74ALS series. The ALU is composed of eight 74F181 integrated circuits with carry-lookahead logic. Stack and microcode memory chips are 35 ns CMOS 4-bit chips. Program memory is 120 ns low power CMOS 8-bit memory chips. Since simple primitives are only two clock cycles long, this gives a best case operating speed of 3.3 million basic high level stack operations per second (MOPS). In actual program operation, the average instruction would take just over two cycles, exclusive of complex micro-instructions such as multiplication, division, block memory moves, etc. This, combined with the fact that subroutine calls are zero-cost operations when combined in an instruction with an opcode, gives an average operational speed of approximately 3.5 MOPS.

Variable benchmarks show speed increases of 5 to 10 times over an 80286 running at 8 MHz with zero-wait-state memory. An expert system benchmark shows an even more impressive performance of in excess of 640,000 logical inferences per second.

Instruction decoding requires a 2-cycle minimum on a microcode word definition.

SUMMARY OF FIGURES

The following is a summary of the figures that will be referred to in the detailed description of the preferred embodiment. The figures are organized into general block diagrams and five groups corresponding to the five printed circuit boards in the preferred embodiment.

FIGURE NUMBER	FILE NAME	DESCRIPTION OF CONTENTS
SYSTEM BLOCK DIAGRAM		
1	SBLOCK	ALU AND MEMORY ADDRESS BLOCK DIAGRAM

are used to select the decoded address to any bank of eight ports in the port address space. FIG. 4 shows the decoders IC11 and IC12 which generate control signals based on the lowest bits of the port addresses. In common usage, the preferred embodiment uses eight output ports and three input ports as follows:

PORT	FUNCTION
OUTPUT	
300	DATA BUS (AUTOMATICALLY SEQUENCED FOR 4 BYTES)
301	MIR (WRITE 4 TIMES JUST LIKE WRITE0)
302	SINGLE STEP BOARD CLOCK
303	START BOARD
304	STOP BOARD
305	SET DMA MODE
306	RESET DATA BUS SEQUENCER & DMA MODE
307	SERVICE REQUEST REG & INTERUPT
INPUT	
300	DATA BUS (AUTOMATICALLY SEQUENCED FOR 4 BYTES)
301	MIR (READ 4 TIMES JUST LIKE READ0)
302	STATUS REGISTER (8 BITS)

FIG. 5 shows the generation of control signals and direct memory access (DMA) handshaking signals for the host interface. The host board is capable of accepting high-speed DMA transfers to or from host computer 103 memory directly to and from computer 100 memory. FIGS. 6–12 show the data paths for conversion between an 8-bit host 103 data bus and the 32-bit data bus 101, as well as the buffering for data and control signals on the ribbon cables connecting the host card to the interface card described next. FIGS. 13–14 show the connector arrangements for the host card to host computer bus connector and for the host card to interface card connectors.

The Interface And Stack Card. The interface and stack card (called the interface card) described by FIGS. 15–40 performs a dual function: It serves as the control for bus transfers from the host card and within computer 100 over data bus 101, and provides both the data stack means 108 and the return stack means 109. FIGS. 15–16 show storage for bits 0–15 of the microcode memory and the micro-instruction register. The micro-instruction format is discussed in Appendix B.

FIG. 17 shows the service request register IC58 which is used by the host computer 103 to request one of 255 possible programmable service types from the computer 100. Also shown is the status register IC57 which is used by computer 100 to signal a request for service from host computer 103. FIGS. 18–20 show data and control signal buffers between the host card and the interface card.

FIGS. 21–22 show the clock generating circuits for computer 100. Jumpers J0 through J3 in FIG. 21, along with a socket to change the crystal oscillator used for OS0 allow selection of a wide range of oscillator frequencies. The preferred frequency for the preferred embodiment is 5.0 million Hertz. FIG. 22 shows that a fast clock FASTC is generated that is several nanoseconds ahead in phase of the system clock XCLK for the purpose of satisfying hold times of chips that require data to be valid after the clock rising edge. FIG. 23 shows the data bus 101 source and destination decoders. The devices in this figure generate signals to select only one device to drive data bus 101 and one device to receive data from bus 101. FIG. 24 shows miscellaneous control gates for microcode memory and the micro-instruction register.

FIGS. 25–28 show the data stack means. The data stack has a 12-bit up/down counter that may be incremented, decremented, or loaded from data bus 101 at the end of every clock cycle. The use of fast static RAM chips for the stack memory itself allows the data stack 108 to be read or written and then the stack pointer 110 to be changed on each clock cycle. FIGS. 30–34 show the return stack means. The implementation of the return stack 109 and return stack pointer 111 is very similar to that of the data stack 108 and data stack pointer 110.

FIGS. 35–40 show connector arrangements for transmitting and receiving signals from other cards in the system and from the host adapter card.

The Data, Arithmetic, and Logic Card. The data, arithmetic and logic card (called the data card) described by FIGS. 41–53 performs all arithmetic and logical manipulation of data for computer 100. FIG. 41 shows storage for bits 16–23 of the microcode memory and the micro-instruction register. The micro-instruction format is discussed in Appendix B.

FIGS. 42A–46 show the arithmetic and logic unit (ALU) 126, bus latch 128, data hi register 127, DHI to data bus 100 driver 130, and ALU multiplexer 129. Data from the DHI register 127 and/or the bus data latch 128 flows through the ALU 126 and multiplexer 129 on each clock cycle, then is written back to the DHI register 127. FIG. 47 shows the DLO register 131.

FIG. 48 shows the logic used to detect when the output of the ALU is exactly zero. This is very useful for conditional branching. FIG. 49 shows the generation of the data bus latch 128 control signal and the shift-in bits to the DLO register 131 and the DHI register 127. These shift-in bits are conditioned to provide capability of one-cycle-per-bit multiplication shift-and-conditional-add and non-restoring division algorithms. FIG. 50 shows the conditioning of ALU 126 input control signals to likewise provide for efficient multiplication and division functions.

FIGS. 51–53 show connector arrangements for transmitting and receiving signals from other cards in the system.

The Address Card. The address card described by FIGS. 54–75 performs the memory addressing functions, microcoded control and branching functions, and memory data manipulation functions for computer 100. FIG. 54 shows storage for bits 24–31 of the microcode memory and the micro-instruction register. The micro-instruction format is discussed in Appendix B.

FIG. 55 shows the arrangement of the RAM address latch 118. The RAM address latch is used to address program memory for all non-instruction operations, for return from subroutine operations, and passes data through for DMA transfers with host 103. FIGS. 56–58 show the address counter 117. The address counter 117 may be incremented and passed through the address latch 118 to step through memory one word at a time during DMA access or block memory operations. The address counter 117 is also incremented when performing a subroutine call operation in order to save a correct subroutine return address in return stack 109. FIG. 59 shows the next address register 119 and page register 120. The next address register is used to store the address field of an instruction that points to the memory address of the next instruction during the instruction fetch and decode operation.

FIG. 60 shows the logic used to control return stack 109 and return stack pointer 111. In particular, this logic implements the subroutine call and return control operations for the return stack means. FIG. 61 shows the instruction latch 125 and micro-program counter 129. FIG. 62 shows the interrupt status register 126. Interrupts are set by a processor condition pulling a "PR" pin of IC53–IC56 low, causing the flip-flop to activate, or by loading a one bit from data bus 101. Any one or more active interrupts causes an interrupt at the next instruction decoding operation. An interrupt mask bit from IC53 pin 5 is used to allow masking of all further interrupts during interrupt processing.

FIG. 63 shows the condition code register 127. This register is set at the end of every clock cycle, and forms the basis of the lowest bit of the next micro-instruction address fetched during the succeeding clock cycle. FIG. 64 shows a special forcing driver for the microcode-memory address that forces an opcode of 1 during interrupt recognition. FIG. 65 shows a timing chain used to control the 2 cycle instruction fetch and decoding operation.

FIGS. 66–69 show the RAM data to data bus 101 transfer logic shown by block 122 on FIG. 1. This transfer logic allows access of arbitrary bytes within the 32-bit memory organization as well as 32-bit full word access on evenly-divisible-by-four memory address locations.

FIGS. 70–75 show connector arrangements for transmitting and receiving signals from other cards in the system.

The Memory Card. The memory card described by FIGS. 76–89 is a single program memory 121 storage card for computer 100. Computer 100 may have one to sixteen of these cards in operation simultaneously to use up to 8 megabytes of memory.

FIG. 76 shows data buffering logic used to satisfy current driving requirements of the memory chips. Similarly, FIG. 77 shows address buffering logic. FIG. 78 shows the memory board selection, bank selection, and chip selection logic. Jumpers J0–J7 may be set to map the memory board to one of 16 non-overlapping 512 kilobyte locations within the first eight megabytes of the available memory space. Only one memory board is activated at a time. Once the memory board is activated, a particular bank of chips (numbered from 0–3) is enabled selecting a 32 kiloword address within the board. If byte memory access is being used, a single chip within the bank is selected for a single byte operation, otherwise all chips within the bank are enabled.

FIGS. 79–86 show the four banks of four RAM chips each.

FIGS. 87–89 show connector arrangements for transmitting and receiving signals from other cards in the system.

SYSTEM SOFTWARE

Computer 100 in this preferred embodiment uses various software packages, including a FORTH kernel, a cross-compiler, a micro-assembler, as well as microcode. The software for these packages, written using MVP-FORTH, are listed in Appendix A. Further, the microcode format is discussed in Appendix B. The User's Manual (less appendices duplicated elsewhere in this document) is included as Appendix C. Some general comments about the software are in order. The Cross-Compiler. The cross-compiler maintains a sealed vocabulary with all the words currently defined for computer 100. At the base of this dictionary are special cross-compiler words such as IF ELSE THEN : and ;. After cross-compilation has started, words are added to this sealed vocabulary and are also cross-compiled into computer 100. Whenever the keyword CROSS-COMPILER is used, any word definitions, constants, variables, etc. will be compiled to computer 100. However, any immediate operations will be taken from the cross-compiler's vocabulary, which is chained to the normal MVP-FORTH vocabulary.

By entering the FORTH word {, the cross-compiler enters the immediate execution mode for computer 100. All words are searched for in the sealed vocabulary for computer 100 and are executed by computer 100 itself. The "START.." "END" that is displayed indicates the start and the end of execution of computer 100. If the execution freezes in between the start and end, that means that computer 100 is hung up. The cross-compiler builds a special FORTH word in computer 100 to execute the desired definition, then performs a HALT instruction. Entering the FORTH word } will leave the computer 100 mode of execution and return to the cross-compiler. No colon definitions or other creation of dictionary entries should be performed while between { and }.

The FORTH word CPU32 will automatically transfer control of the system to computer 100 via its Forth language cold start command. The host MVP-FORTH will then execute an idle loop waiting for computer 100 to request services. The word BYE will return control back the host's MVP FORTH.

The current cross-compiler can not keep track of the dictionary pointer DP, etc., in computer 100 if it is out of sync with the cross-compiler's copy. This means that no cross- C compiling or micro-assembly may be done after the FORTH of computer 100 has altered the dictionary in any way. This could be fixed at a later date by updating the cross-compiler's variables from computer 100 after every BYE command of computer 100.

Cross-compiled code should be kept to a minimum, since it is tricky to write. After a bare minimum kernel is up and running, computer 100 should do all further FORTH compilation. The Micro-assembler. The micro-assembler is a tool to save the programmer from having to set all the bits for microcode by hand. It allows the use of mnemonics for setting the micro-operation fields in a micro-instruction, and, for the most part, automatically handles the micro-instruction addressing scheme.

The micro-assembler is written to be co-resident with the cross-compiler. It uses the same routines for computer 100 and sealed host vocabulary dictionary handling, etc. Currently all microcode must be defined before the board starts altering its dictionary, but this could be changed as discussed previously.

In the terminology used here, a micro-instruction is a 32-bit instruction in microcode, while a micro-operation is formed by one or more microcode fields within a single micro-instruction.

Appendix B gives a quick reference to all the hardware-defined micro-instruction fields supported by the micro-assembler. The usage and operation of each field of the micro-instruction format is covered in detail in Part Two of the User's Manual included as Appendix C. Since the microcode layout is very horizontal, there is a direct relationship between bit settings and control line inputs to various chips on computer 100. As with most horizontally microcoded machines, as many micro-

operations as desired may take place at the same time, although some operations don't do anything useful when used together. Microcode Definitions Format. The micro-assembler has a few keywords to make life easier for the micro-programmer. The word OP-CODE: starts a microcode definition. The input parameter is the page number from 0-OFF hex that the op-code resides in. For example, the word ± is op-code 7. This means that whenever computer **100** interprets a hex 038xxxxx (where the x's represent don't care bit values), the word ± will be executed in microcode. The character string after OP-CODE: is the name of the op-code that will be added to the cross-compiler and computer **100** dictionaries. It is the programmer's responsibility to ensure that two op-codes are not assigned to the same microcode memory page. The variable CURRENT-OPCODE contains the page currently assigned by OP-CODE:. It may be changed to facilitate multi-page definitions.

The word :: signifies the start of the definition of a micro-instruction. The number before :: must be from 0 to 7, and signifies the offset from 0 to 7 within the current micro-program memory page for that micro-instruction. Micro-instructions may be defined in any order desired. When directly setting the micro-instruction register (MIR) for interactive execution, the word >> may be used without a preceding number instead of the sequence 0 ::.

The word ;; signifies the end of a micro-instruction and stores the micro-instruction into the appropriate location in micro-program memory.

The word ;;END signifies the end of a definition of a FORTH microcoded primitive.

If the FORTH vocabulary is in use, the programmer may single-step microcoded programs. Use the >> word to start a micro-instruction. Instead of using ;;, use ;SET to copy the micro-instruction to the MIR. This allows reading resources of computer **100** to the host **103** with the X@ word or storing resource values with the X! word. Using ;DO instead of ;; will load the instruction into the MIR and cycle the clock once. This is an excellent way of single-stepping microcode. The User's Manual in Appendix C and the Diagnostics of computer **100** given in Appendix A part III provide examples of how to use these features. End/Decode. END and DECODE are the two micro-operations that perform the FORTH NEXT function and perform subroutine calls, subroutine returns, and unconditional branches in parallel with other operations. DECODE is always in the next to last micro-instruction of a micro-coded instruction. It causes the interrupt register **126** to be clocked near the falling clock edge, and loads highest 9 bits of the instruction into the instruction latch **125** at the following rising clock edge. Thereafter, instruction fetching and decoding proceeds according to the actions described in Appendix C part II. END is a micro-operation that marks the last instruction in a program and forces a jump to offset 0 of the next instruction's microcoded memory page. Microcode Next Address Generation. The micro-assembler automatically generates an appropriate microcode jump to the next sequential offset within a page. This means that if a 3 is used before the :: word, then the micro-assembler will assume that the next micro-instruction is at offset 4 unless a JMP= micro-instruction is used to tell it otherwise.

The JMP= micro-operation allows forcing non-sequential execution or conditional branching simultaneously with other micro-operations. A JMP=000, JMP=001, ... , JMP=111 command forces an unconditional microcode jump to the offset within the same page specified by the binary operand after JMP=. For example, JMP=101 would force a jump to offset 5 for the next micro-cycle.

A conditional jump allows jumping to one of the two locations depending on the value of one of the 8 condition codes. The unconditional jump described in the preceding paragraph is just a special conditional jump in which the condition picked is a constant that is always set to 0 or 1. The sign bit conditional jump is used below as an example.

A conditional jump sets the lowest bit of the next micro-instruction address to the value of the condition that was valid at the end of the previous microcycle. The syntax is JMP=00S, where "S" can be replaced by any of the conditions: Z, L, C, S, 0, 1. The first two bits are always numeric, indicating the top two binary bits of the jump destination address within the micro-program memory page. The example JMP=10S would jump to offset 4 within the micro-program memory page if the sign bit were 0, and location 5 if it were 1.

Appendix C is the user manual for computer **100**, and describes other information of interest in the operation of the preferred embodiment of the invention.

It will thus be appreciated that the described preferred embodiment achieves the desired features and advantages of the invention. While the invention has been particularly shown and described with reference to the foregoing preferred embodiment, it will be understood by those skilled in the art that other changes in form and detail may be made therein without departing from the spirit and scope of the invention, as defined in the claims.

What we claim is:

1. A writable instruction set computer comprising:

data bus means for transferring data having a predetermined number of bits;

addressable and writable main program memory means coupled to said data bus means for storing macrocode, including instructions having the predetermined number of bits, and for storing data from and loading stored data onto said data bus means;

memory address logic means coupled to said data bus means and said main program memory means for addressing said main program memory means;

addressble and writable micro-program memory means coupled to said main program memory means for storing microcode instructions addressed by the macrocode instructions;

arithmetic logic unit (ALU) means coupled to said data bus means for performign operations on data received from said data bus means as defined by the microcode stored in said micro-program memory means;

data stack memory means coupled to said data bus means for storing data received from said data bus means for use during program execution;

return stack memory means physically separate from said main memory means, and coupled to said data bus means and to said memory address logic means for storing subroutine return address used during program execution, said memory address logic means addressing said main program memory means with the subroutine return address stored in said return stack memory means while said ALU means performs operations on data transferred from said data stack memory means on said data bus means;

clock means for generating a cyclic clock signal; and

execution control logic means coupled to said microprogram memory means, ALU means, data stack memory means, return stack memory means, data bus means, and clock means for executing the microcode instructions, including performing only one data transfer on said data bus means for each clock signal cycle;

said data bus means providing only one communication path for transferring bidirectionally data between said ALU means, said data stack memory means and said main program memory means.

2. A computer according to claim 1 wherein said main program memory means stores each instruction as the combination of an opcode and a main program memory address.

3. A computer according to claim 2 wherein said address included in said instruction comprises the address of the location of the succeeding instruction in said main program memory.

4. A computer according to claim 3 wherein said execution control logic means is further for executing the operation specified by the opcode of a current macrocode instruction while, simultaneously with the operation executing, said memory address logic means fetches the macrocode instruction corresponding to the address included in the current macrocode instruction.

5. A computer according to claim 4 wherein said main program memory means further stores for a machine language program instruction, an indicator indicating whether the succeeding operation is a subroutine return, a d said memory address logic means is further responsive to address information received from said stack memory means for executing a subroutine return simultaneously with the executing of the current operation, when the indicator indicates that the next operation is a subroutine return.

6. A computer according to claim 5 wherein said main program memory means stores a condition code having one of a plurality of values including a predetermined value, and a macrocode instruction comprises a conditional branch opcode requiring execution of a subroutine call if the value of the condition code is the predetermined value, and a subroutine call address, said memory address logic means further executing the subroutine call while said execution control logic means executes the conditional branch opcode, said memory address logic means being responsive to said execution control logic means for aborting the execution of the subroutine call if the value of the condition code is not the predetermined value.

7. A computer according to claim 1 wherein said ALU means comprises first and second input ALU ports and an output ALU port, said computer further comprising transparent latch means having an input latch port coupled to said data bus means and an output latch port coupled to said first input ALU port, said latch means being controllable for either transferring data input on said input latch port to said output latch port or retaining data input on said input latch port without it appearing on said output latch poret, and data register means having a register input port coupled to to said output ALU port and a register output port coupled to said second input ALU port and to said data bus means, said transparent latch means being for storing temporarily data received from said data bus means while data stored in said data register means is output to said data bus means.

8. A computer according to claim 1 wherein each macrocode instruction includes an opcode, and further comprising:

data stack pointer means coupled to said data bus means and said data stack memory means for only storing one pointer pointing to an element in said data stack memory mean,s wherein said execution control logic means is further for setting the pointer to point to any element in said data stack memory means without altering the contents of said stack memory means, the one pointer being the only means for accessing an element in said data stack memory means; and

interrupt means coupled to said execution control logic means, and responsive to interrupt signals for generating an interrupt opcode when an interrupt signal indicates that the program execution is to be interrupted;

said execution control logic means being responsive to the interrupt opcode for itnerrupting program execution by isnerting the interrupt opcode in place of the next macrocode opcode, and thereby interrupting the program execution only when a next macrocode opcode is to be executed by said execution control logic means, said execution control logic means further controlling execution of the macrocode such that the pointer stored in said data stack pointer means is set to point to a predetermined data stack element prior to executing each new macrocode opcode, whereby the pointer can be changed to point to different data stack elements during execution of a macrocode opcode