United States Patent [19]

Rasala et al.

[11] **4,386,399**

[45] **May 31, 1983**

[54] **DATA PROCESSING SYSTEM**

[75] Inventors: **Edward Rasala**, Westboro; **Steven Wallach**, Framingham; **Carl J. Alsing**, Hopkington; **Kenneth D. Holberger**, North Grafton; **Charles J. Holland**, Northboro; **Thomas West**, Boxboro; **James M. Guyer**, Marlboro; **Richard W. Coyle**, Dunstable; **Michael L. Ziegler**, Whitinsville; **Michael B. Druke**, Chelmsford, all of Mass.

[73] Assignee: **Data General Corporation**, Westboro, Mass.

[21] Appl. No.: **143,561**

[22] Filed: **Apr. 25, 1980**

[51] **Int. Cl.**[3] **G06F 9/36**

[52] **U.S. Cl.** **364/200**

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

[56] **References Cited**

U.S. PATENT DOCUMENTS

3,934,232	1/1976	Curley et al.	364/200
4,050,058	9/1977	Garlic	364/200
4,051,461	9/1977	Hashimoto et al.	364/200
4,160,289	7/1979	Bambara et al.	364/200

Primary Examiner—Joseph F. Ruggiero

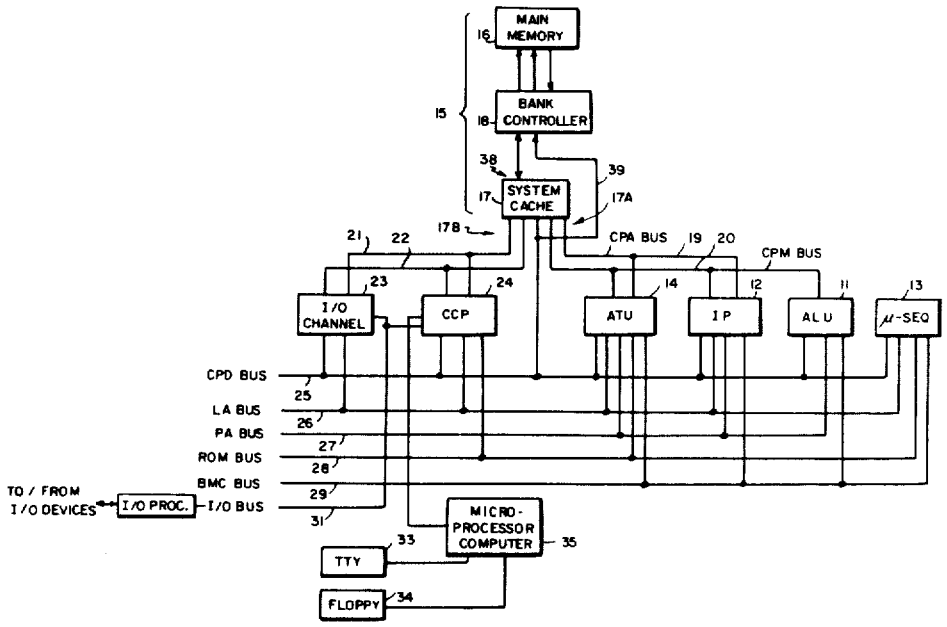
Assistant Examiner—Michael R. Fleming

Attorney, Agent, or Firm—Robert F. O'Connell

[57] **ABSTRACT**

A data processing system which handles thirty-two bit logical addresses which can be derived from either sixteen bit logical addresses or thirty-two bit logical addresses, the latter being translated into physical addresses by unique translation means. The system includes means for decoding macro-instructions of both a basic and an extended instruction set, each macro-instruction containing in itself selected bit patterns which uniquely identify which type of instruction is to be decoded. The decoded macro-instructions provide the starting address of one or more micro-instructions, which address is supplied to a unique micro-instruction sequencing unit which appropriately decodes a selected field of each micro-instruction to obtain each successive micro-instruction. The system uses hierarchical memory storage using eight storage segments (rings), access to the rings being controlled in a privileged manner according to different levels of privilege. The memory system uses a bank of main memory modules which interface with the central processor system via a dual port cache memory, block data transfers between the main memory and the cache memory being controlled by a bank controller unit.

6 Claims, 429 Drawing Figures



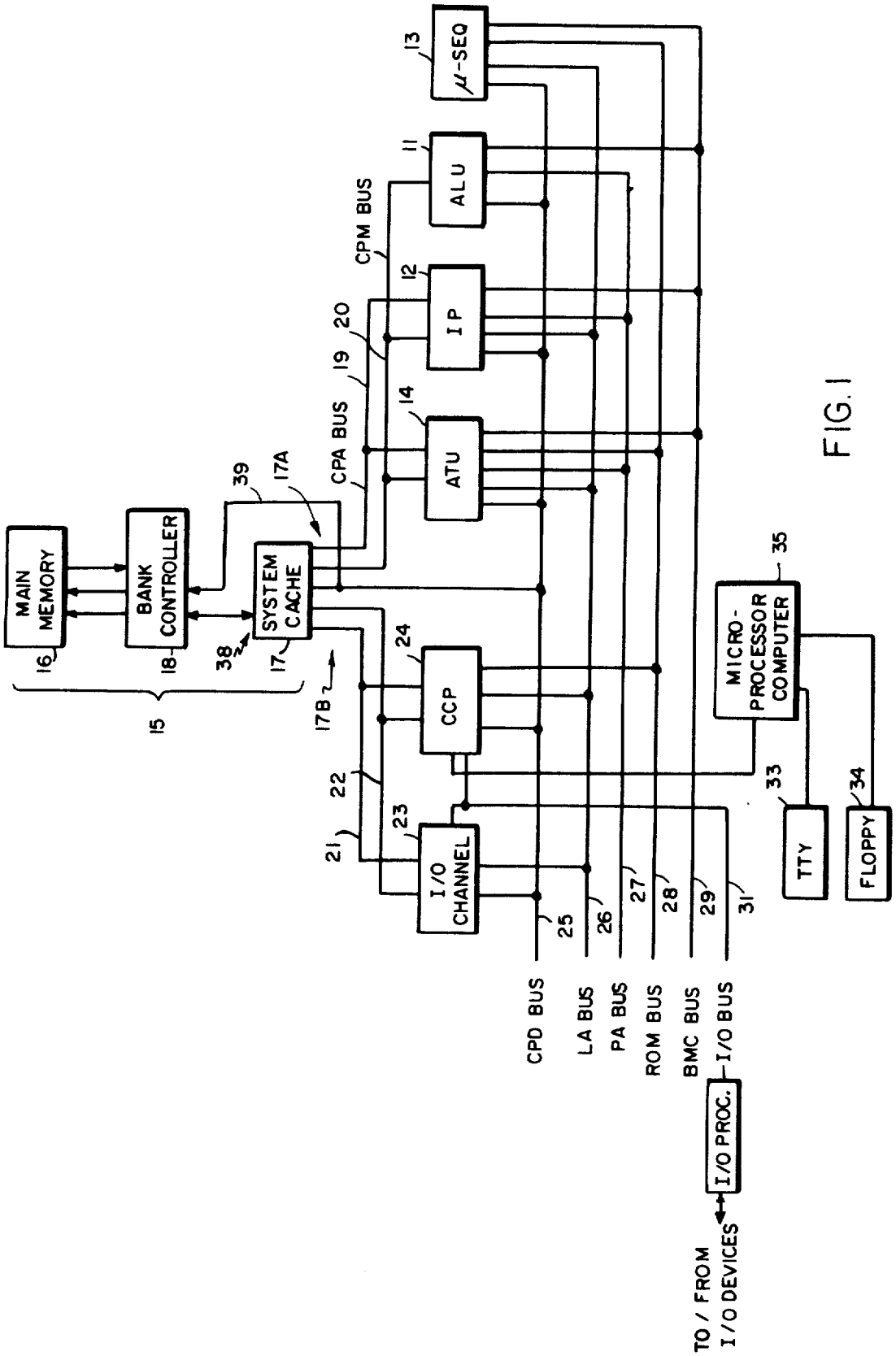
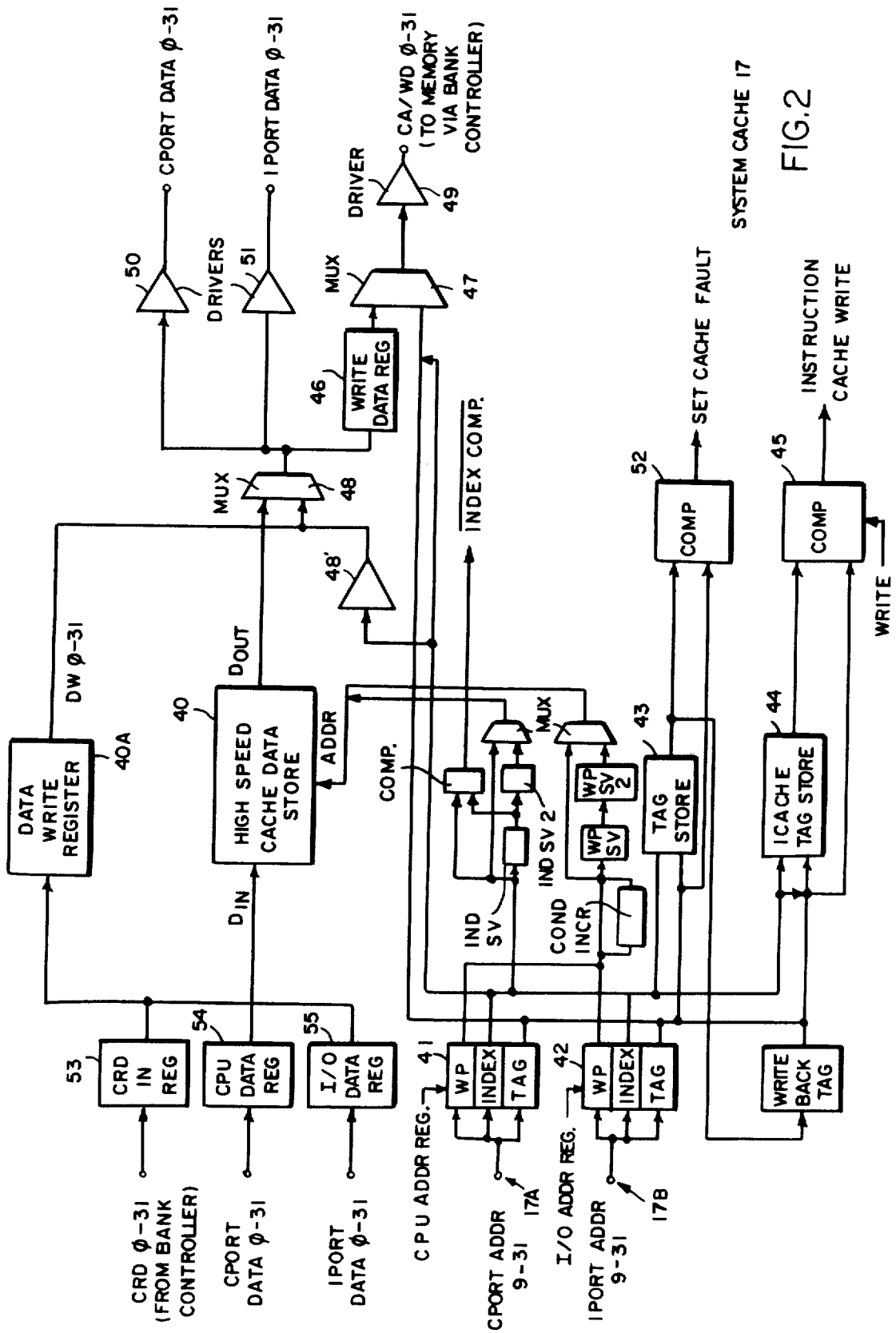

MAIN MEMORY
BANK CONTROLLER
SYSTEM CACHE
μ-SEQ
ALU
IP
ATU
CCP
I/O CHANNEL
MICRO-PROCESSOR COMPUTER
TTY
FLOPPY
I/O PROC.
CPM BUS
CPA BUS
CPD BUS
LA BUS
PA BUS
ROM BUS
BMC BUS
I/O BUS
TO / FROM I/O DEVICES
15
16
17
17A
17B
18
19
20
21
22
23
24
25
26
27
28
29
31
33
34
35
38
39
11
12
13
14

FIG. 1


CPORT DATA Ø-31
I PORT DATA Ø-31
CA/WD Ø-31 (TO MEMORY VIA BANK CONTROLLER)
DRIVER
49
50
DRIVERS
51
MUX
47
WRITE DATA REG
46
MUX
48
48'
INDEX COMP.
SYSTEM CACHE 17
FIG.2
SET CACHE FAULT
INSTRUCTION CACHE WRITE
52
COMP
45
COMP
WRITE
DW Ø-31
DOUT
40
HIGH SPEED CACHE DATA STORE
ADDR
DATA WRITE REGISTER
40A
COMP.
MUX
43
TAG STORE
44
ICACHE TAG STORE
WP SV 2
WP SV
IND SV 2
IND SV
COND INCR
DIN
53
CRD IN REG
54
CPU DATA REG
55
I/O DATA REG
41
WP
INDEX
TAG
42
WP
INDEX
TAG
WRITE BACK TAG
CRD Ø-31 (FROM BANK CONTROLLER)
CPORT DATA Ø-31
IPORT DATA Ø-31
CPU ADDR REG.
CPORT ADDR 9-31
17A
I/O ADDR REG.
IPORT ADDR 9-31
17B

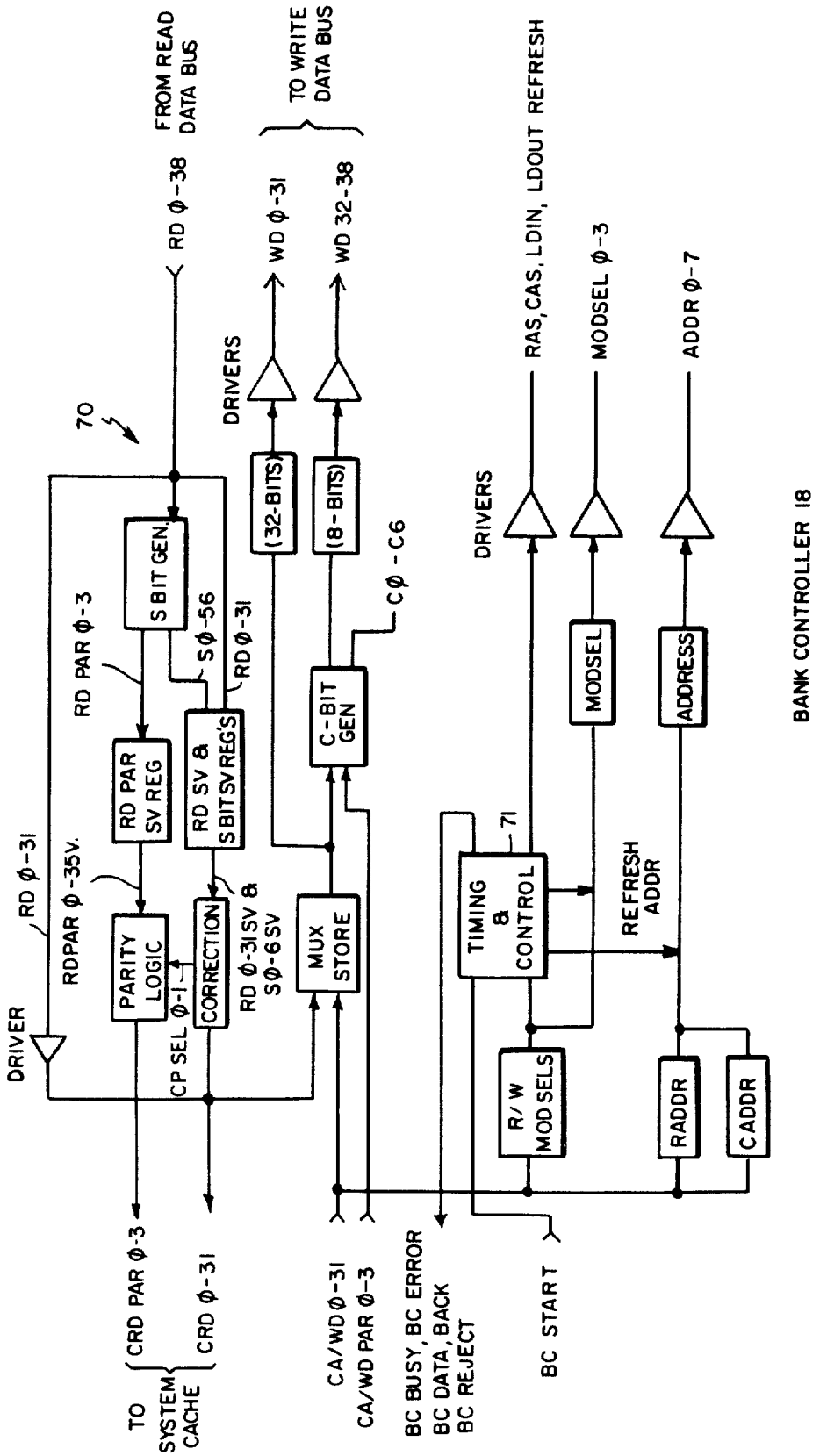

FROM READ DATA BUS
RD Ø-38
TO WRITE DATA BUS
WD Ø-31
WD 32-38
RAS, CAS, LDIN, LDOUT REFRESH
MODSEL Ø-3
ADDR Ø-7
70
DRIVERS
(32-BITS)
(8-BITS)
DRIVERS
S BIT GEN.
CØ-C6
RD PAR Ø-3
SØ-56
RD Ø-31
MODSEL
ADDRESS
BANK CONTROLLER 18
RD PAR SV REG
RD SV & S BIT SV REG'S
C-BIT GEN
RD Ø-31
RDPAR Ø-35V.
71
TIMING & CONTROL
REFRESH ADDR
PARITY LOGIC
CORRECTION
RD Ø-31 SV & SØ-6 SV
MUX STORE
DRIVER
CP SEL Ø-1
R/W MOD SELS
RADDR
CADDR
CRD PAR Ø-3
CRD Ø-31
TO SYSTEM CACHE
CA/WD Ø-31
CA/WD PAR Ø-3
BC BUSY, BC ERROR
BC DATA, BACK
BC REJECT
BC START

FIG. 3

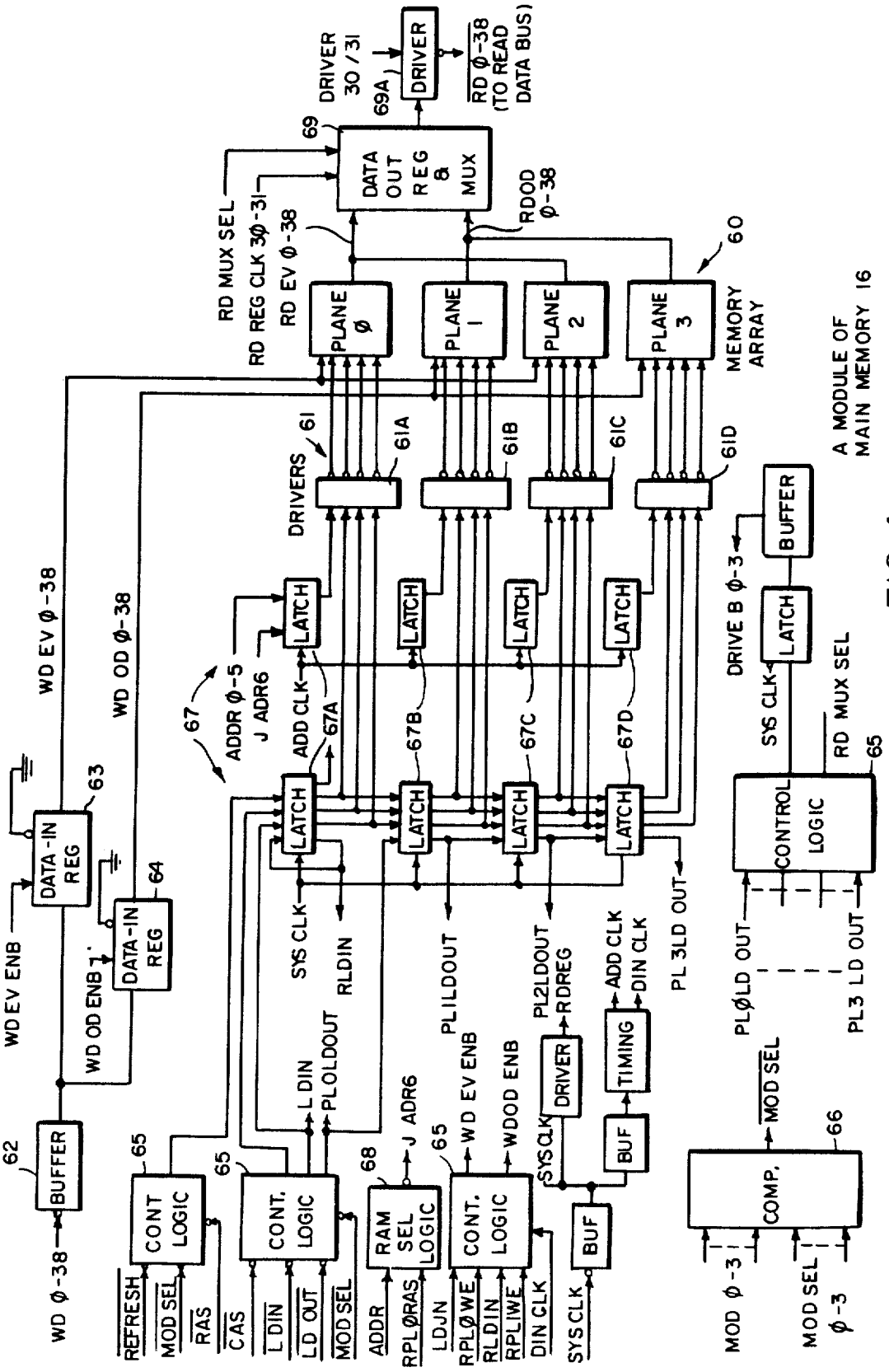
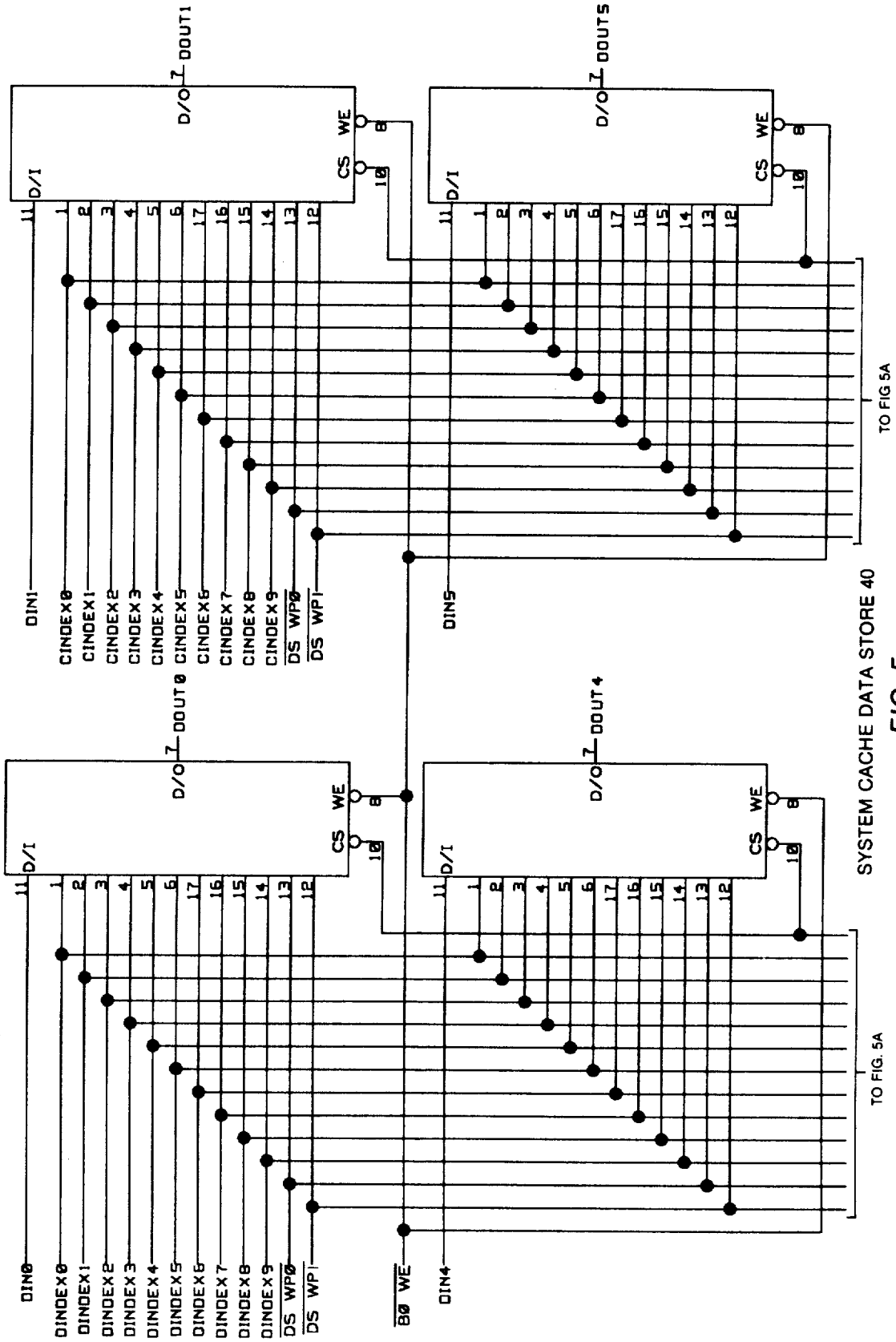

DRIVER 30/31
69A
DRIVER
RD Ø-38 (TO READ DATA BUS)
69
DATA OUT REG & MUX
RD MUX SEL
RD REG CLK 3Ø-31
RD EV Ø-38
RDOD Ø-38
60
PLANE Ø
PLANE 1
PLANE 2
PLANE 3
MEMORY ARRAY
A MODULE OF MAIN MEMORY 16
DRIVERS
61
61A
61B
61C
61D
BUFFER
DRIVE B Ø-3
LATCH
SYS CLK
RD MUX SEL
WD EV Ø-38
WD OD Ø-38
67
ADDR Ø-5
J ADR6
ADD CLK
67A
67B
67C
67D
63
DATA-IN REG
64
DATA-IN REG
65
CONTROL LOGIC
WD EV ENB
WD OD ENB
SYS CLK
RLDIN
PLILDOUT
PL2LDOUT
RDREG
ADD CLK
DIN CLK
PL 3LD OUT
PLØLD OUT
PL3 LD OUT
L DIN
PLØLDOUT
J ADR6
WD EV ENB
WDOD ENB
SYSCLK
DRIVER
TIMING
BUF
MOD SEL
66
COMP.
62
BUFFER
WD Ø-38
65
CONT LOGIC
65
CONT. LOGIC
68
RAM SEL LOGIC
65
CONT. LOGIC
BUF
REFRESH
MOD SEL
RAS
CAS
L DIN
LD OUT
MOD SEL
ADDR
RPLØRAS
LDIN
RPLØWE
RLDIN
RPLIWE
DIN CLK
SYS CLK
MOD Ø-3
MOD SEL Ø-3

FIG. 4



SYSTEM CACHE DATA STORE 40

FIG. 5

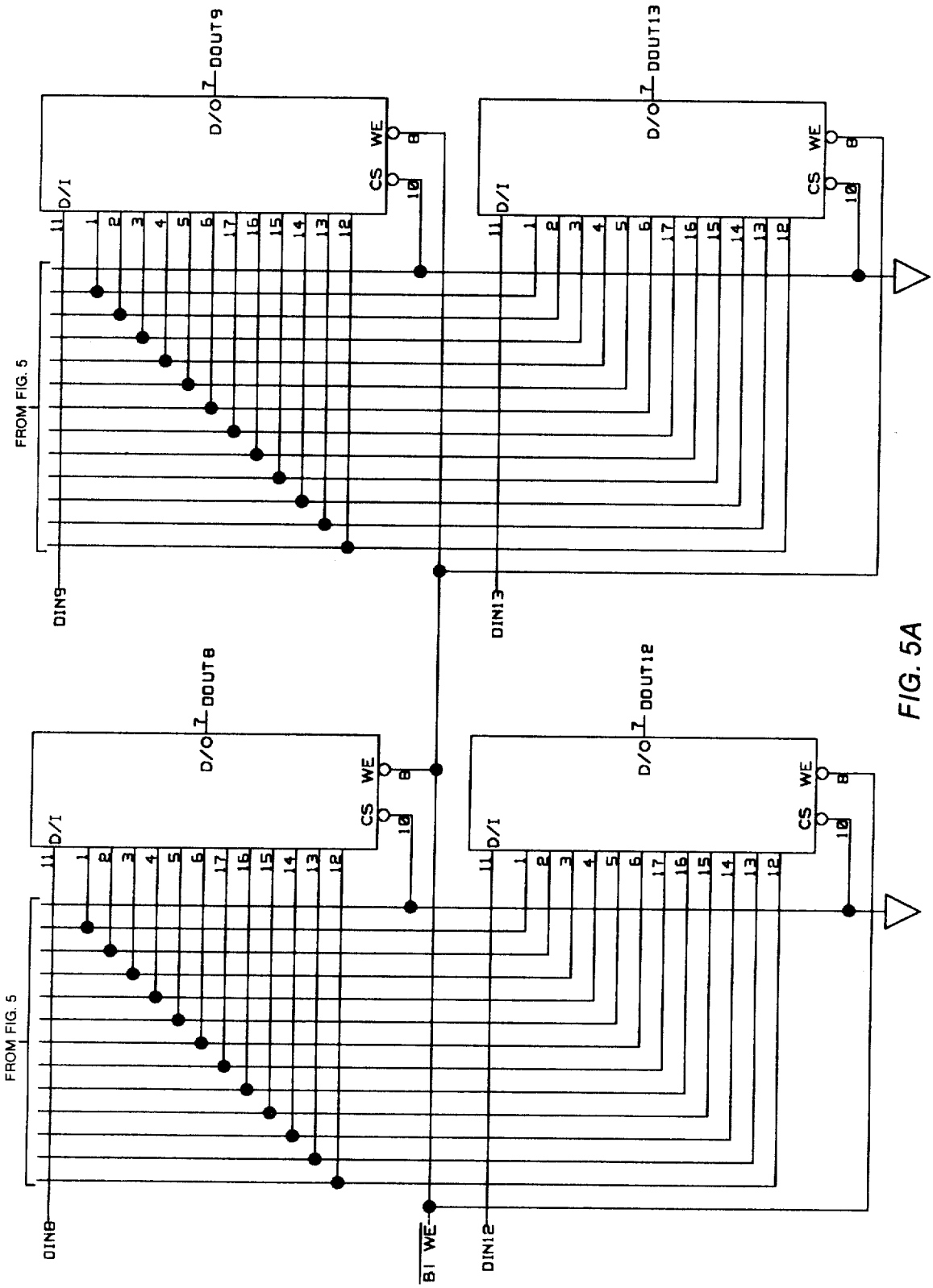

FROM FIG. 5
DIN9
D/I
D/O
DOUT9
CS
WE
DIN13
DOUT13
DIN8
DOUT8
BI WE
DIN12
DOUT12

FIG. 5A

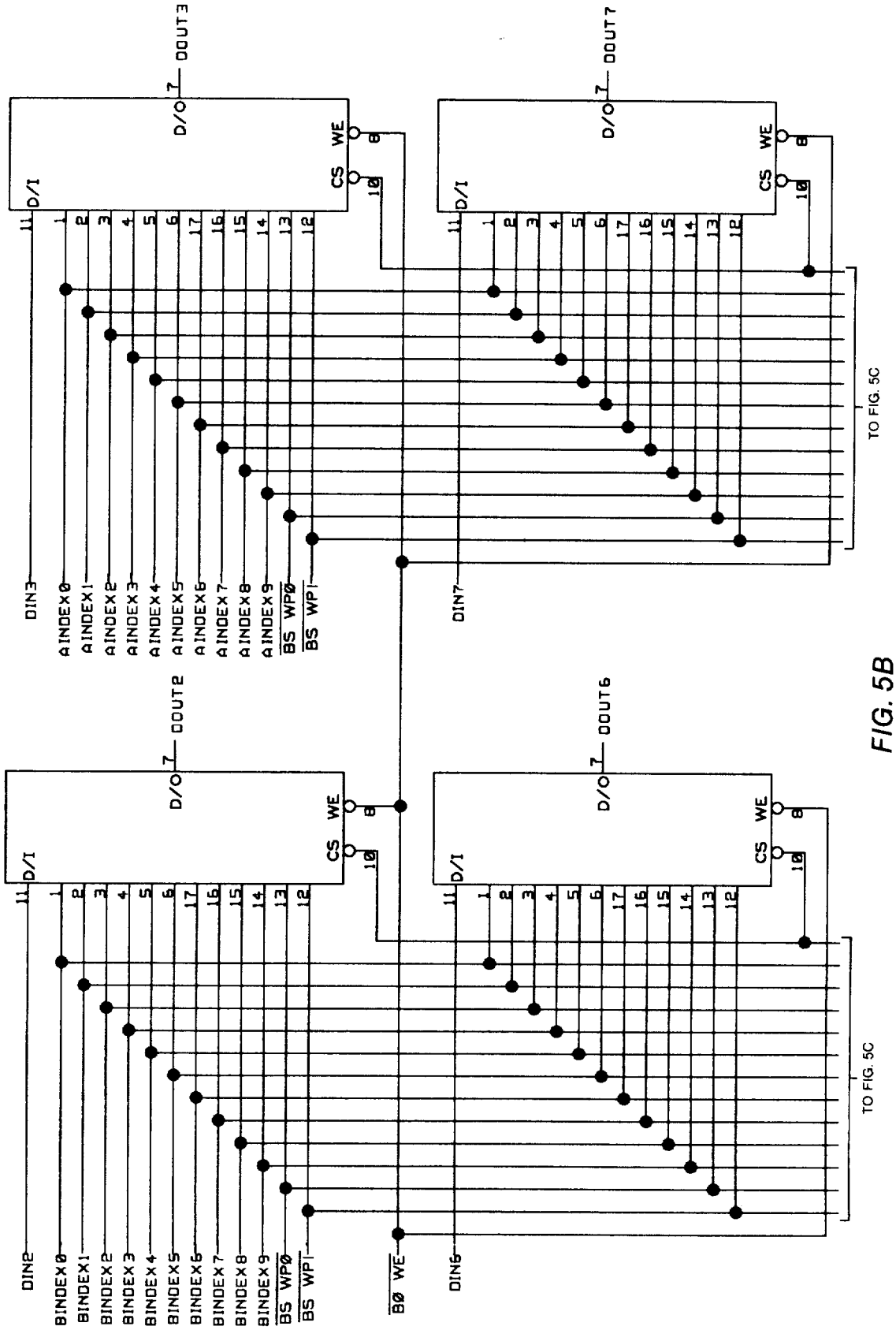

DOUT3
D/O
D/I
CS
WE
DIN3
AINDEX0
AINDEX1
AINDEX2
AINDEX3
AINDEX4
AINDEX5
AINDEX6
AINDEX7
AINDEX8
AINDEX9
BS WP0
BS WP1
DOUT7
DIN7
TO FIG. 5C
DOUT2
DIN2
BINDEX0
BINDEX1
BINDEX2
BINDEX3
BINDEX4
BINDEX5
BINDEX6
BINDEX7
BINDEX8
BINDEX9
BS WP0
BS WP1
B0 WE
DOUT6
DIN6
TO FIG. 5C

FIG. 5B

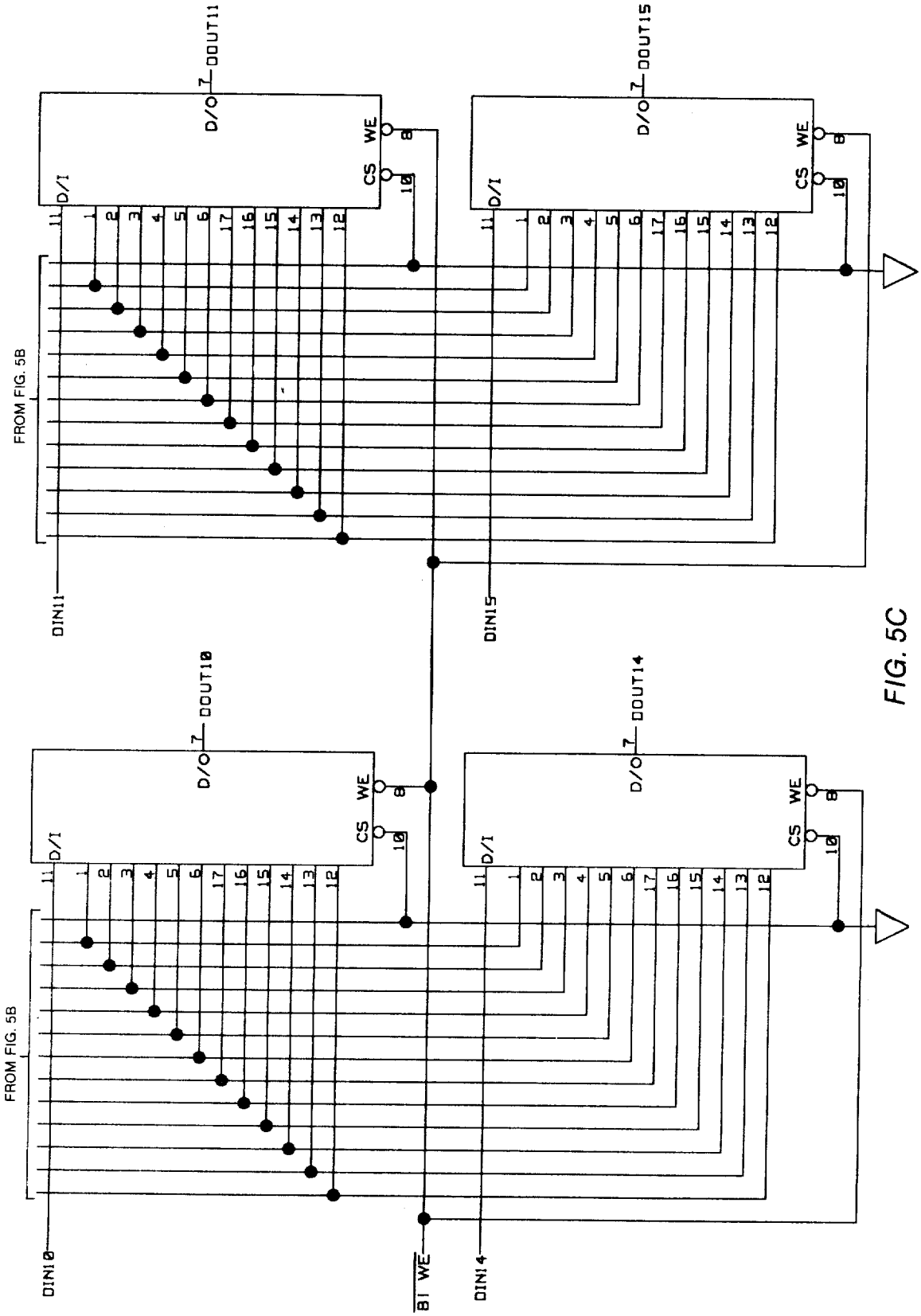

FROM FIG. 5B
DIN11
D/I
D/O
DOUT11
CS
WE
DIN15
DOUT15
DIN10
DOUT10
BI WE
DIN14
DOUT14

FIG. 5C

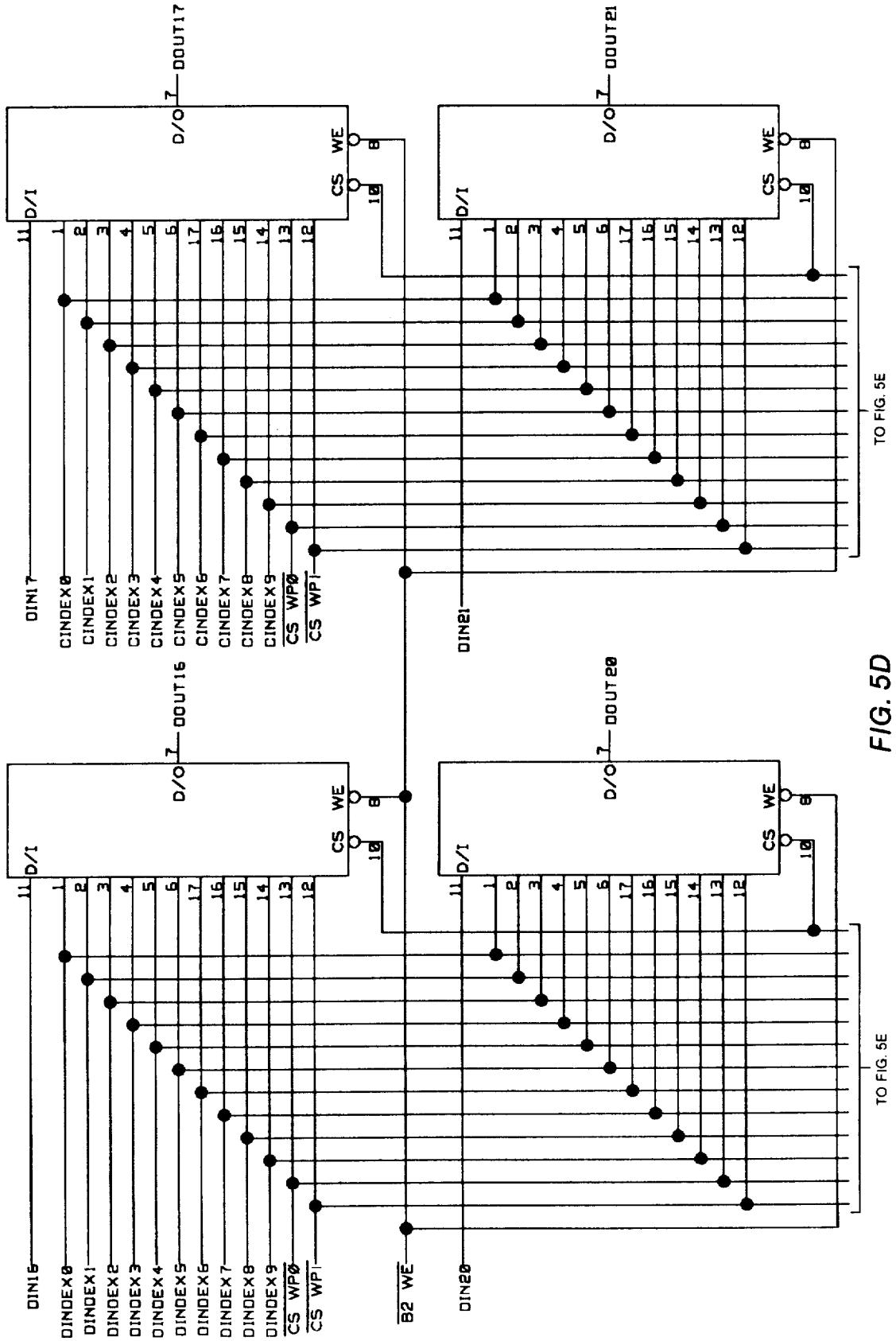

DOUT17
DOUT21
DOUT16
DOUT20
D/O
D/I
CS
WE
DIN17
DIN21
DIN16
DIN20
DINDEX0
DINDEX1
DINDEX2
DINDEX3
DINDEX4
DINDEX5
DINDEX6
DINDEX7
DINDEX8
DINDEX9
CS WP0
CS WP1
B2 WE
TO FIG. 5E
TO FIG. 5E

FIG. 5D

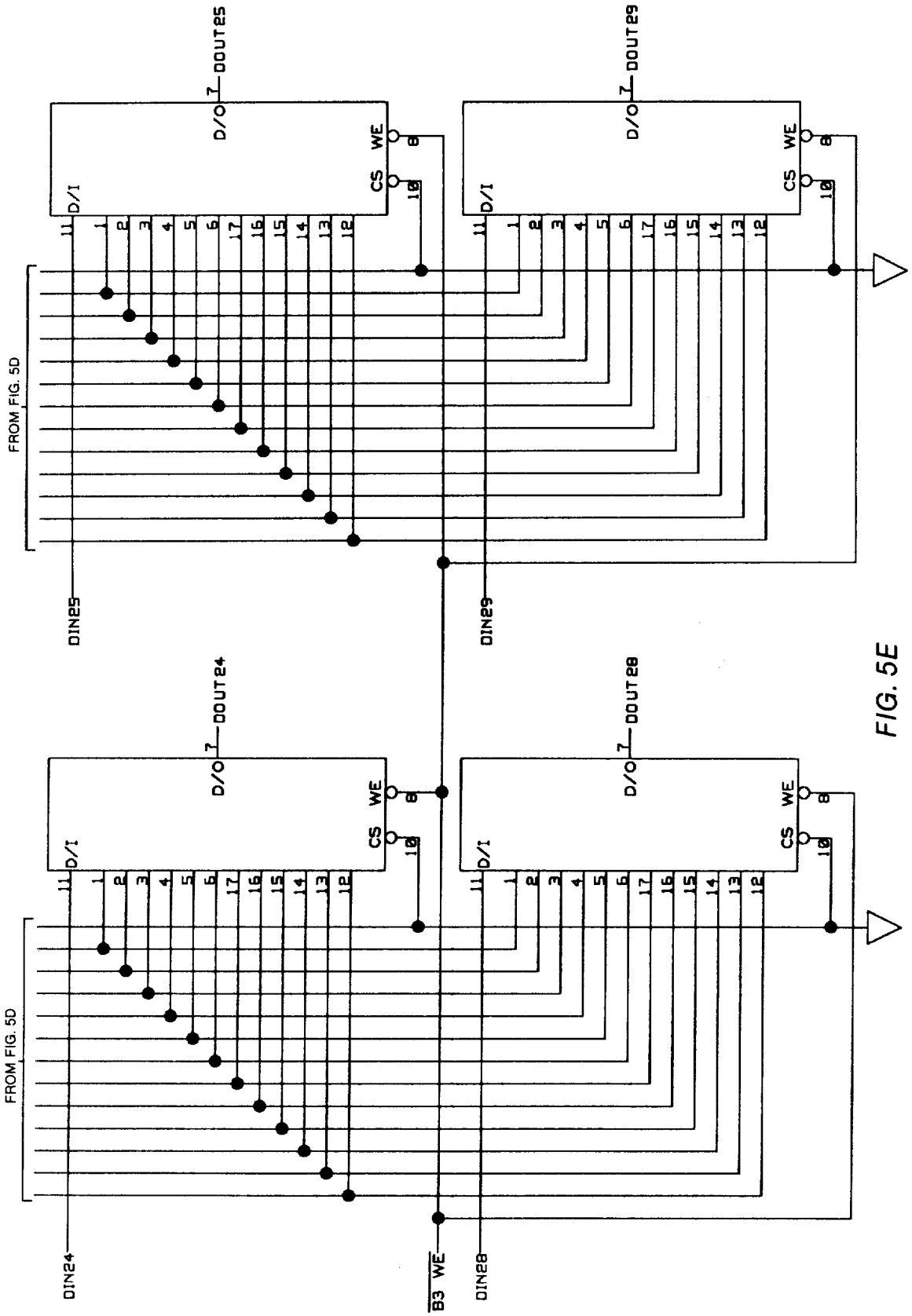

FROM FIG. 5D
DIN25
DIN29
DIN24
DIN28
B3 WE
D/I
D/O
CS
WE
DOUT25
DOUT29
DOUT24
DOUT28

FIG. 5E

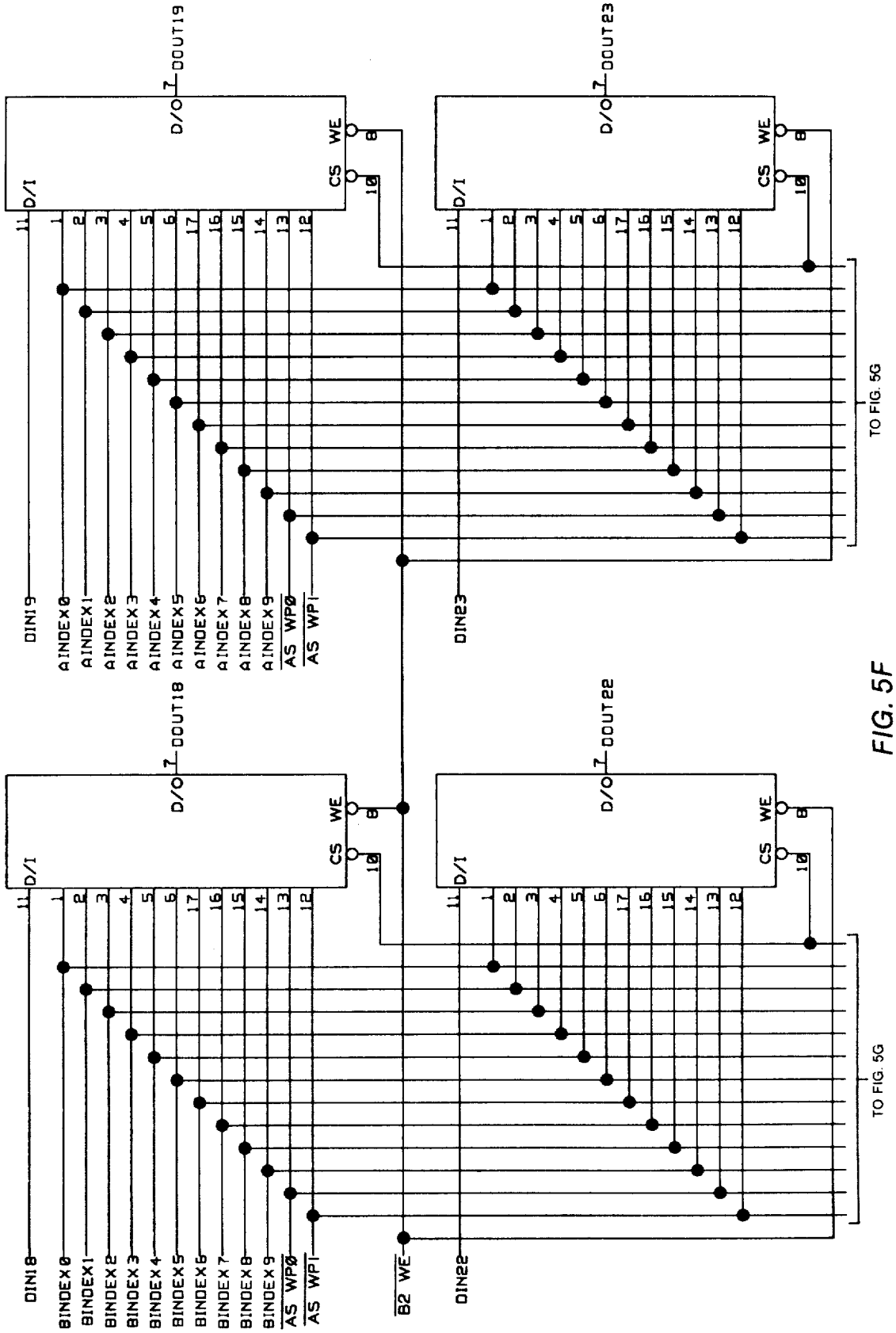

DIN19
AINDEX0
AINDEX1
AINDEX2
AINDEX3
AINDEX4
AINDEX5
AINDEX6
AINDEX7
AINDEX8
AINDEX9
AS WP0
AS WP1
DOUT19
DOUT23
DIN23
DIN18
BINDEX0
BINDEX1
BINDEX2
BINDEX3
BINDEX4
BINDEX5
BINDEX6
BINDEX7
BINDEX8
BINDEX9
AS WP0
AS WP1
B2 WE
DIN22
DOUT18
DOUT22
D/I
D/O
CS
WE
TO FIG. 5G
TO FIG. 5G

FIG. 5F

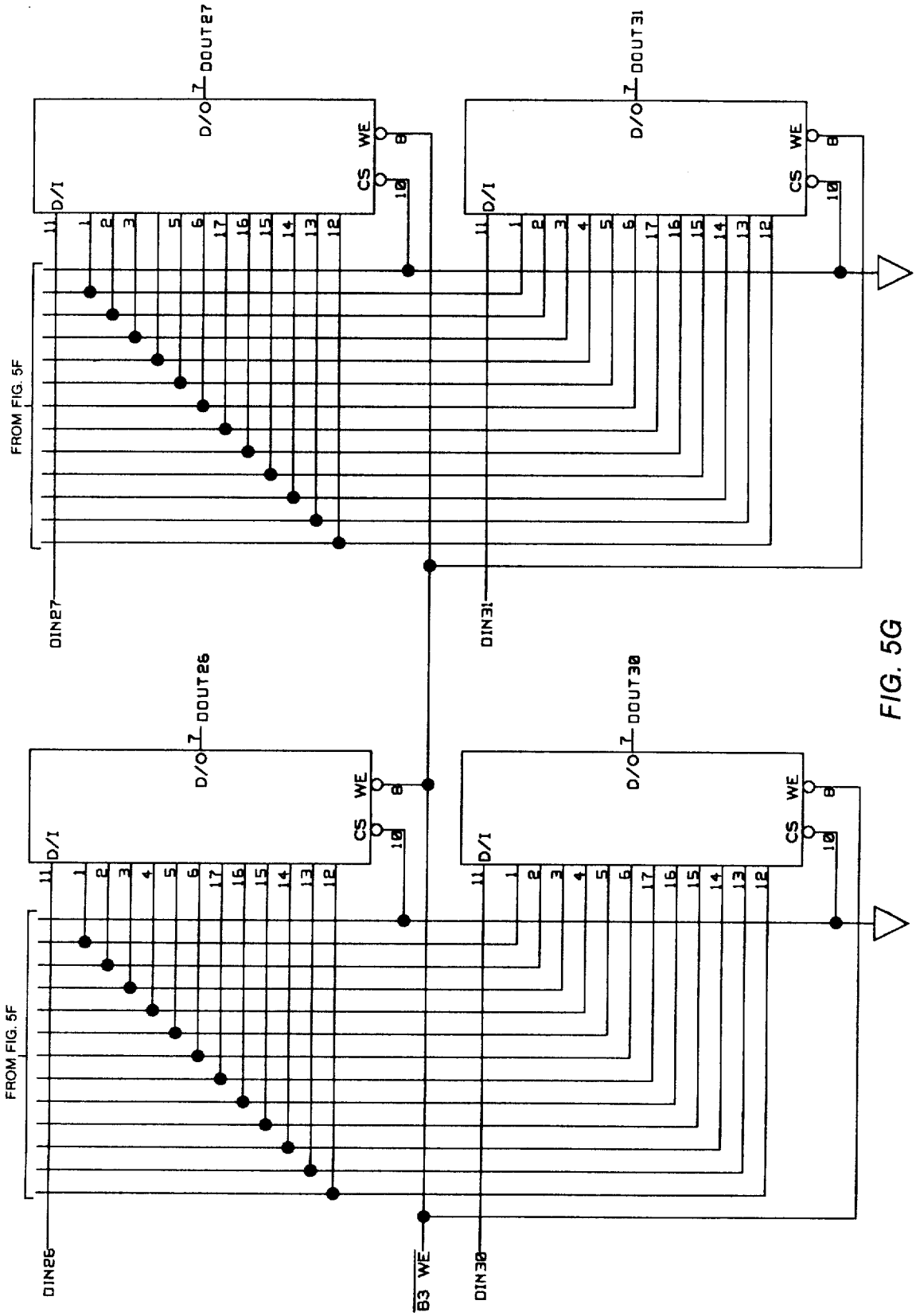
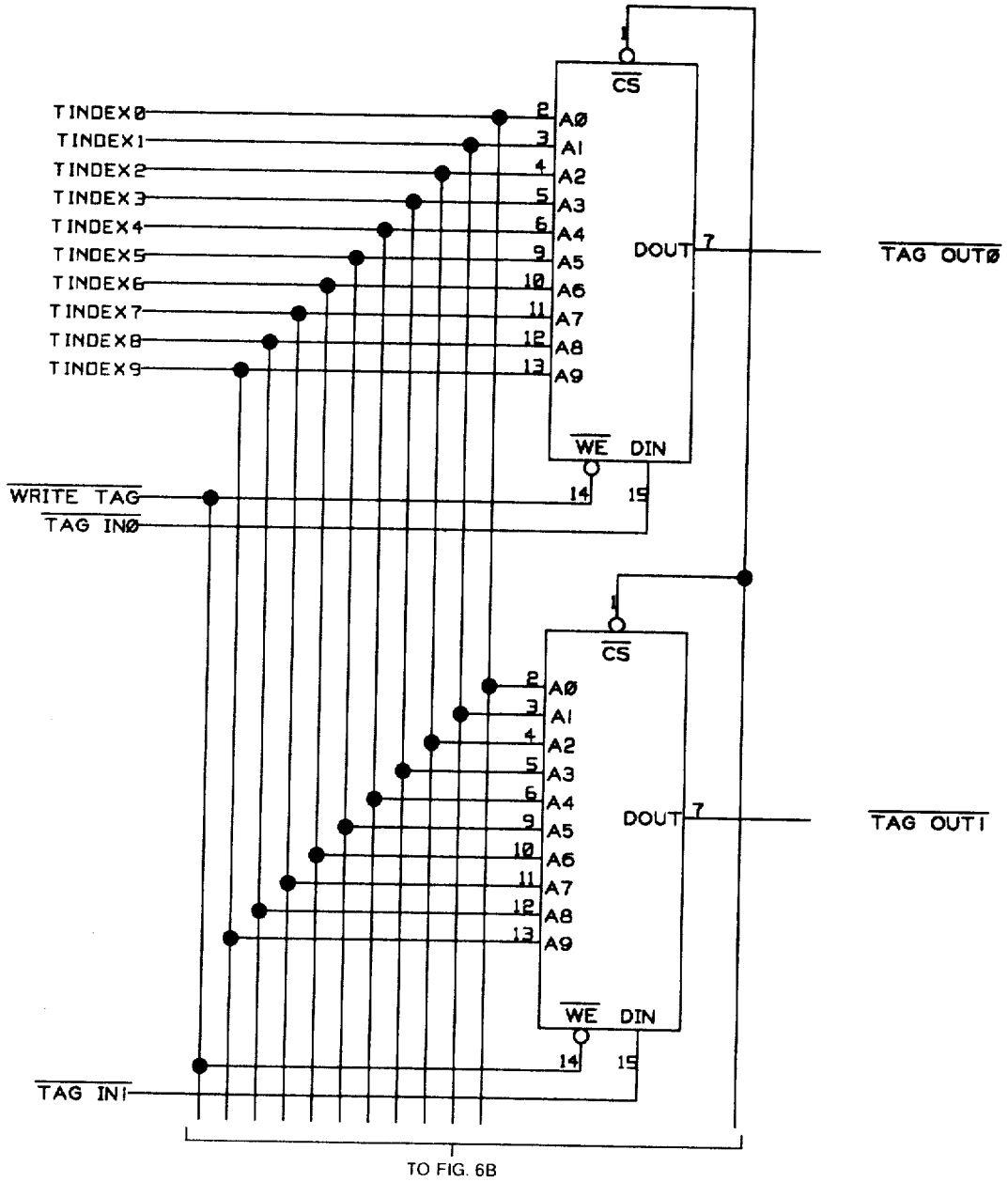

FROM FIG. 5F
DIN27
DIN31
DIN26
DIN30
D/I
D/O
DOUT27
DOUT31
DOUT26
DOUT30
CS
WE
B3 WE

FIG. 5G



TAG STORE 43

FIG. 6

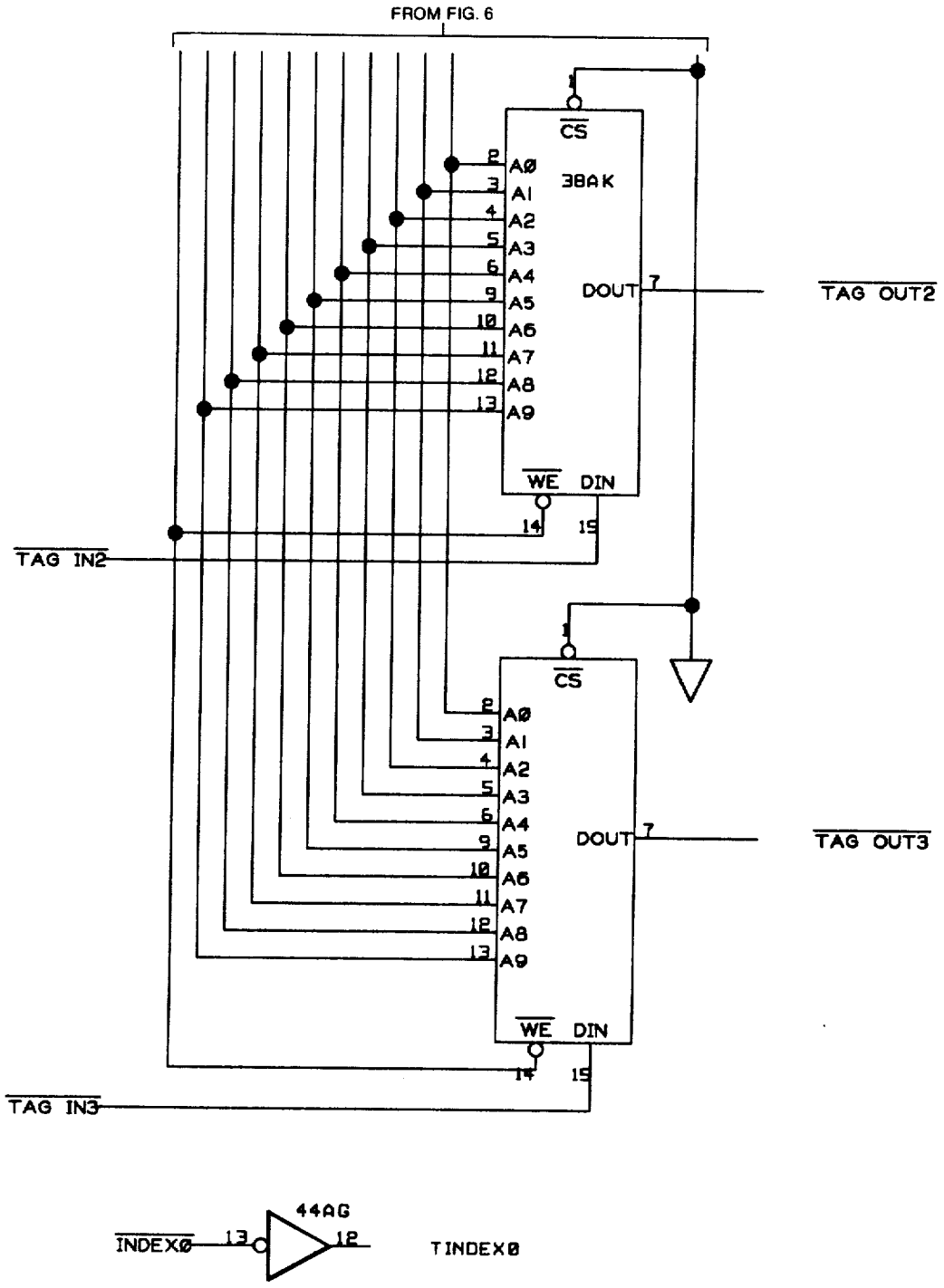

FROM FIG. 6
CS
3BAK
A0
A1
A2
A3
A4
A5
A6
A7
A8
A9
DOUT
TAG OUT2
WE
DIN
TAG IN2
CS
A0
A1
A2
A3
A4
A5
A6
A7
A8
A9
DOUT
TAG OUT3
WE
DIN
TAG IN3
44AG
INDEX0
TINDEX0

FIG. 6A

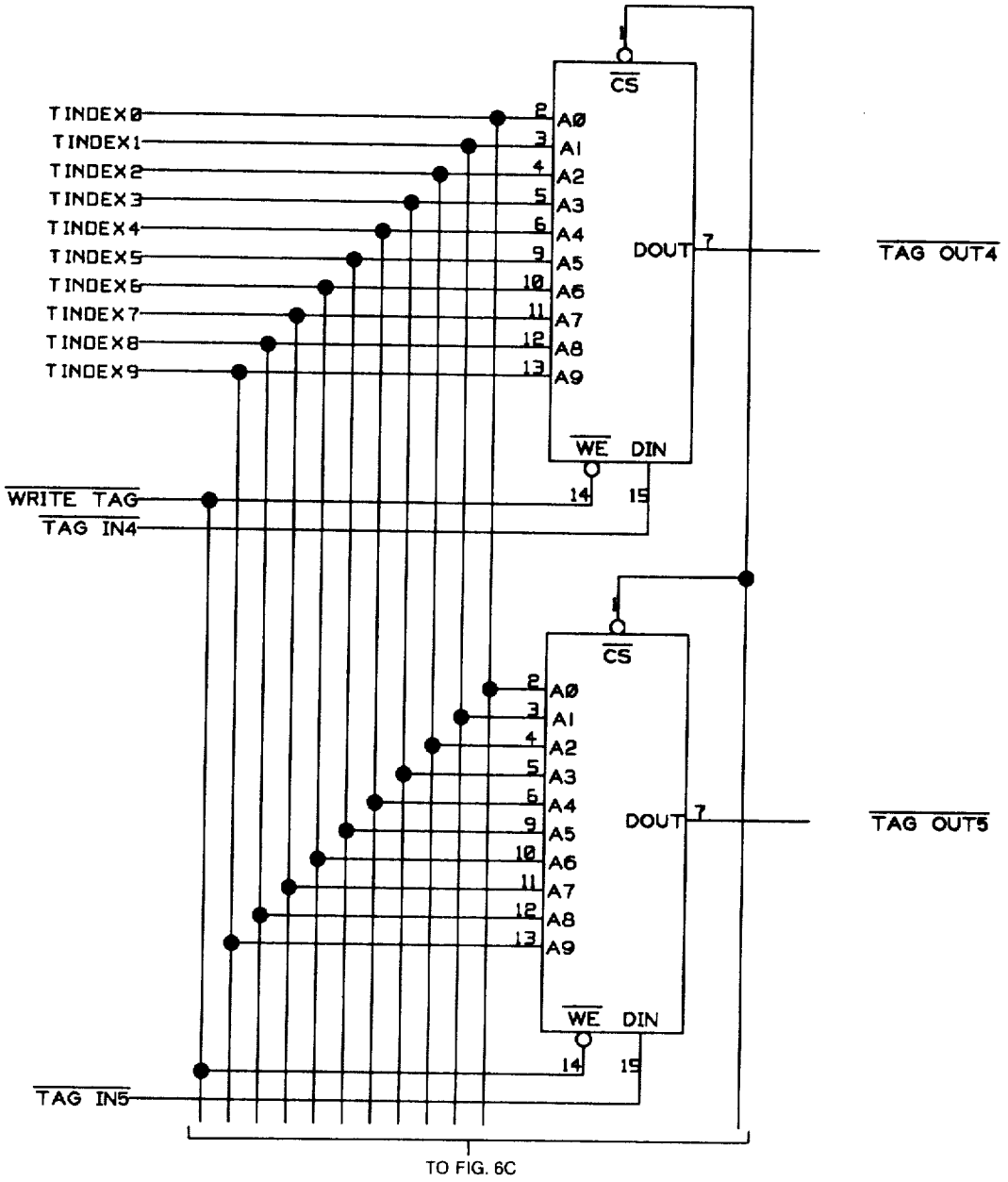

TINDEX0
TINDEX1
TINDEX2
TINDEX3
TINDEX4
TINDEX5
TINDEX6
TINDEX7
TINDEX8
TINDEX9
WRITE TAG
TAG IN4
TAG IN5
CS
A0
A1
A2
A3
A4
A5
A6
A7
A8
A9
DOUT
WE
DIN
TAG OUT4
TAG OUT5
TO FIG. 6C

FIG. 6B

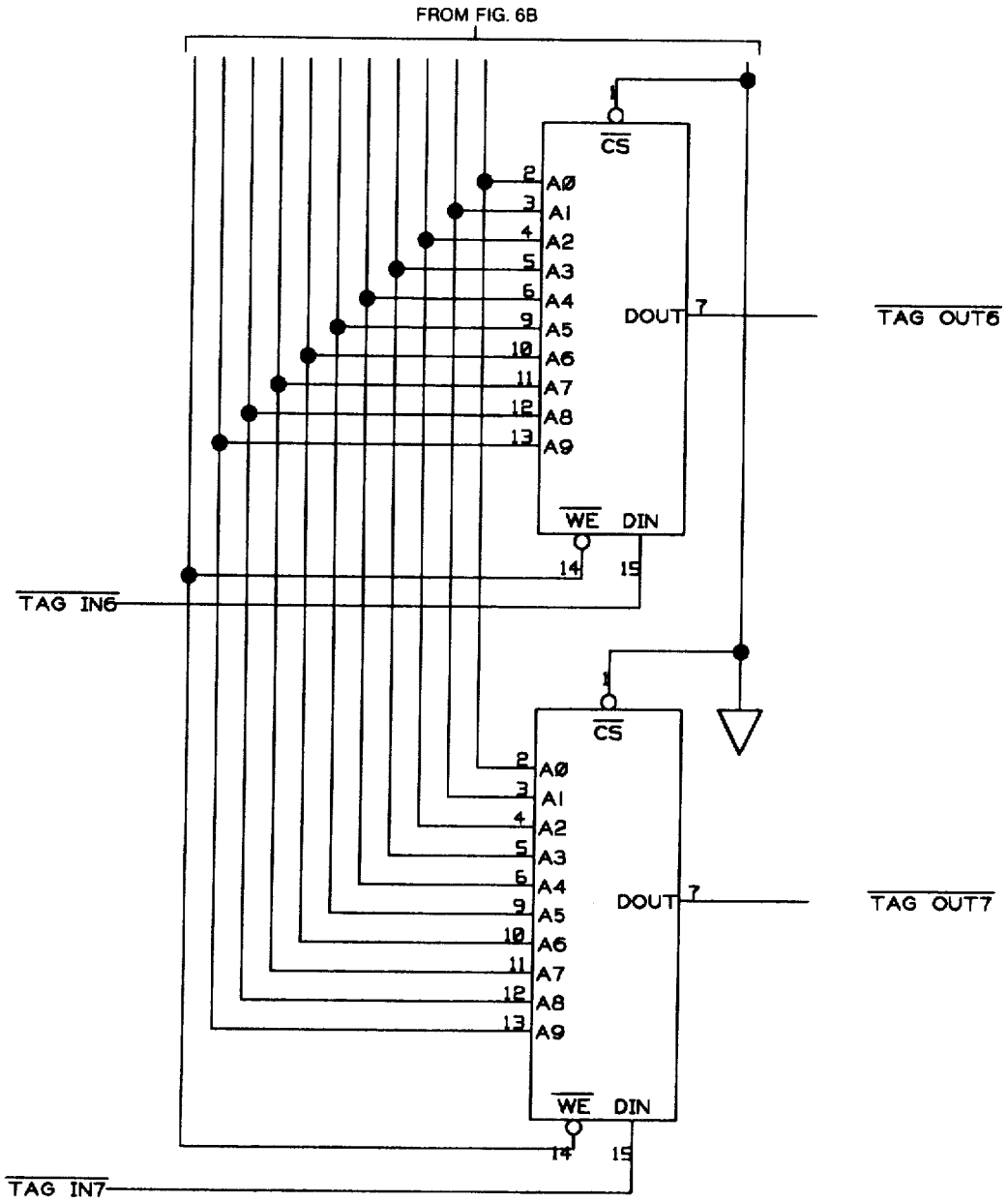

FROM FIG. 6B
CS
A0
A1
A2
A3
A4
A5
A6
A7
A8
A9
DOUT
TAG OUT6
WE
DIN
TAG IN6
CS
A0
A1
A2
A3
A4
A5
A6
A7
A8
A9
DOUT
TAG OUT7
WE
DIN
TAG IN7

FIG. 6C

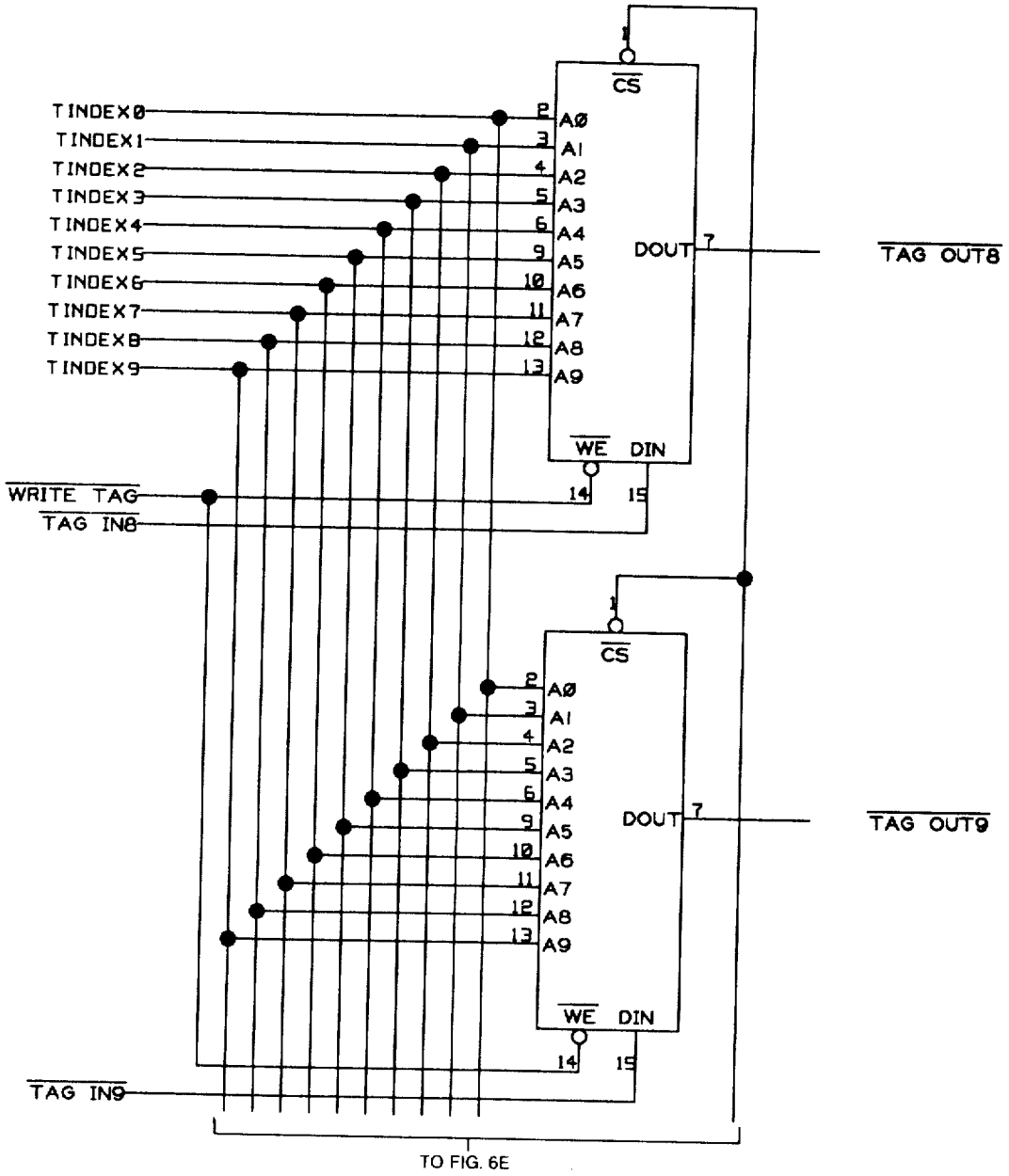

TINDEX0
TINDEX1
TINDEX2
TINDEX3
TINDEX4
TINDEX5
TINDEX6
TINDEX7
TINDEX8
TINDEX9
WRITE TAG
TAG IN8
TAG IN9
CS
A0
A1
A2
A3
A4
A5
A6
A7
A8
A9
DOUT
WE
DIN
TAG OUT8
TAG OUT9
TO FIG. 6E

FIG. 6D

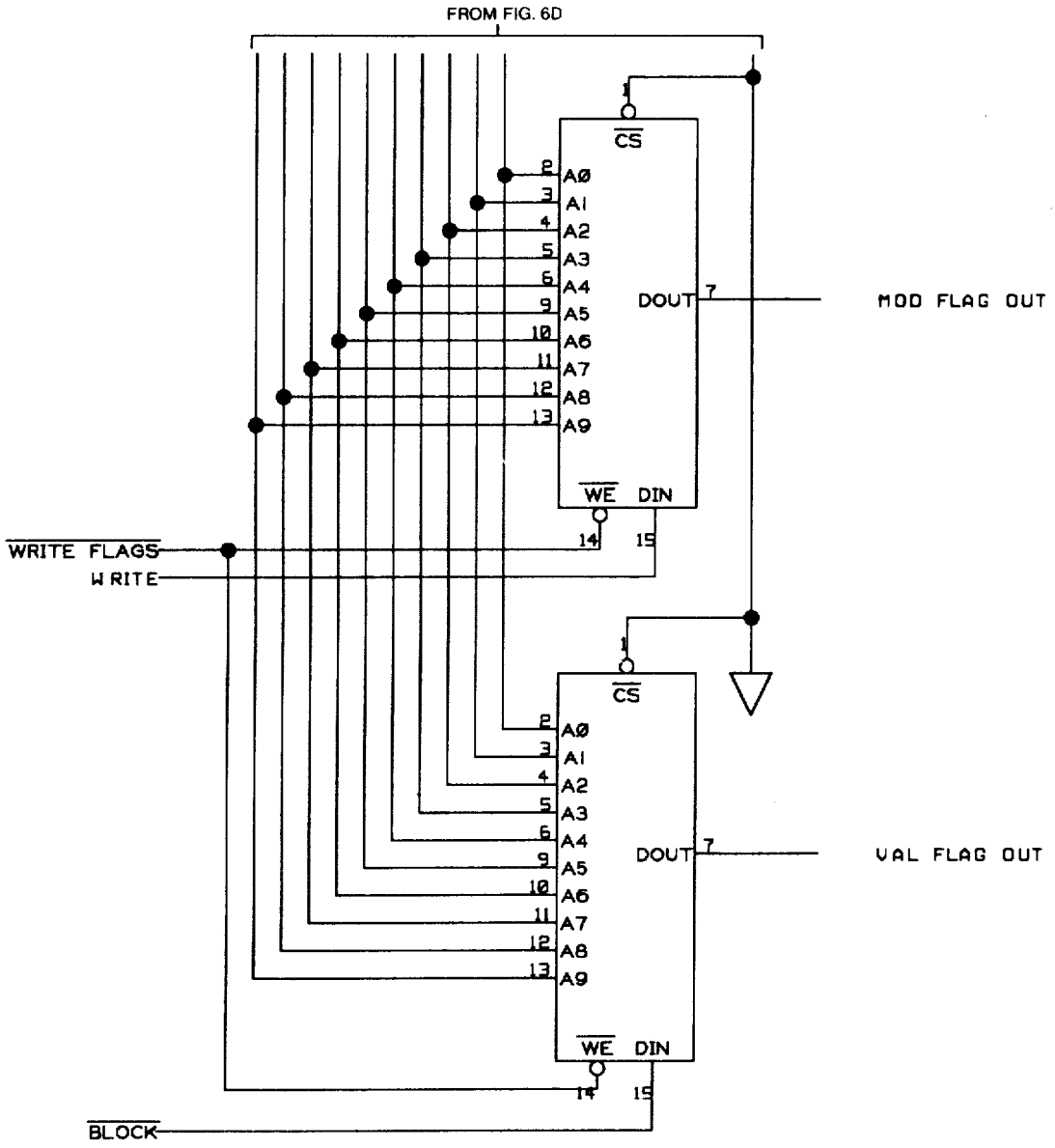
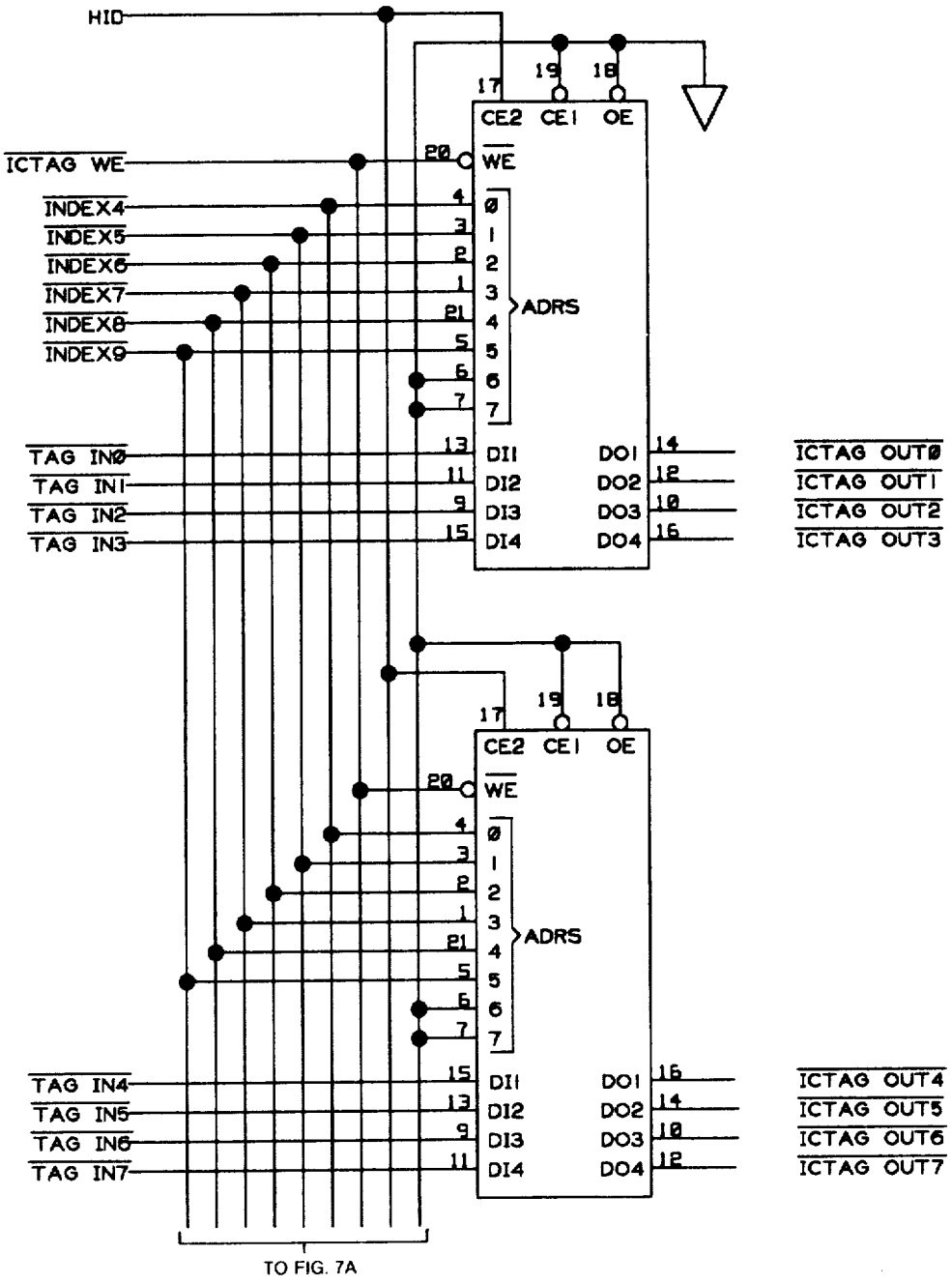

FROM FIG. 6D
CS
A0
A1
A2
A3
A4
A5
A6
A7
A8
A9
DOUT
MOD FLAG OUT
WE
DIN
WRITE FLAGS
WRITE
VAL FLAG OUT
BLOCK

FIG. 6E



ICACHE TAG STORE COPY

FIG. 7

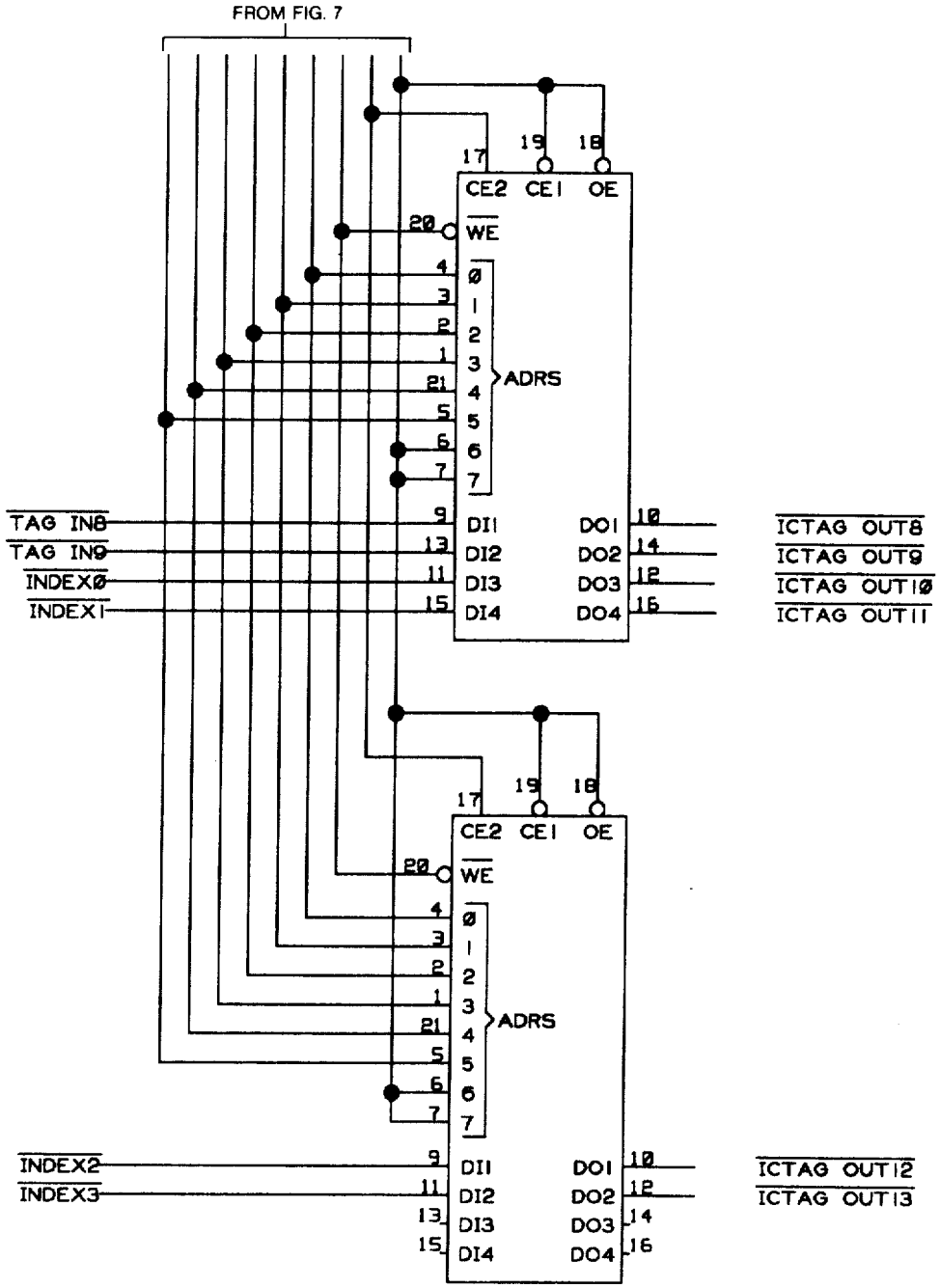
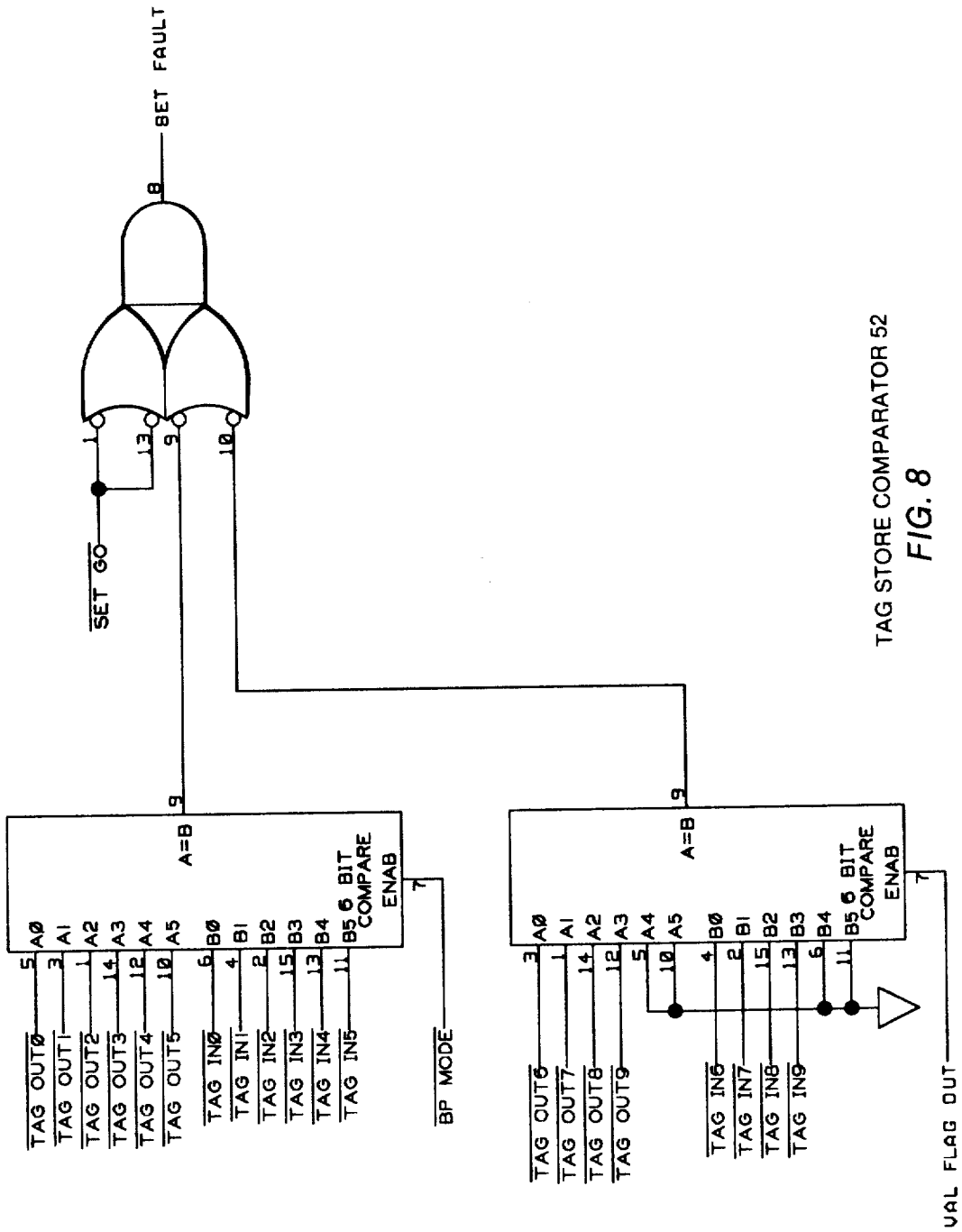
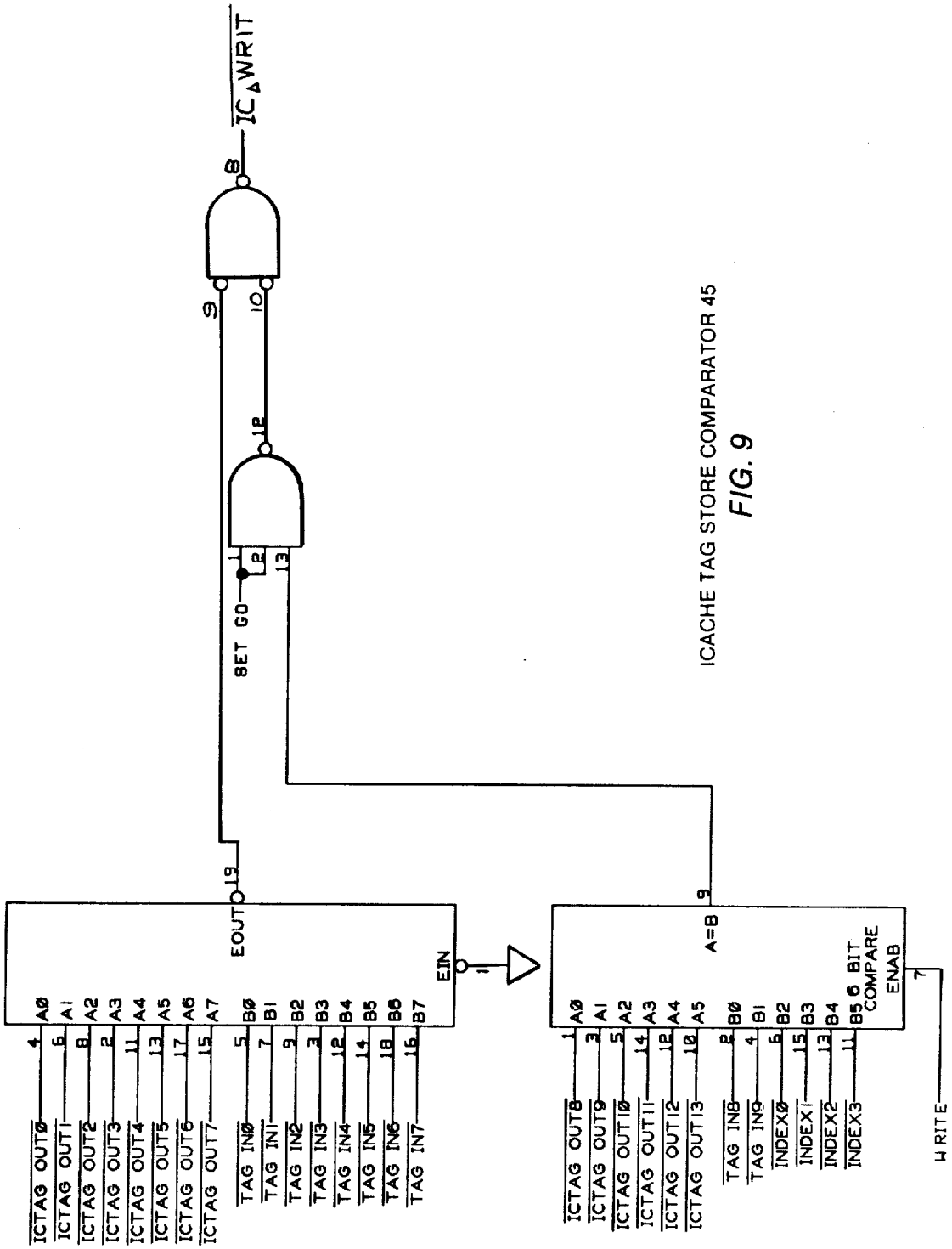

FROM FIG. 7
CE2
CE1
OE
WE
ADRS
DI1
DI2
DI3
DI4
DO1
DO2
DO3
DO4
TAG IN8
TAG IN9
INDEX0
INDEX1
ICTAG OUT8
ICTAG OUT9
ICTAG OUT10
ICTAG OUT11
INDEX2
INDEX3
ICTAG OUT12
ICTAG OUT13

FIG. 7A



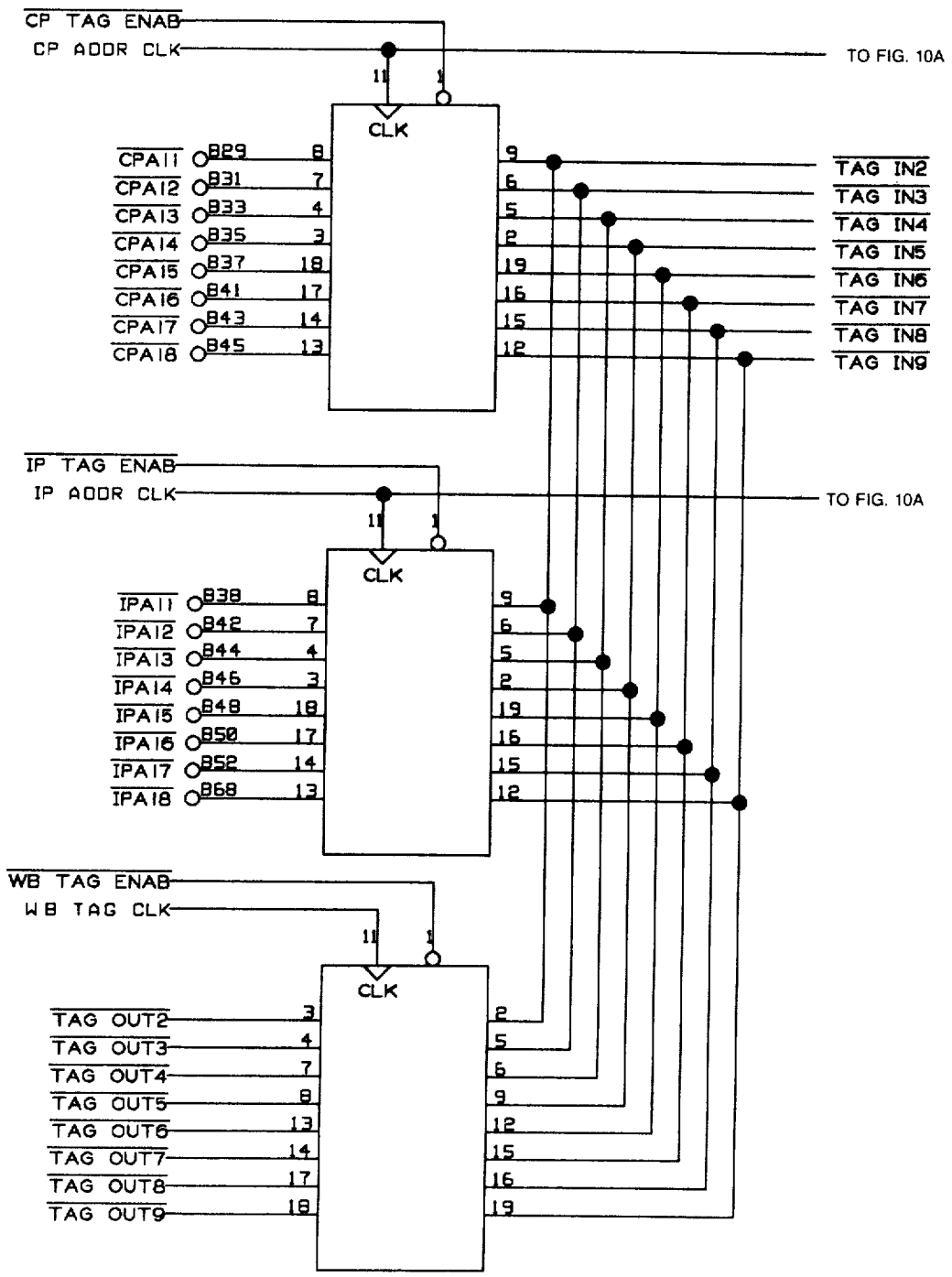
TAG STORE COMPARATOR 52

FIG. 8



ICACHE TAG STORE COMPARATOR 45

FIG. 9



C PORT AND I PORT ADDRESS REGISTERS 41 AND 42
AND WRITE BACK TAG UNIT

FIG. 10

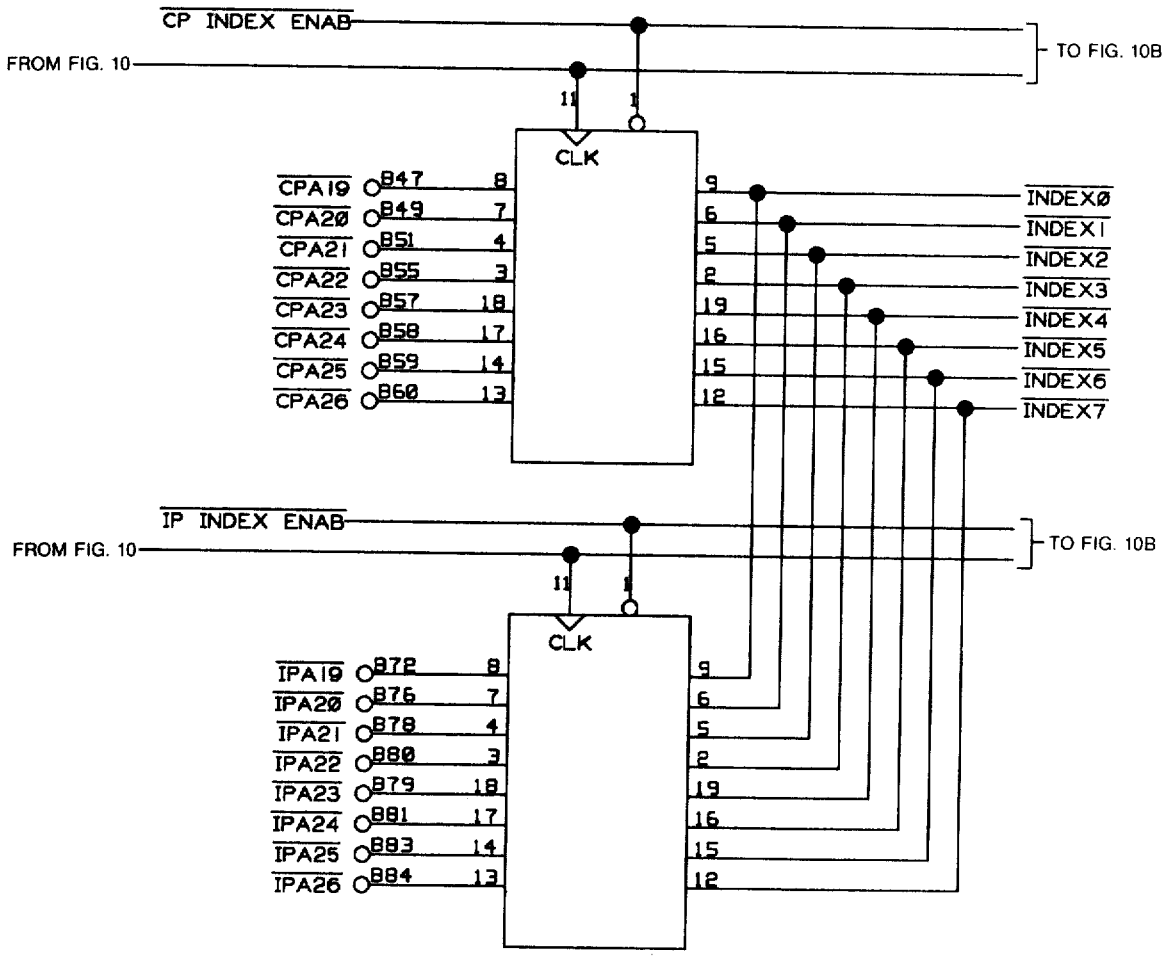

CP INDEX ENAB
FROM FIG. 10
TO FIG. 10B
CLK
CPA19
CPA20
CPA21
CPA22
CPA23
CPA24
CPA25
CPA26
B47
B49
B51
B55
B57
B58
B59
B60
INDEX0
INDEX1
INDEX2
INDEX3
INDEX4
INDEX5
INDEX6
INDEX7
IP INDEX ENAB
FROM FIG. 10
TO FIG. 10B
CLK
IPA19
IPA20
IPA21
IPA22
IPA23
IPA24
IPA25
IPA26
B72
B76
B78
B80
B79
B81
B83
B84

FIG. 10A

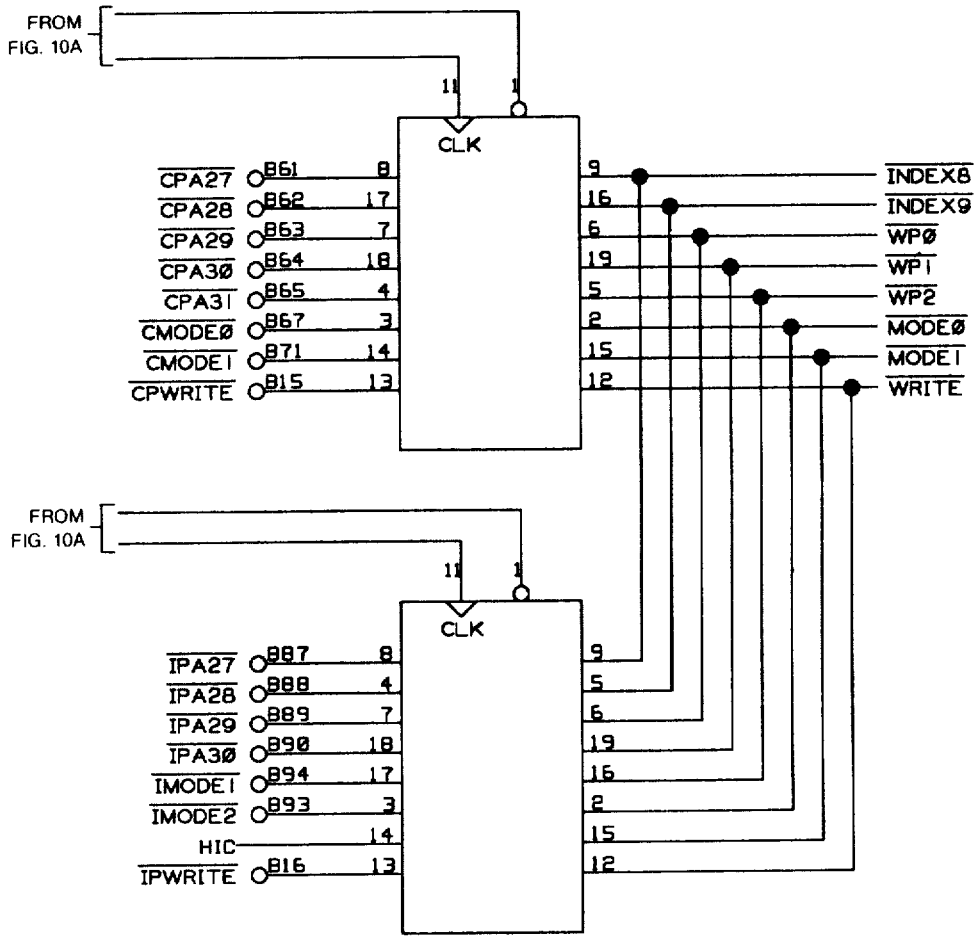
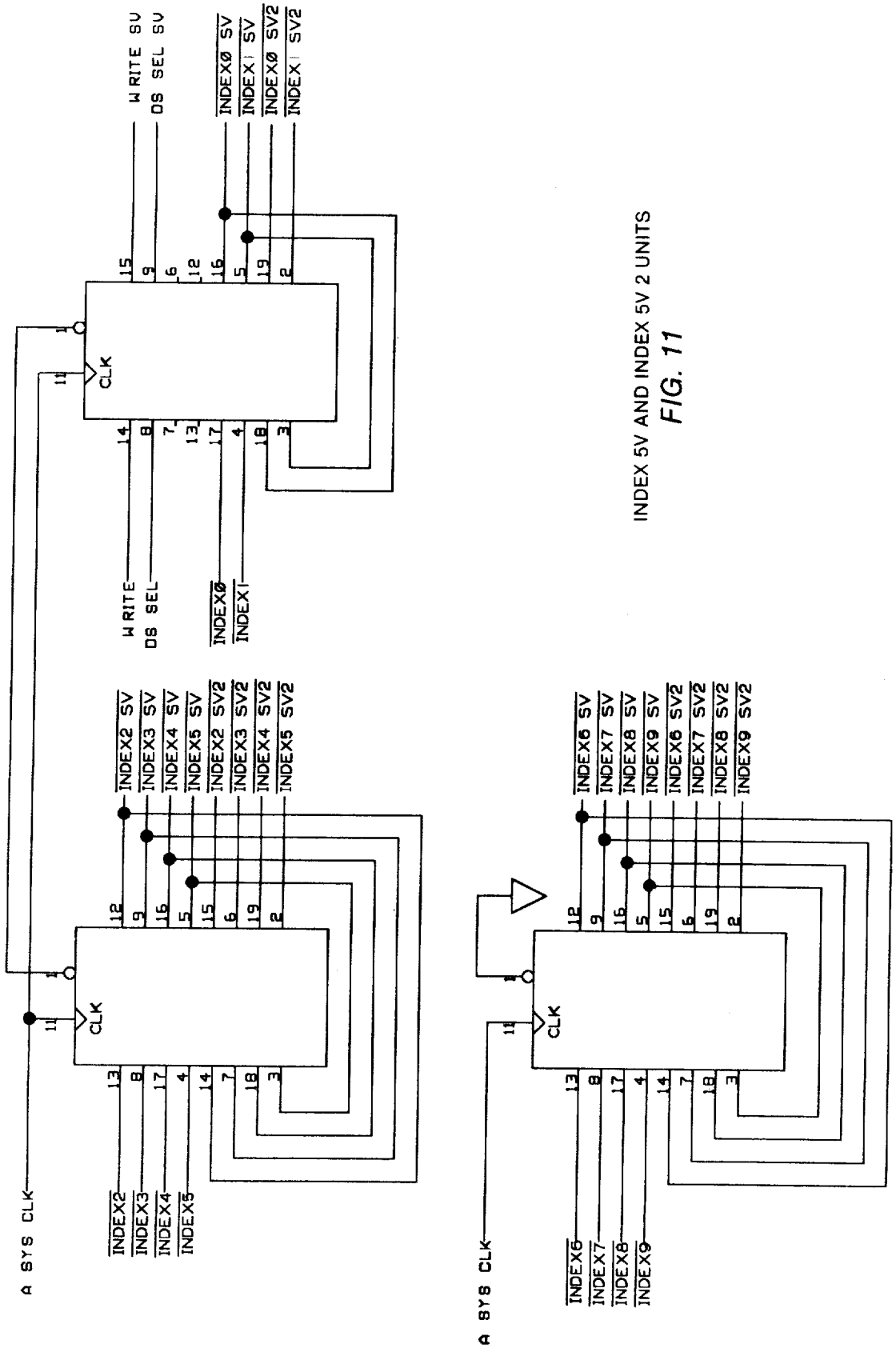
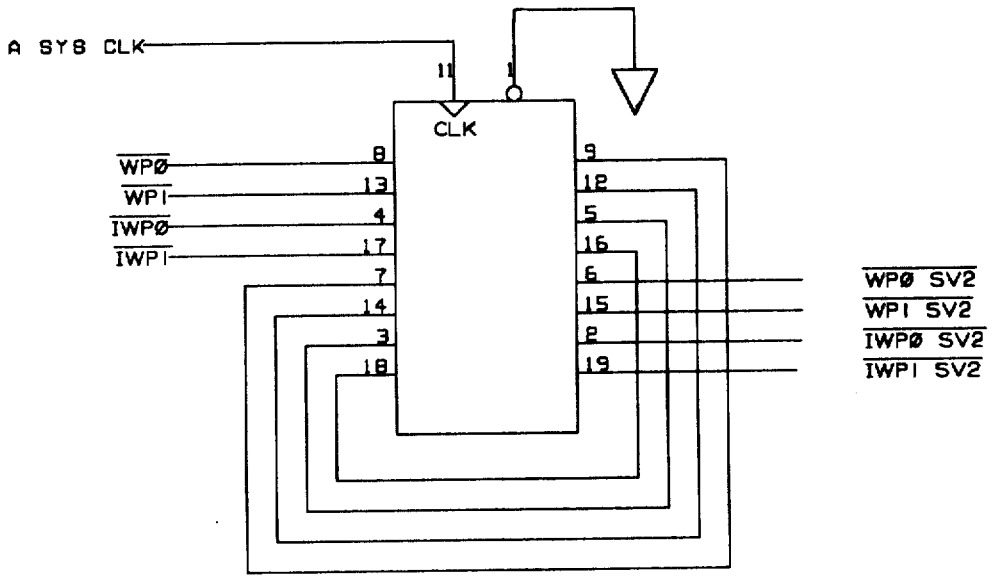

FROM
FIG. 10A
11
1
CLK
CPA27
CPA28
CPA29
CPA30
CPA31
CMODE0
CMODE1
CPWRITE
B61
B62
B63
B64
B65
B67
B71
B15
8
17
7
18
4
3
14
13
9
16
6
19
5
2
15
12
INDEX8
INDEX9
WP0
WP1
WP2
MODE0
MODE1
WRITE
FROM
FIG. 10A
11
1
CLK
IPA27
IPA28
IPA29
IPA30
IMODE1
IMODE2
HIC
IPWRITE
B87
B88
B89
B90
B94
B93
B16
8
4
7
18
17
3
14
13
9
5
6
19
16
2
15
12

FIG. 10B



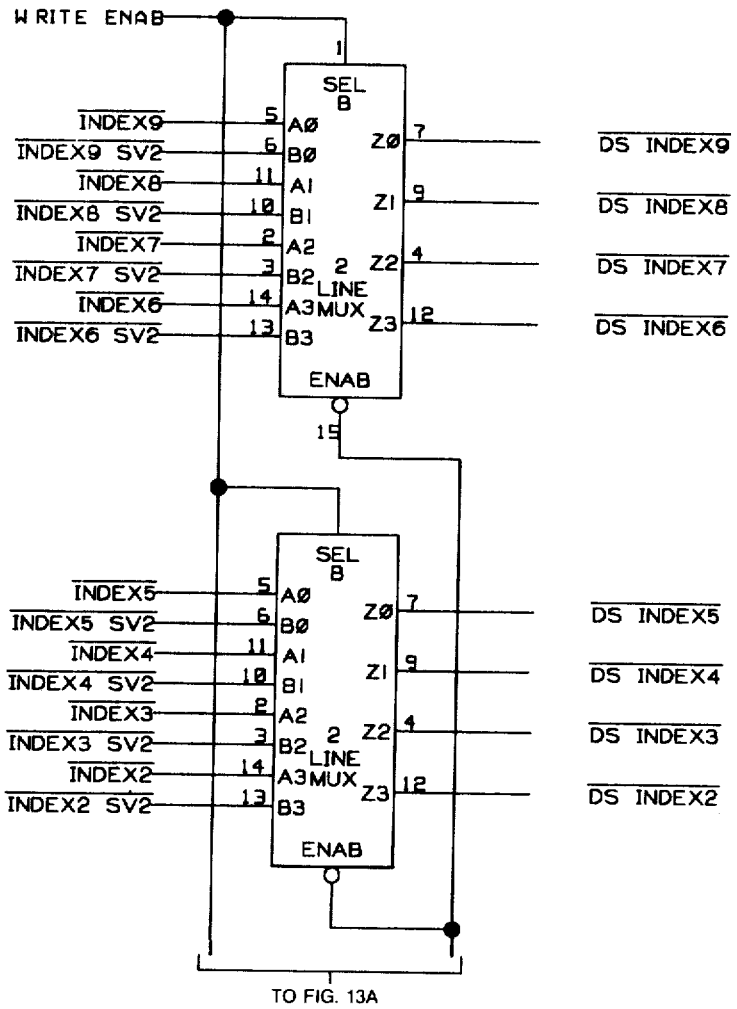
INDEX 5V AND INDEX 5V 2 UNITS

FIG. 11



WP 5V 2 UNIT

FIG. 12



INDEX MUX AND WP MUX

FIG. 13

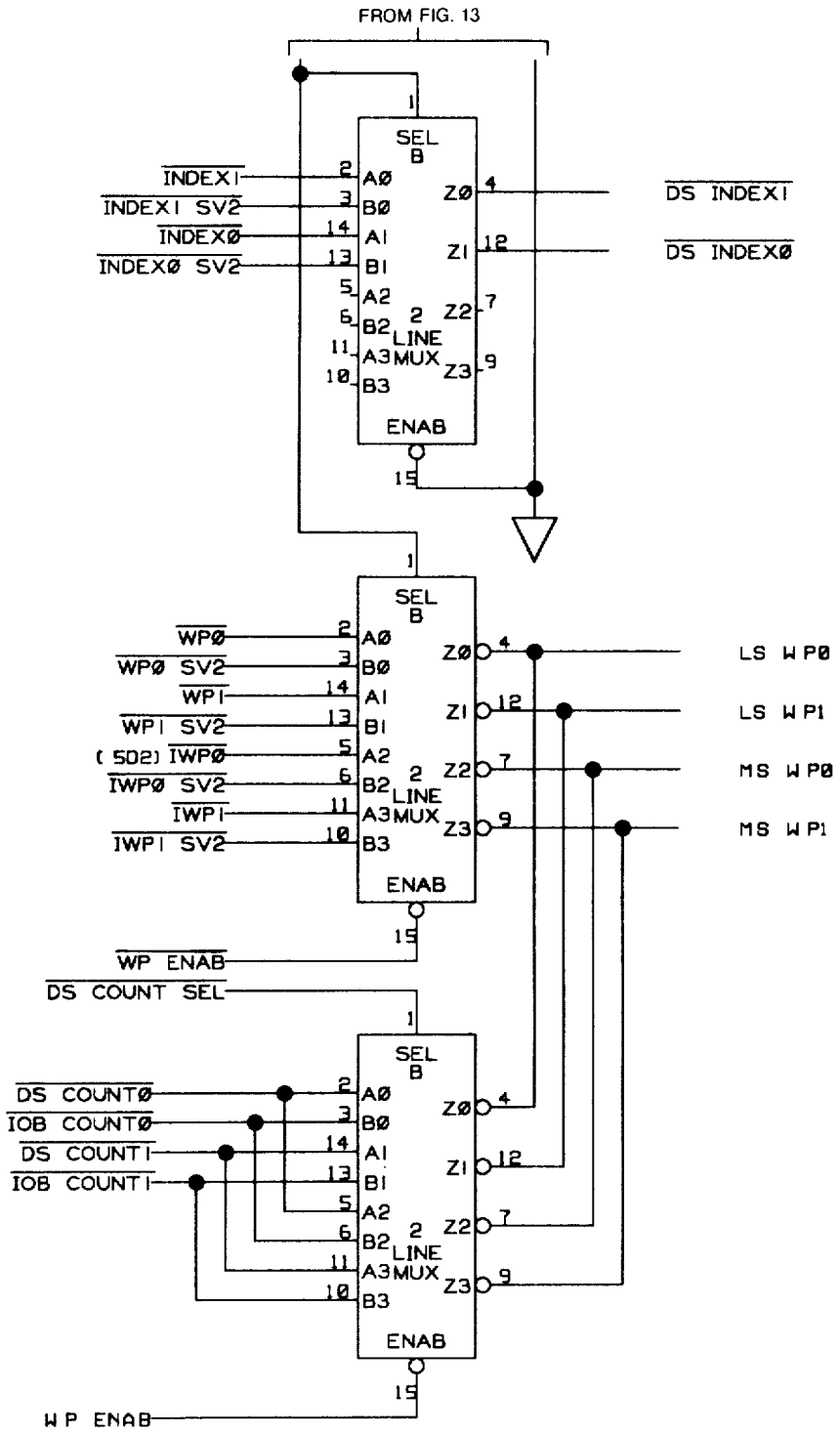
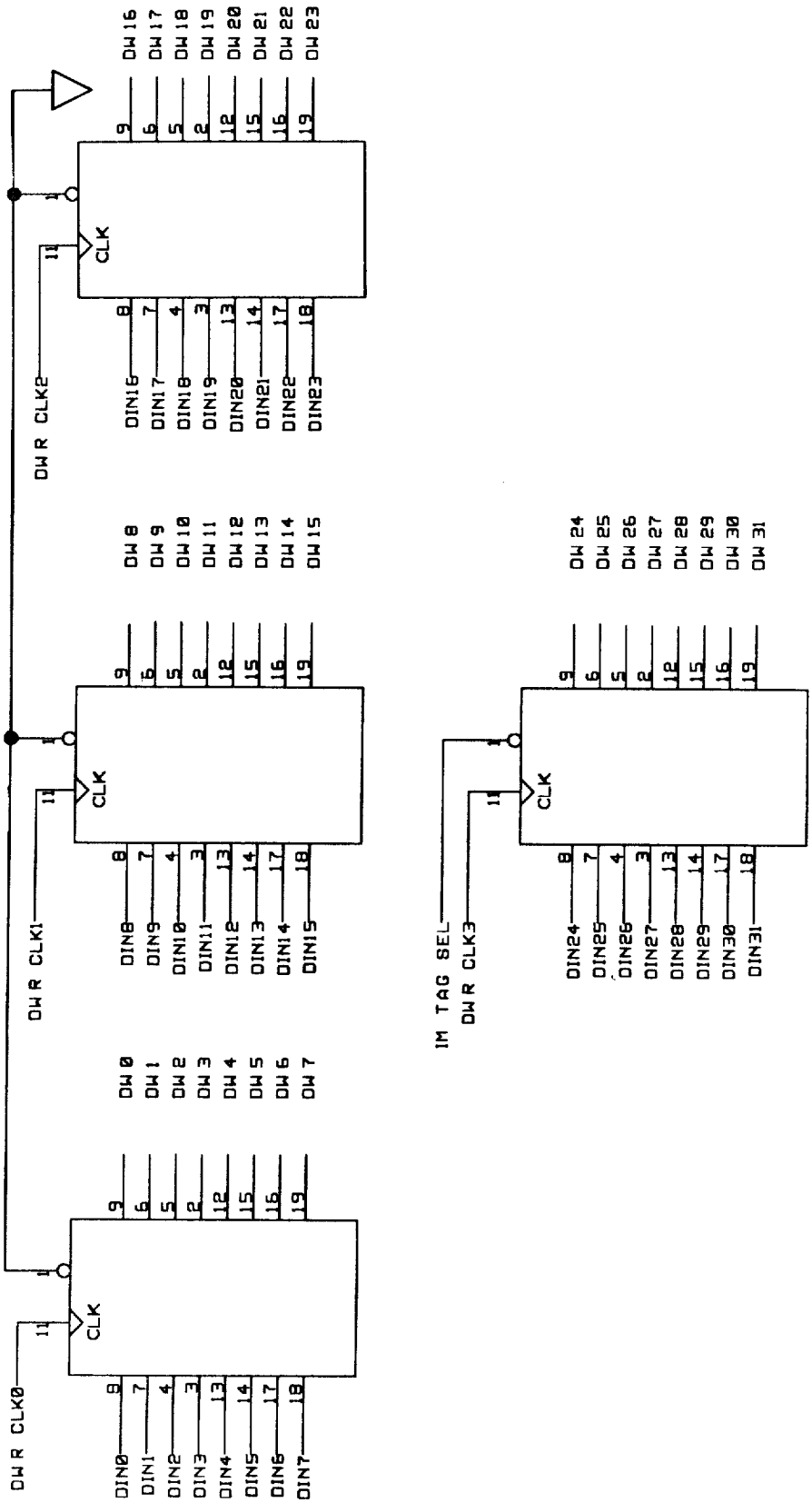
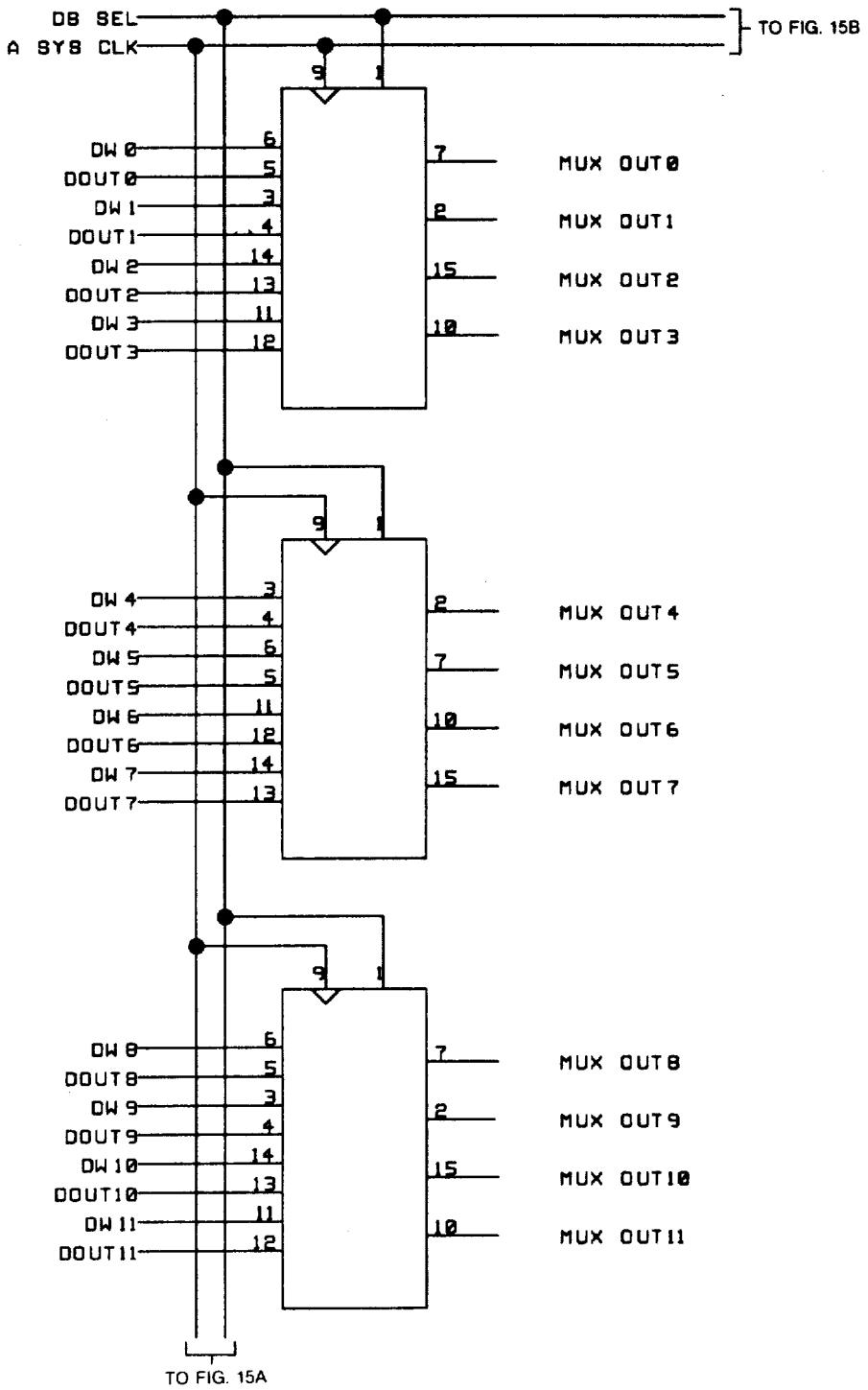

FROM FIG. 13
SEL B
INDEX1
INDEX1 SV2
INDEX0
INDEX0 SV2
A0
B0
A1
B1
A2
B2
A3
B3
Z0
Z1
Z2
Z3
2 LINE MUX
ENAB
DS INDEX1
DS INDEX0
WP0
WP0 SV2
WP1
WP1 SV2
(502) IWP0
IWP0 SV2
IWP1
IWP1 SV2
LS WP0
LS WP1
MS WP0
MS WP1
WP ENAB
DS COUNT SEL
DS COUNT0
IOB COUNT0
DS COUNT1
IOB COUNT1
WP ENAB

FIG. 13A



DATA WRITE REGISTER 40A

FIG. 14



MULTIPLEXER 48 AND INDEX DRIVER 48'

FIG. 15

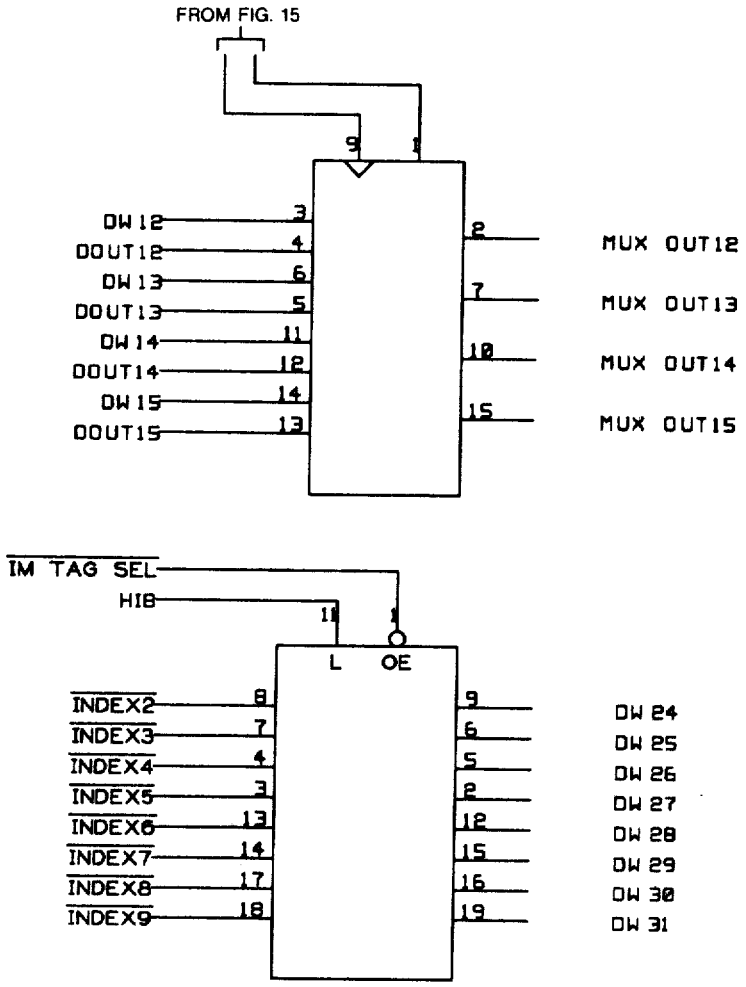

FROM FIG. 15
9
1
DW 12
DOUT12
DW 13
DOUT13
DW 14
DOUT14
DW 15
DOUT15
3
4
6
5
11
12
14
13
2
7
10
15
MUX OUT12
MUX OUT13
MUX OUT14
MUX OUT15
IM TAG SEL
HIB
11
1
L
OE
INDEX2
INDEX3
INDEX4
INDEX5
INDEX6
INDEX7
INDEX8
INDEX9
8
7
4
3
13
14
17
18
9
6
5
2
12
15
16
19
DW 24
DW 25
DW 26
DW 27
DW 28
DW 29
DW 30
DW 31

FIG. 15A

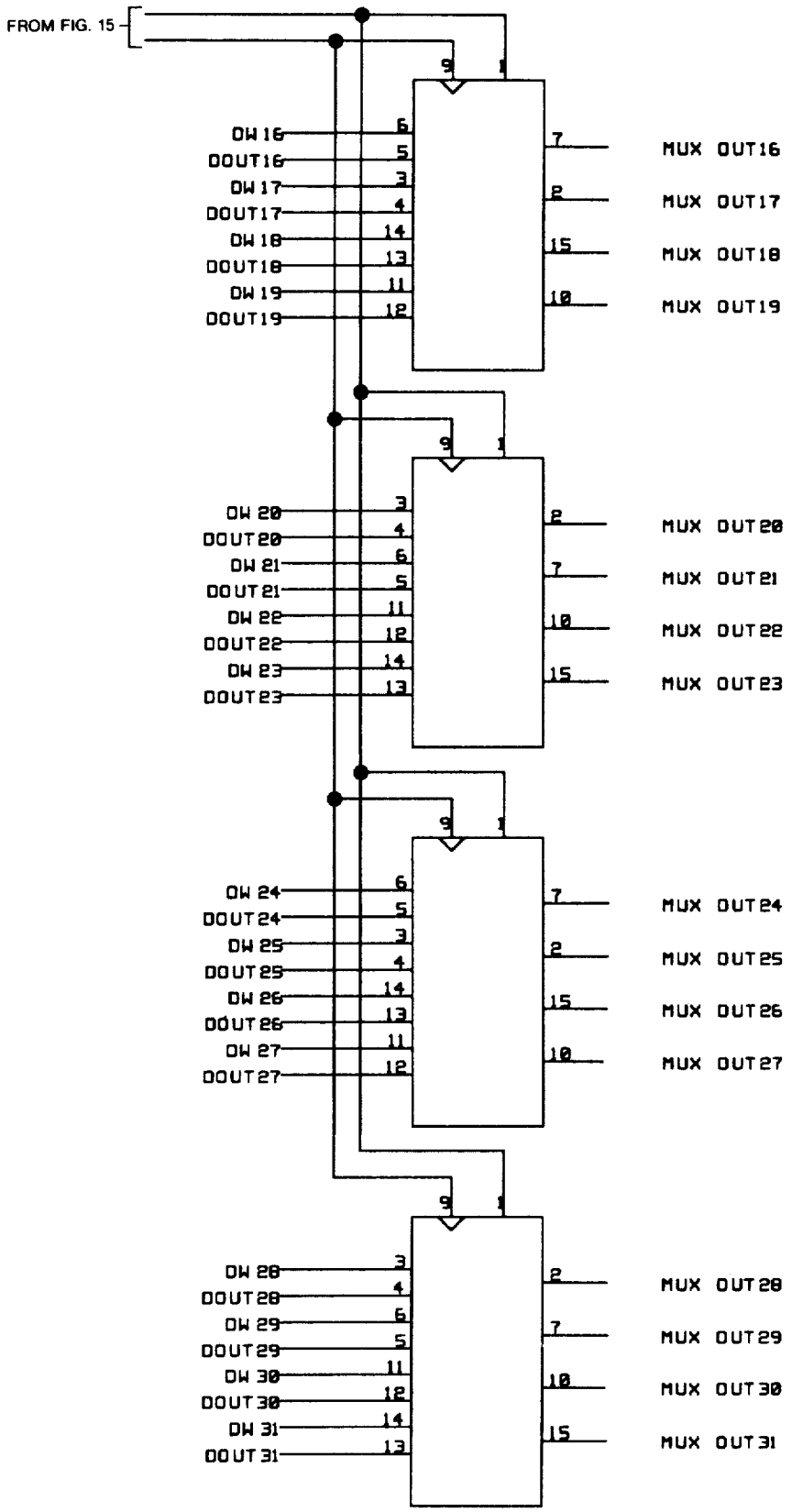
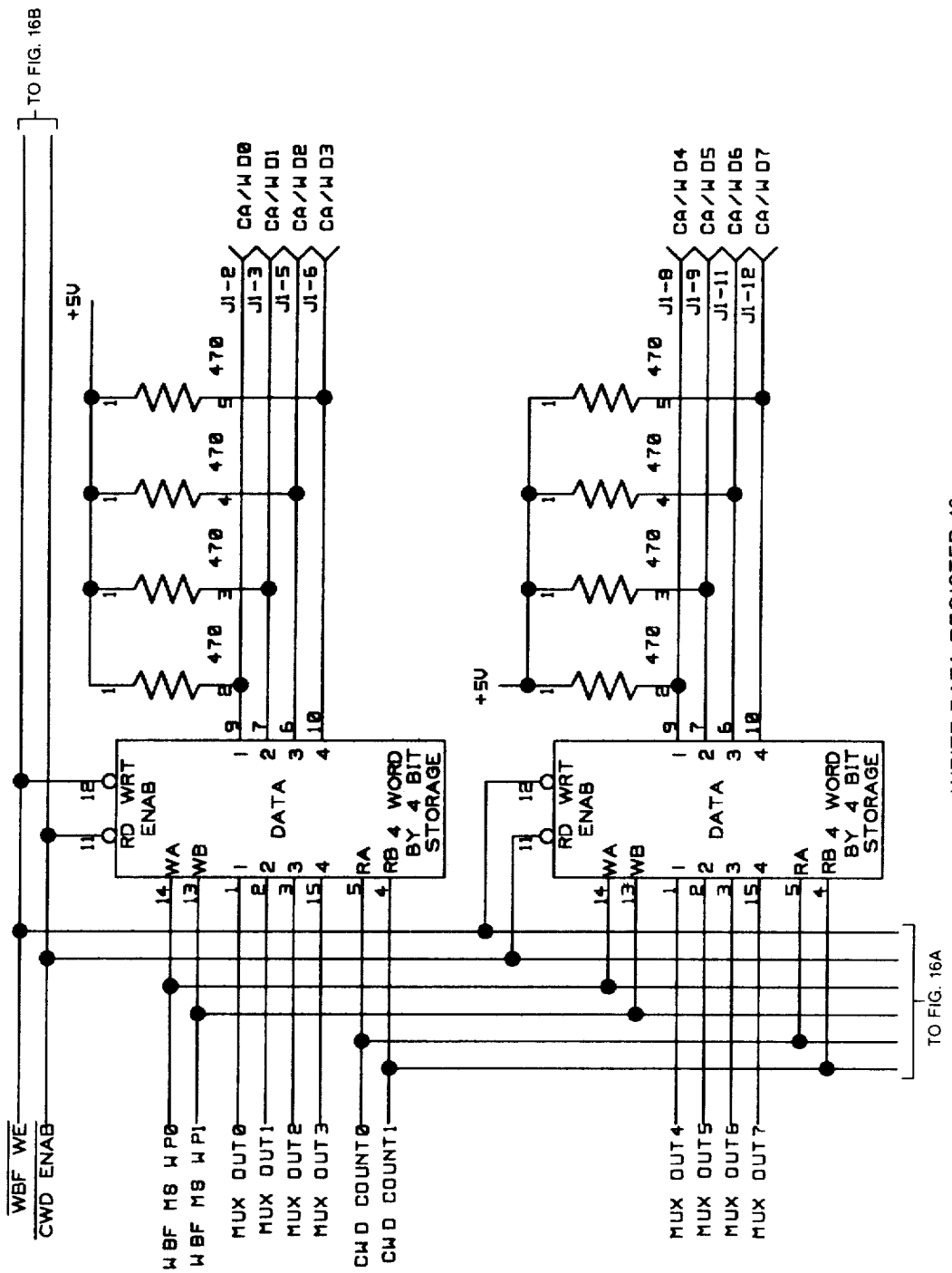

FROM FIG. 15
DW 16
DOUT16
DW 17
DOUT17
DW 18
DOUT18
DW 19
DOUT19
MUX OUT16
MUX OUT17
MUX OUT18
MUX OUT19
DW 20
DOUT20
DW 21
DOUT21
DW 22
DOUT22
DW 23
DOUT23
MUX OUT20
MUX OUT21
MUX OUT22
MUX OUT23
DW 24
DOUT24
DW 25
DOUT25
DW 26
DOUT26
DW 27
DOUT27
MUX OUT24
MUX OUT25
MUX OUT26
MUX OUT27
DW 28
DOUT28
DW 29
DOUT29
DW 30
DOUT30
DW 31
DOUT31
MUX OUT28
MUX OUT29
MUX OUT30
MUX OUT31

FIG. 15B



WRITE DATA REGISTER 46

FIG. 16

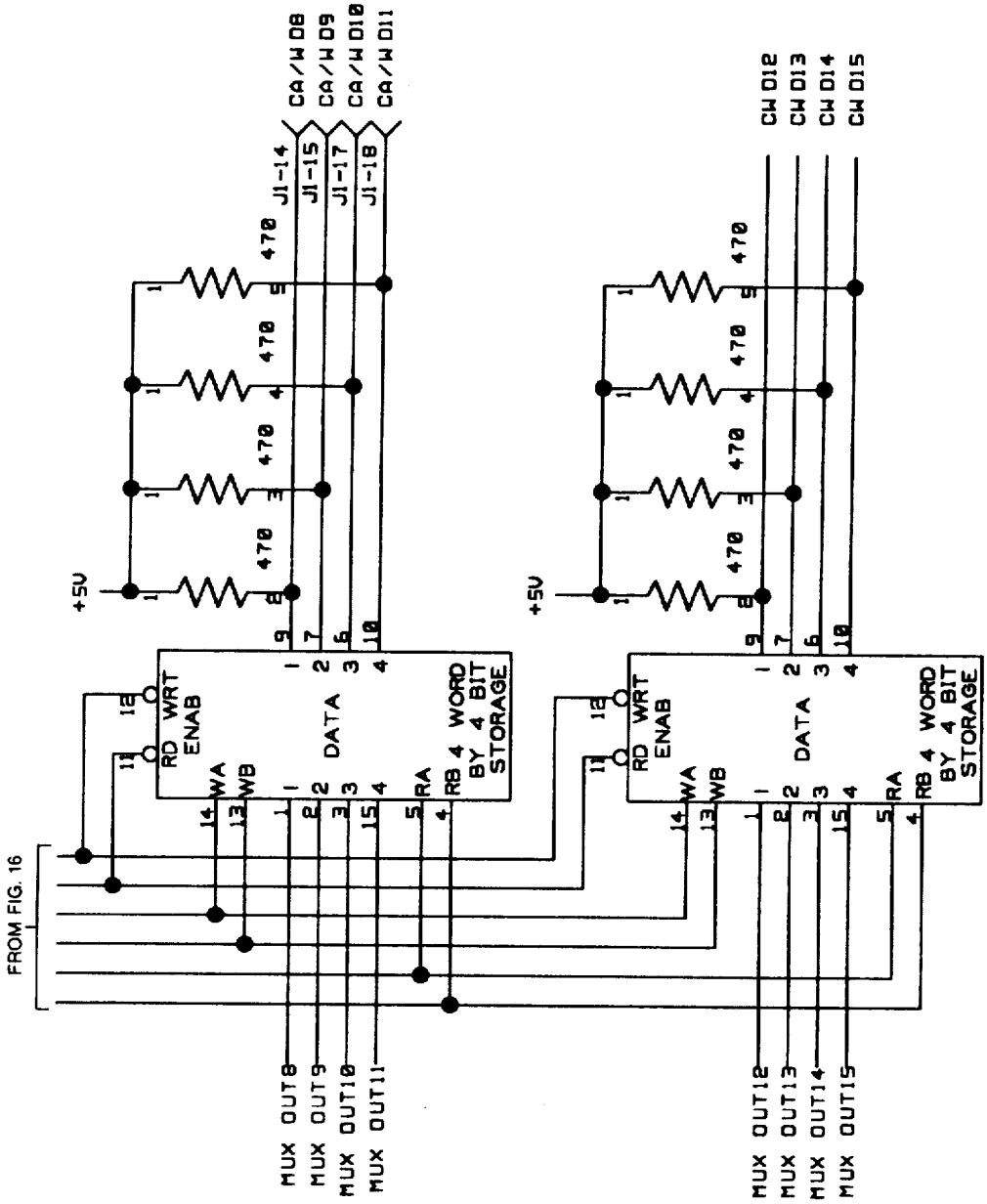

FROM FIG. 16
+5V
470
470
470
470
J1-14
J1-15
J1-17
J1-18
CA/W D8
CA/W D9
CA/W D10
CA/W D11
RD WRT
ENAB
WA
WB
DATA
RA
RB
4 WORD
BY 4 BIT
STORAGE
MUX OUT8
MUX OUT9
MUX OUT10
MUX OUT11
+5V
470
470
470
470
CW D12
CW D13
CW D14
CW D15
RD WRT
ENAB
WA
WB
DATA
RA
RB
4 WORD
BY 4 BIT
STORAGE
MUX OUT12
MUX OUT13
MUX OUT14
MUX OUT15

FIG. 16A

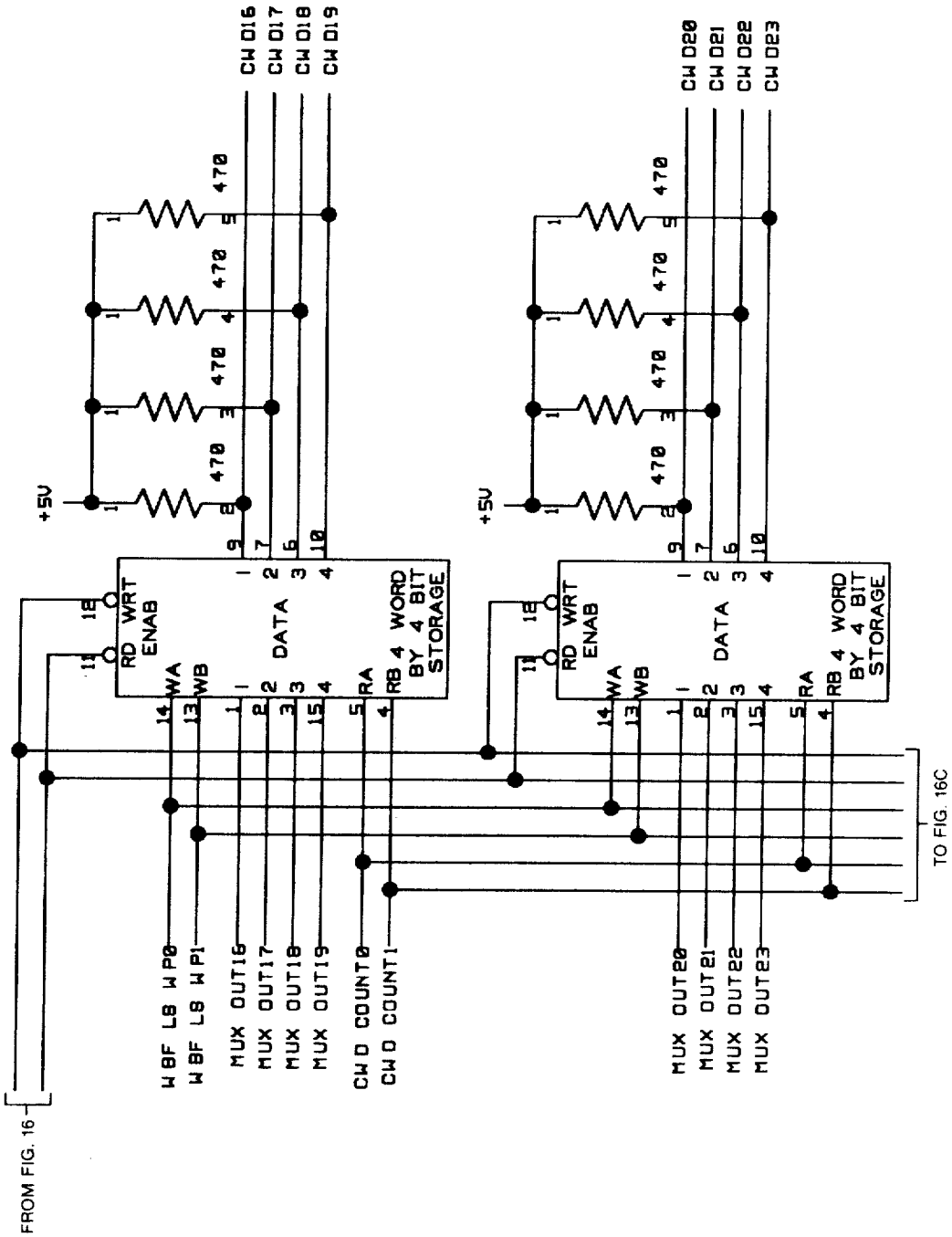

FROM FIG. 16
W BF LS W P0
W BF LS W P1
MUX OUT16
MUX OUT17
MUX OUT18
MUX OUT19
CW D COUNT0
CW D COUNT1
MUX OUT20
MUX OUT21
MUX OUT22
MUX OUT23
TO FIG. 16C
+5V
470
CW D16
CW D17
CW D18
CW D19
CW D20
CW D21
CW D22
CW D23
RD WRT ENAB
WA
WB
DATA
RA
RB
4 WORD BY 4 BIT STORAGE

FIG. 16B

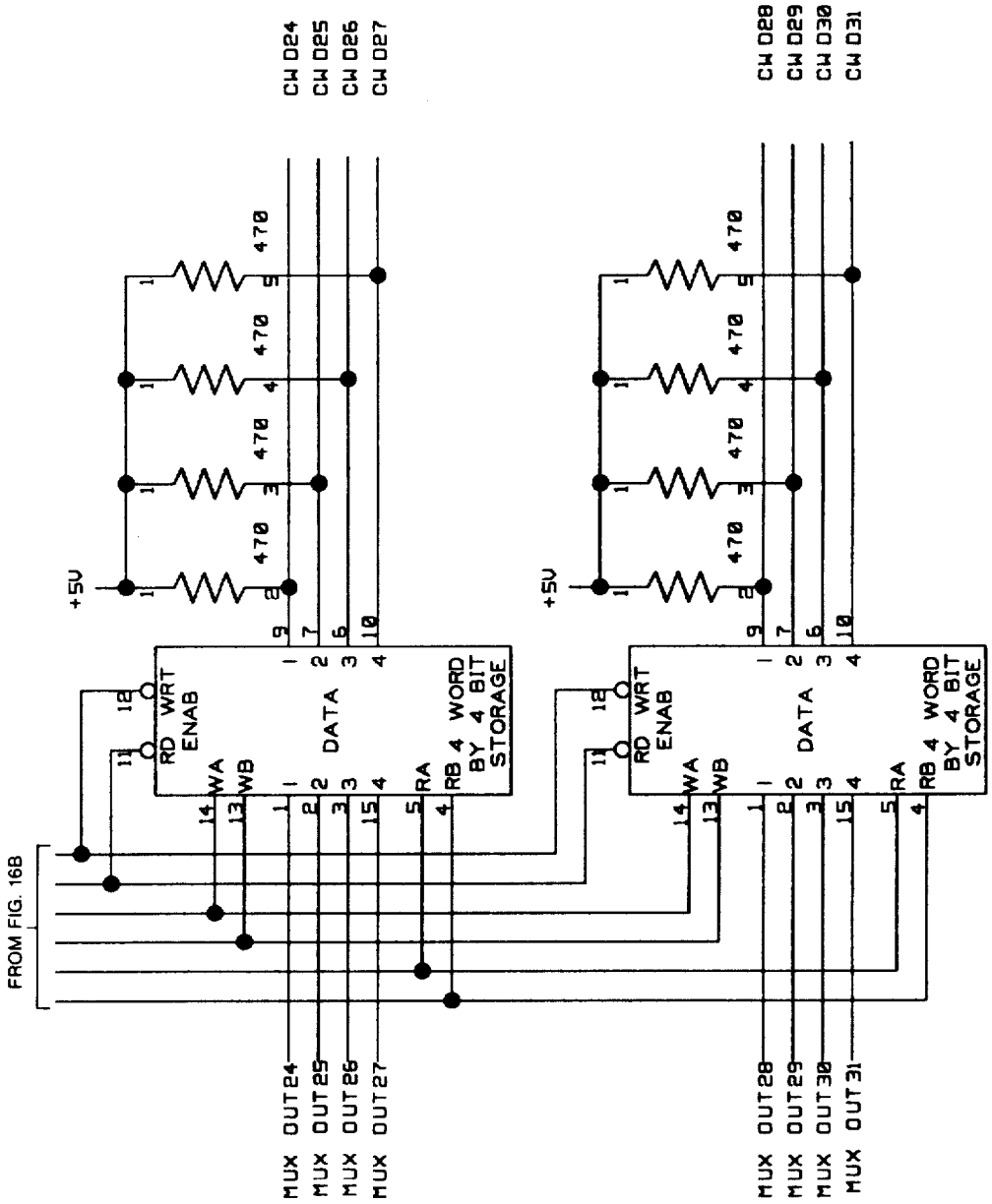
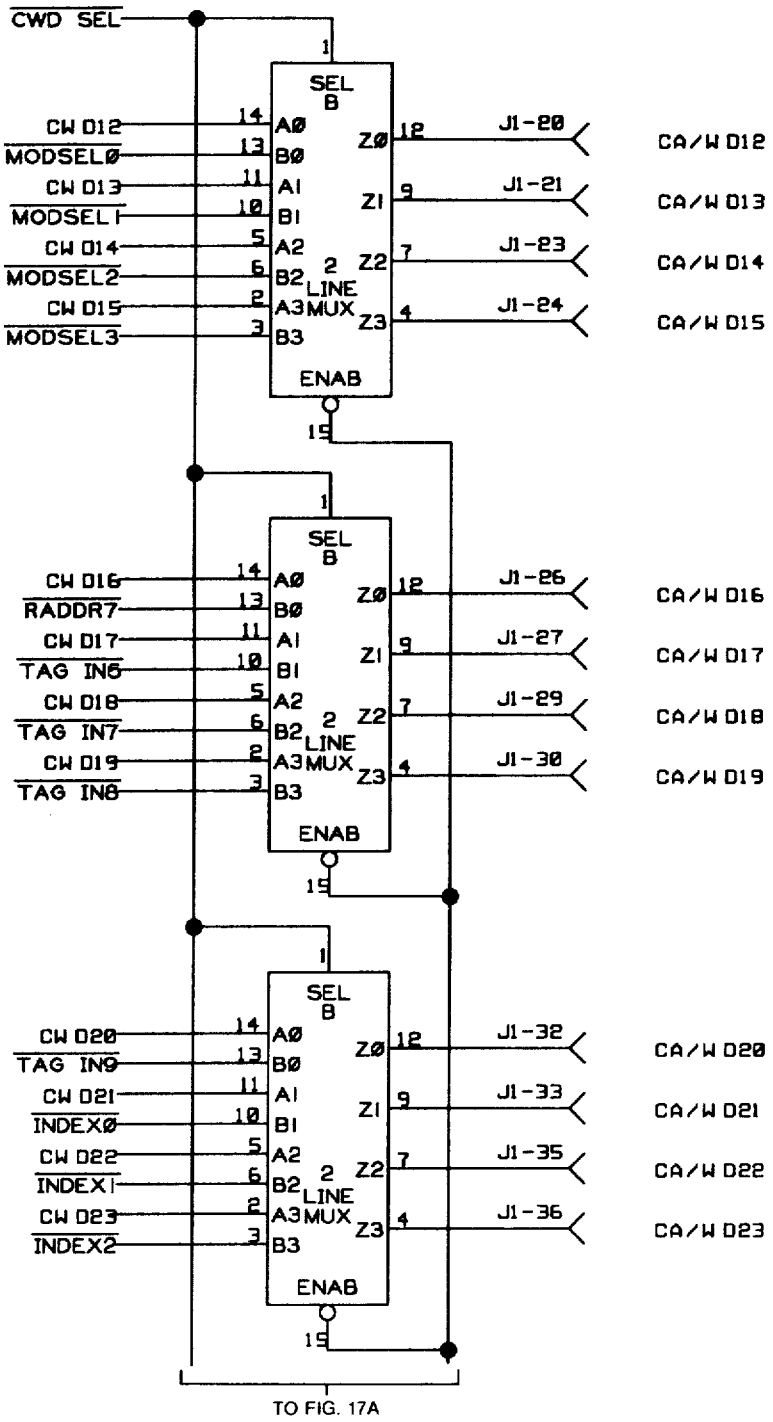

FROM FIG. 16B
+5V
470
470
470
470
RD WRT ENAB
WA
WB
DATA
RA
RB 4 WORD BY 4 BIT STORAGE
MUX OUT24
MUX OUT25
MUX OUT26
MUX OUT27
CW D24
CW D25
CW D26
CW D27
+5V
470
470
470
470
RD WRT ENAB
WA
WB
DATA
RA
RB 4 WORD BY 4 BIT STORAGE
MUX OUT28
MUX OUT29
MUX OUT30
MUX OUT31
CW D28
CW D29
CW D30
CW D31

FIG. 16C



MULTIPLEXOR 47

FIG. 17

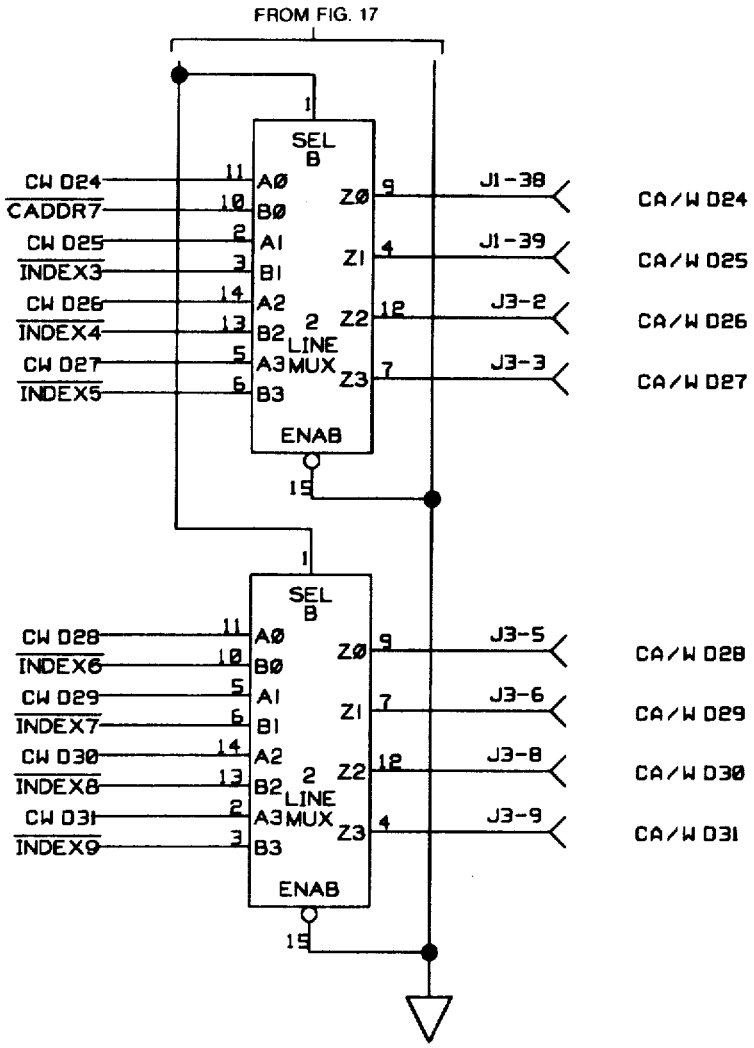
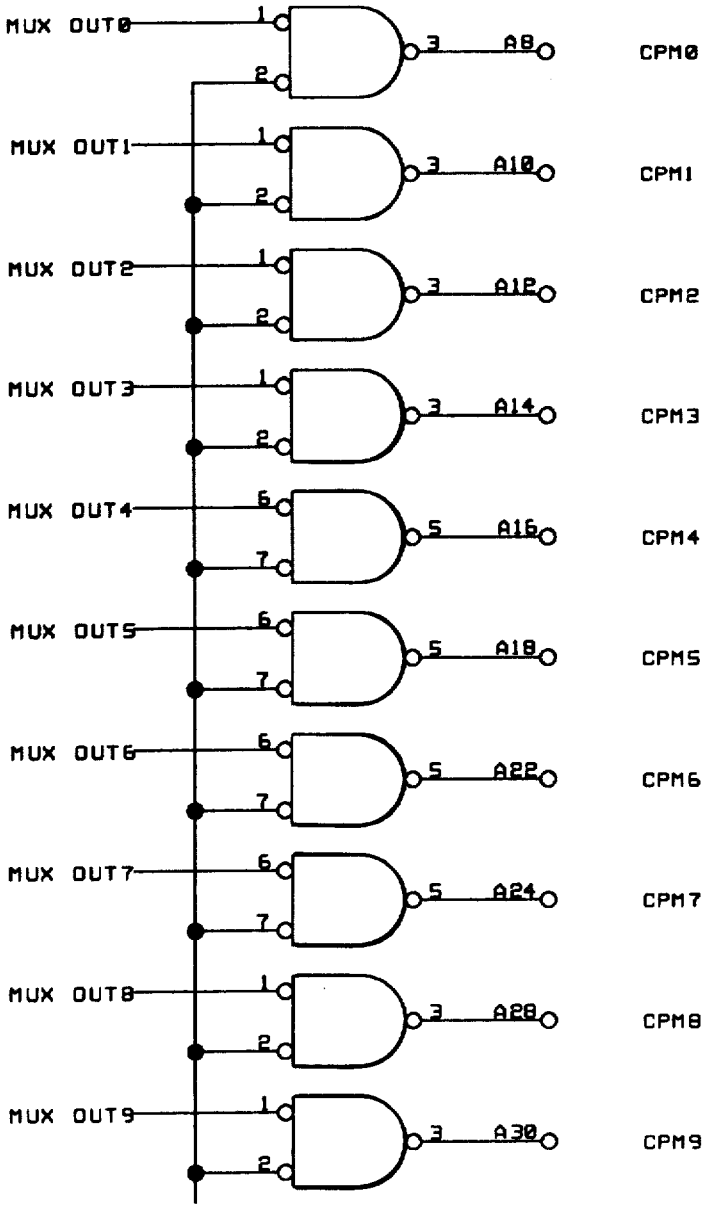

FROM FIG. 17
SEL B
CW D24
CADDR7
CW D25
INDEX3
CW D26
INDEX4
CW D27
INDEX5
A0
B0
A1
B1
A2
B2
A3
B3
2 LINE MUX
Z0
Z1
Z2
Z3
ENAB
J1-38
J1-39
J3-2
J3-3
CA/W D24
CA/W D25
CA/W D26
CA/W D27
SEL B
CW D28
INDEX6
CW D29
INDEX7
CW D30
INDEX8
CW D31
INDEX9
A0
B0
A1
B1
A2
B2
A3
B3
2 LINE MUX
Z0
Z1
Z2
Z3
ENAB
J3-5
J3-6
J3-8
J3-9
CA/W D28
CA/W D29
CA/W D30
CA/W D31

FIG. 17A



TO FIG. 18A

DRIVER UNITS 50 AND 51 AND DRIVER LOGIC

FIG. 18

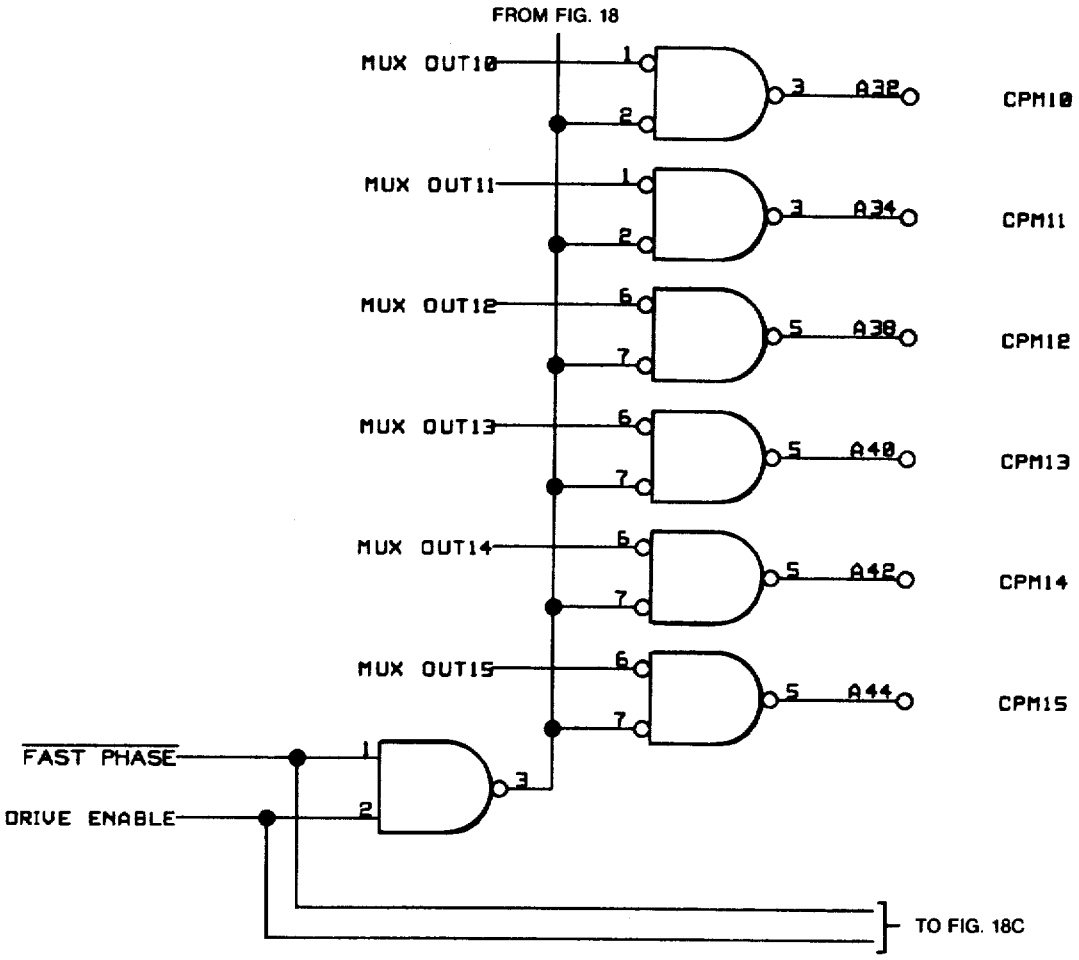

FROM FIG. 18
MUX OUT10
MUX OUT11
MUX OUT12
MUX OUT13
MUX OUT14
MUX OUT15
A32
A34
A38
A40
A42
A44
CPM10
CPM11
CPM12
CPM13
CPM14
CPM15
FAST PHASE
DRIVE ENABLE
TO FIG. 18C

FIG. 18A

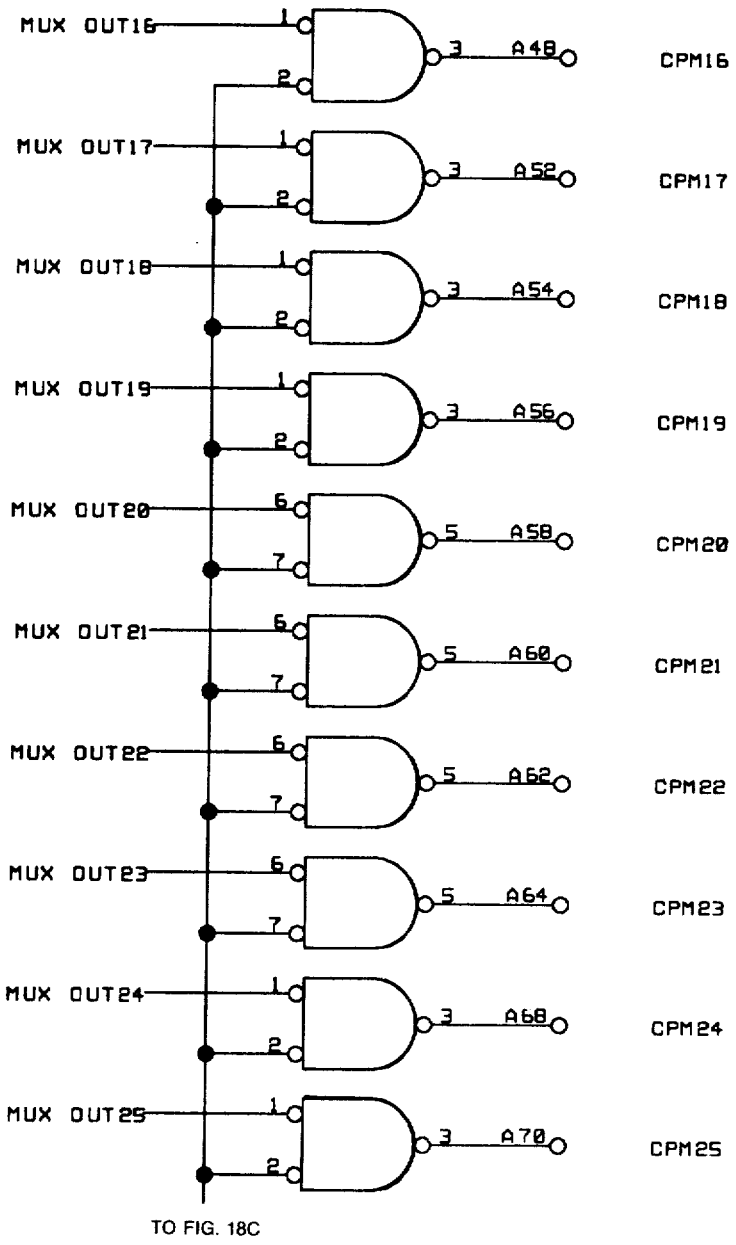

MUX OUT16
MUX OUT17
MUX OUT18
MUX OUT19
MUX OUT20
MUX OUT21
MUX OUT22
MUX OUT23
MUX OUT24
MUX OUT25
A48
A52
A54
A56
A58
A60
A62
A64
A68
A70
CPM16
CPM17
CPM18
CPM19
CPM20
CPM21
CPM22
CPM23
CPM24
CPM25
TO FIG. 18C

FIG. 18B

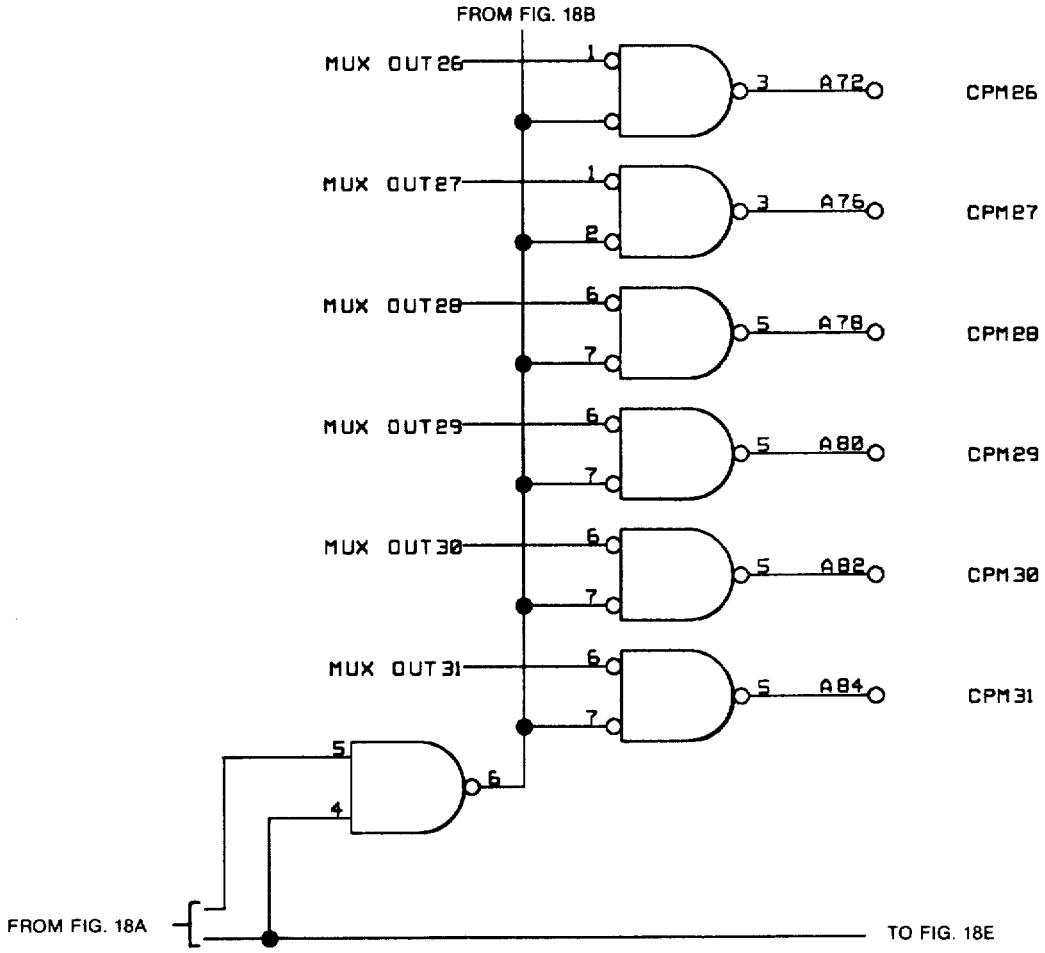

FROM FIG. 18B
MUX OUT26
MUX OUT27
MUX OUT28
MUX OUT29
MUX OUT30
MUX OUT31
A72
A76
A78
A80
A82
A84
CPM26
CPM27
CPM28
CPM29
CPM30
CPM31
FROM FIG. 18A
TO FIG. 18E

FIG. 18C

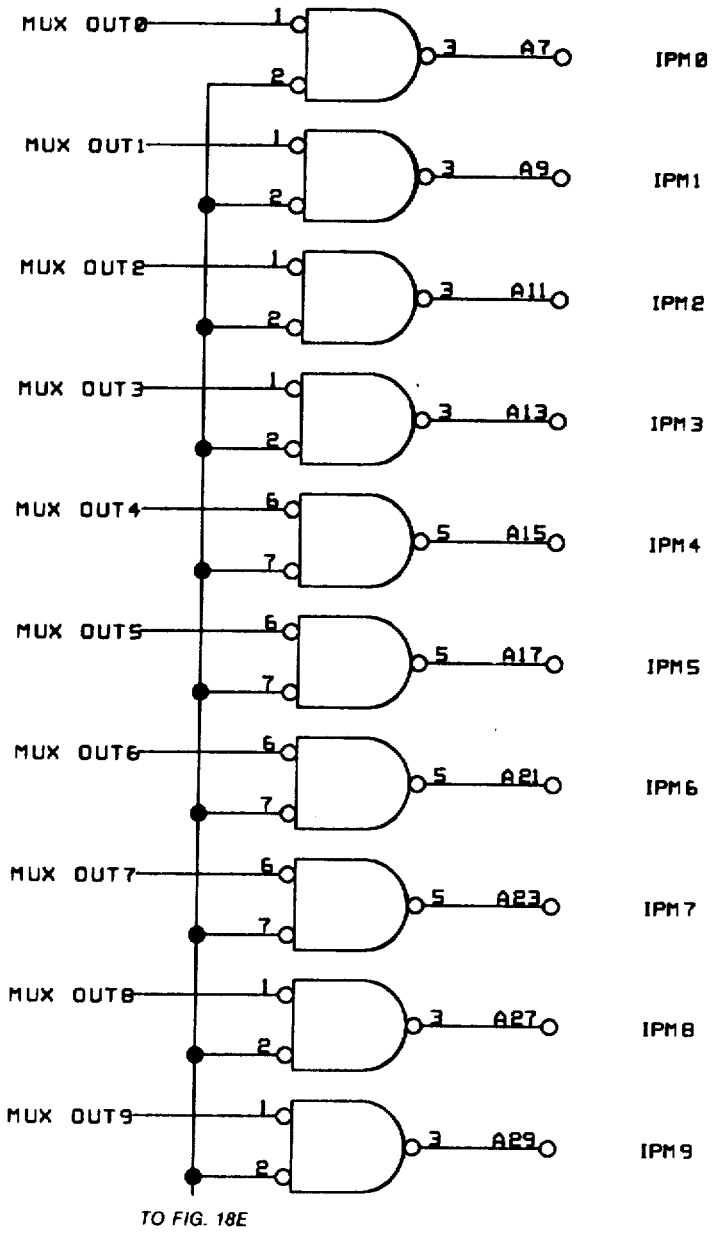
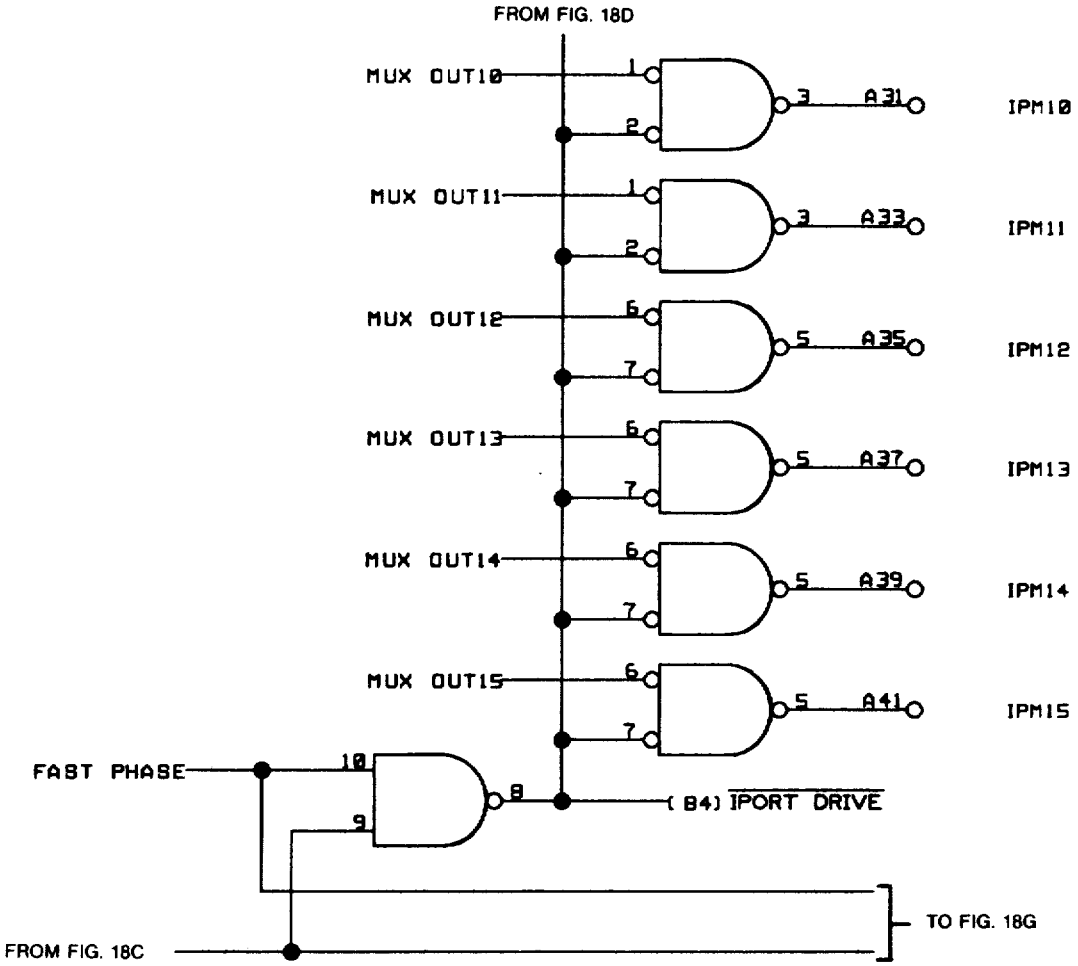

MUX OUT0
MUX OUT1
MUX OUT2
MUX OUT3
MUX OUT4
MUX OUT5
MUX OUT6
MUX OUT7
MUX OUT8
MUX OUT9
A7
A9
A11
A13
A15
A17
A21
A23
A27
A29
IPM0
IPM1
IPM2
IPM3
IPM4
IPM5
IPM6
IPM7
IPM8
IPM9
TO FIG. 18E

FIG. 18D


FROM FIG. 18D
MUX OUT10
MUX OUT11
MUX OUT12
MUX OUT13
MUX OUT14
MUX OUT15
A31
A33
A35
A37
A39
A41
IPM10
IPM11
IPM12
IPM13
IPM14
IPM15
FAST PHASE
(B4) IPORT DRIVE
TO FIG. 18G
FROM FIG. 18C

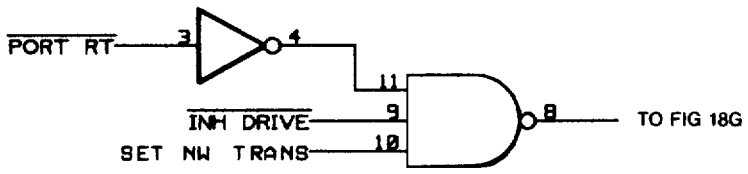

PORT RT
INH DRIVE
SET NW TRANS
TO FIG 18G

FIG. 18E

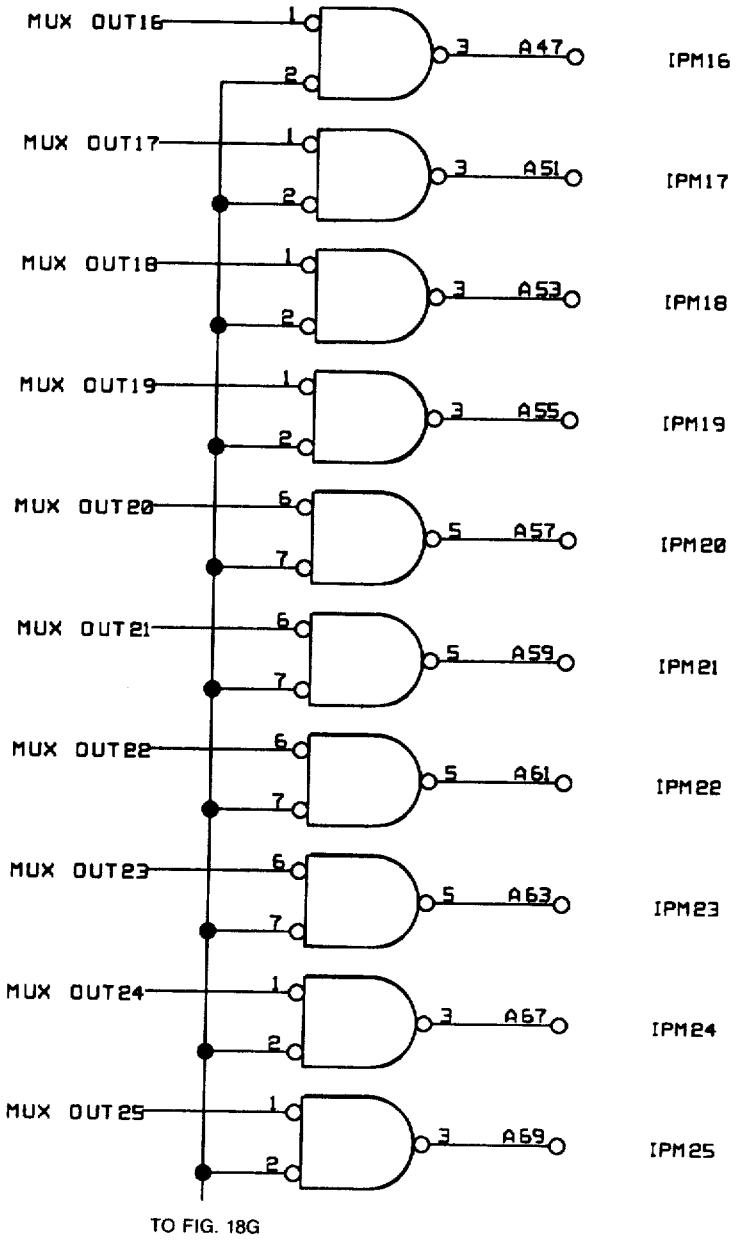
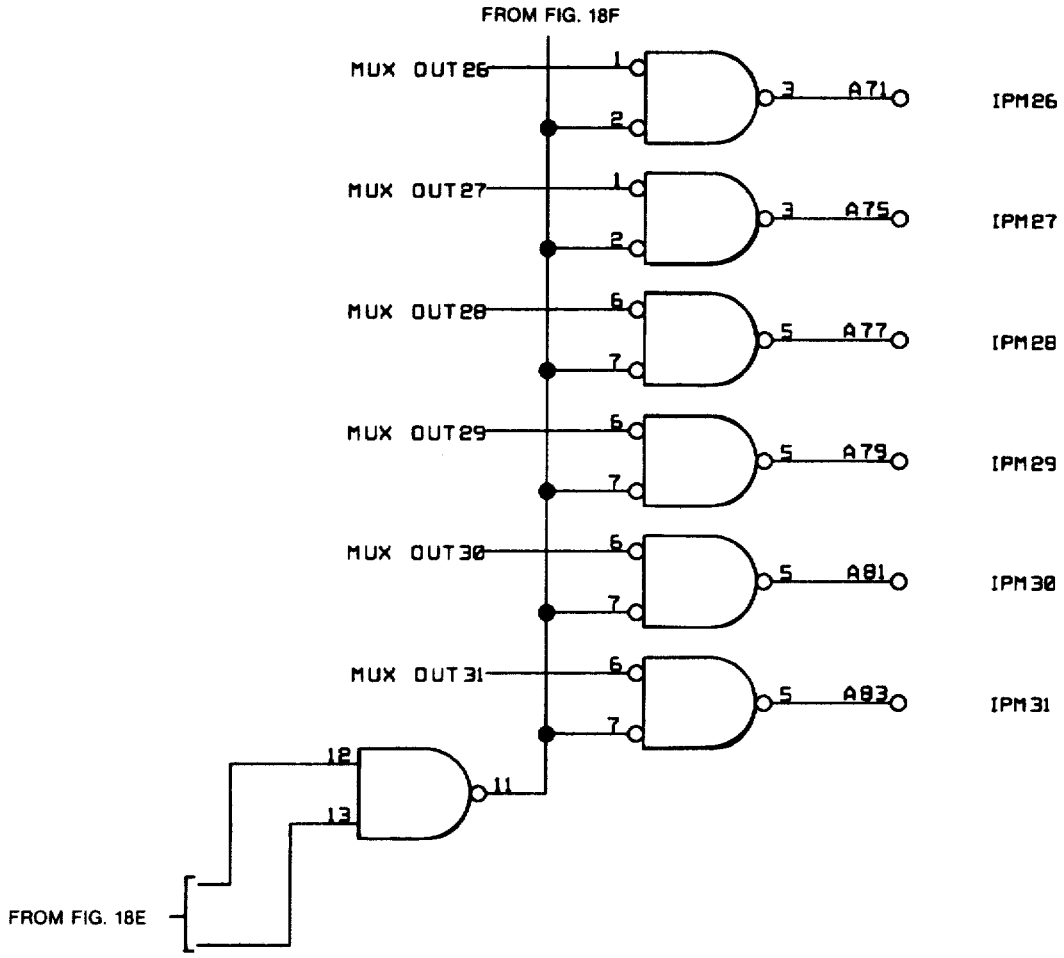

MUX OUT16
MUX OUT17
MUX OUT18
MUX OUT19
MUX OUT20
MUX OUT21
MUX OUT22
MUX OUT23
MUX OUT24
MUX OUT25
A47
A51
A53
A55
A57
A59
A61
A63
A67
A69
IPM16
IPM17
IPM18
IPM19
IPM20
IPM21
IPM22
IPM23
IPM24
IPM25
TO FIG. 18G

FIG. 18F


FROM FIG. 18F
MUX OUT26
MUX OUT27
MUX OUT28
MUX OUT29
MUX OUT30
MUX OUT31
A71
A75
A77
A79
A81
A83
IPM26
IPM27
IPM28
IPM29
IPM30
IPM31
FROM FIG. 18E

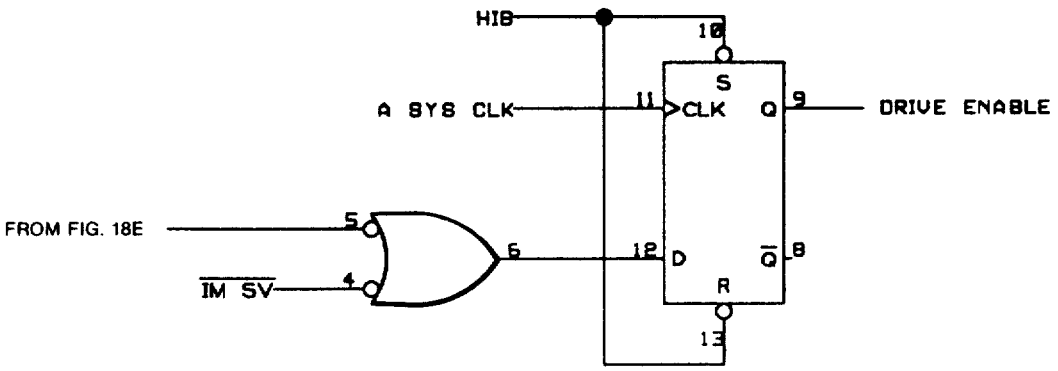
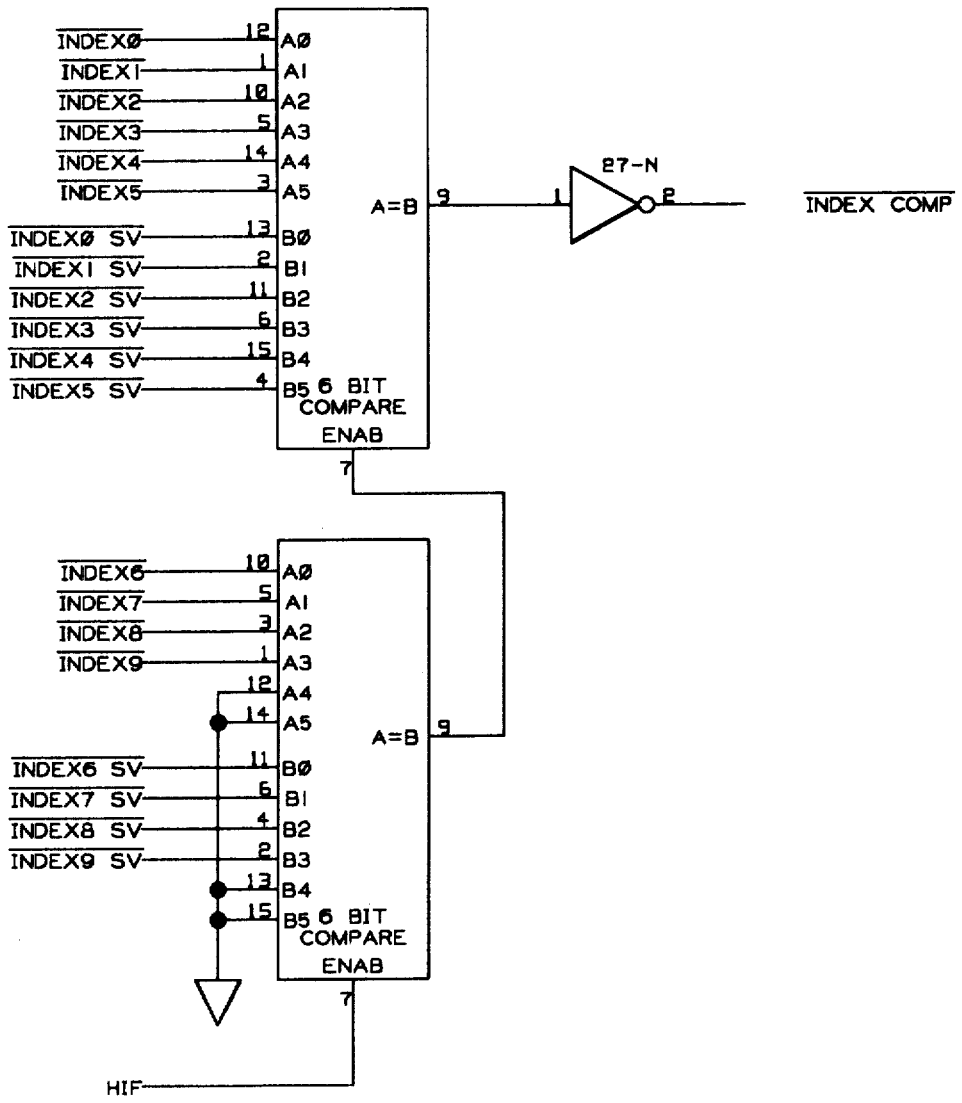
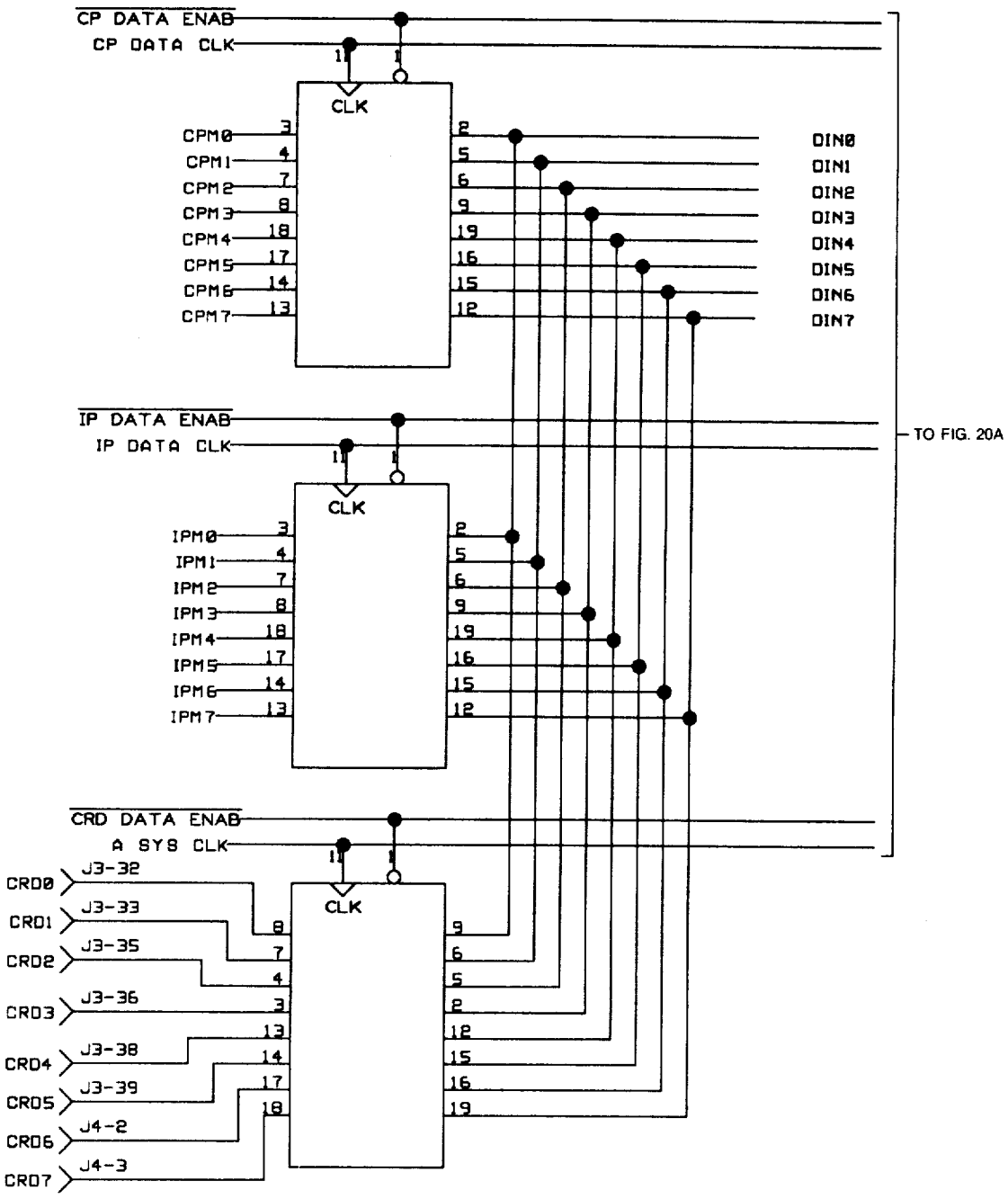

HIB
A SYS CLK
CLK
S
Q
DRIVE ENABLE
FROM FIG. 18E
IM SV
D
R

FIG. 18G



INDEX/INDEX SV COMPARATOR

FIG. 19



CPU BUFFER DATA REGISTER 54, I/O BUFFER DATA REGISTER 55, AND CRD IN REGISTER 53

FIG. 20

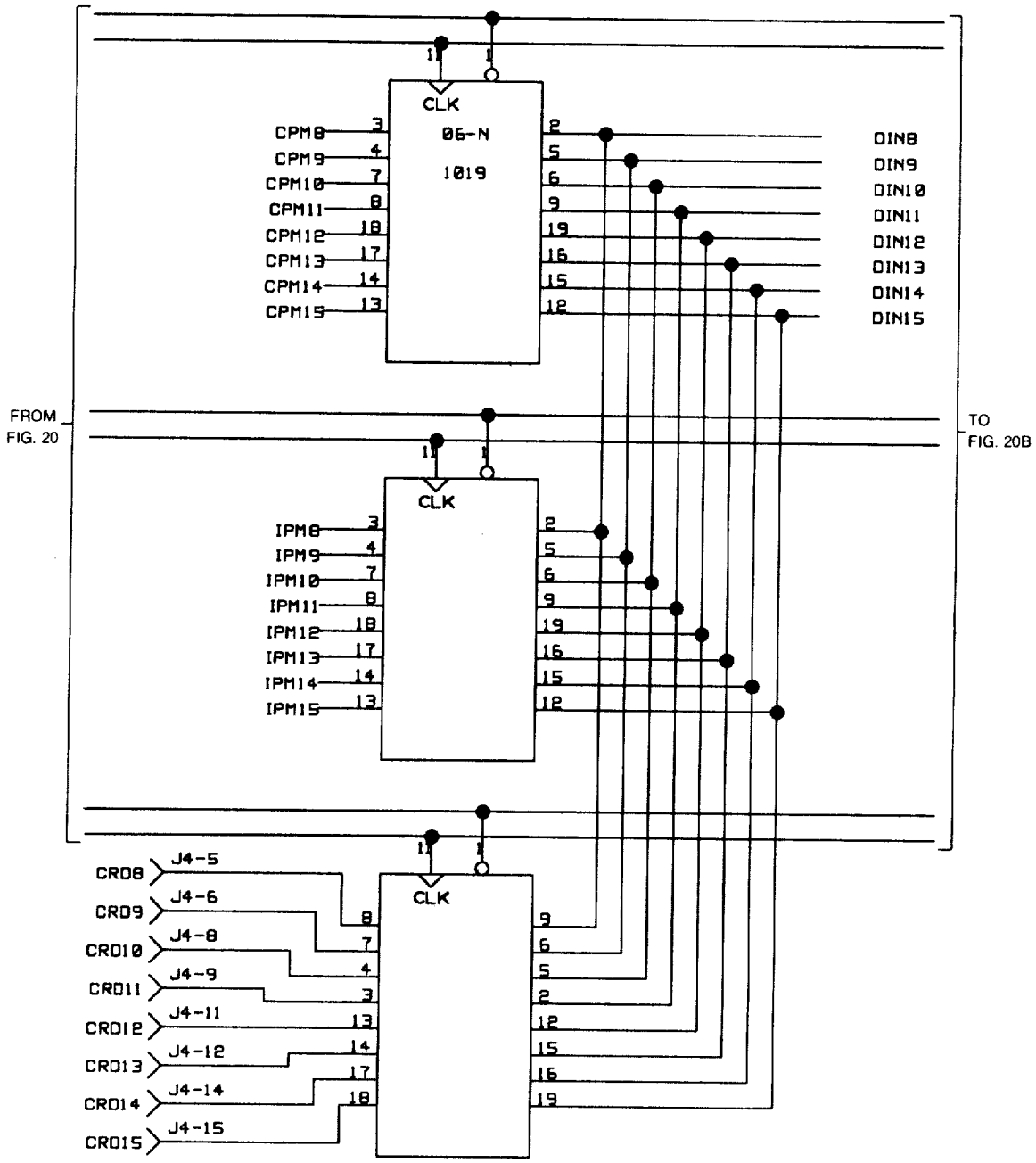

FROM FIG. 20
TO FIG. 20B
CLK
06-N
1019
CPM8
CPM9
CPM10
CPM11
CPM12
CPM13
CPM14
CPM15
DIN8
DIN9
DIN10
DIN11
DIN12
DIN13
DIN14
DIN15
IPM8
IPM9
IPM10
IPM11
IPM12
IPM13
IPM14
IPM15
CRD8
CRD9
CRD10
CRD11
CRD12
CRD13
CRD14
CRD15
J4-5
J4-6
J4-8
J4-9
J4-11
J4-12
J4-14
J4-15

FIG.20A

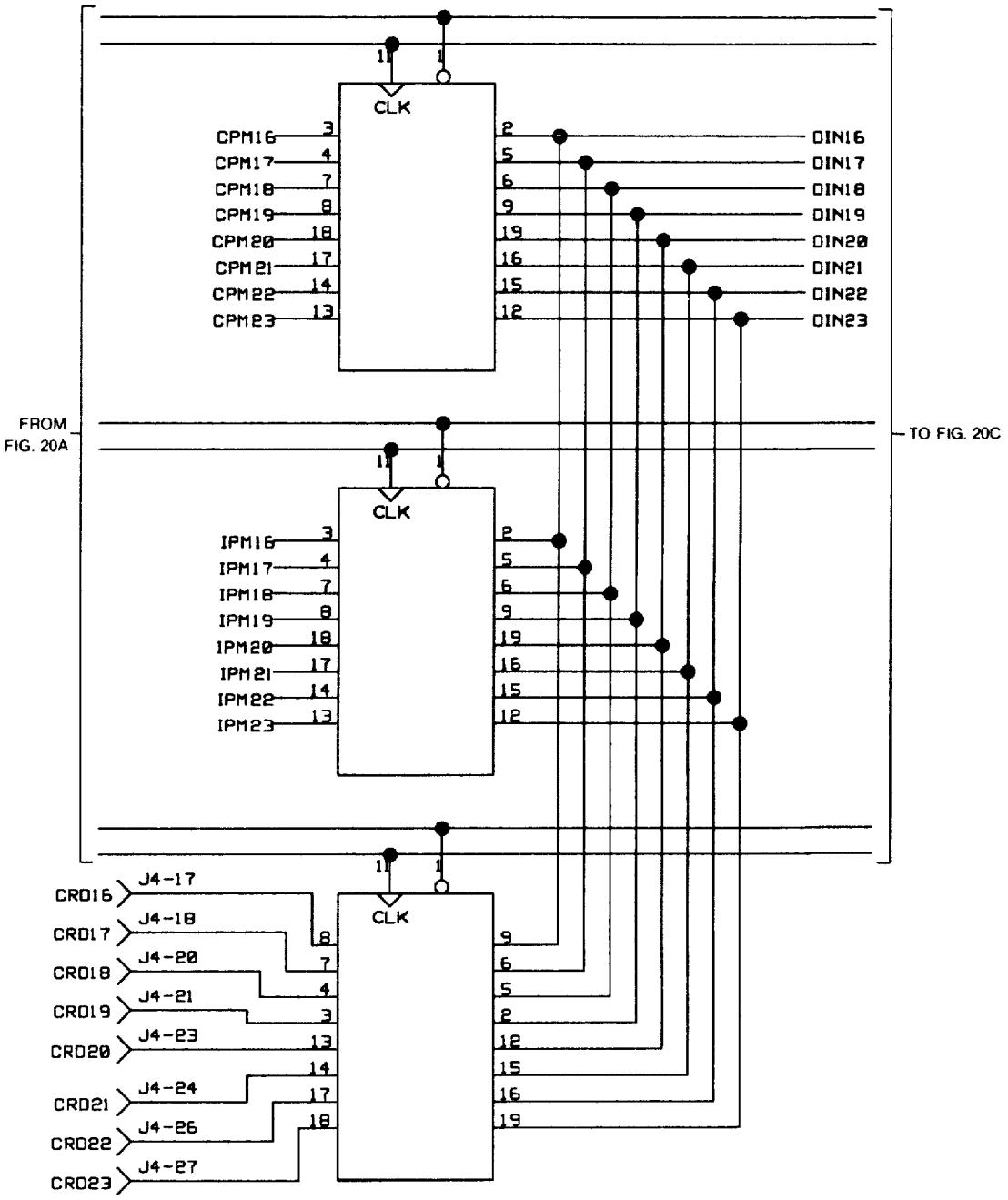

CLK
CPM16
CPM17
CPM18
CPM19
CPM20
CPM21
CPM22
CPM23
DIN16
DIN17
DIN18
DIN19
DIN20
DIN21
DIN22
DIN23
FROM FIG. 20A
TO FIG. 20C
IPM16
IPM17
IPM18
IPM19
IPM20
IPM21
IPM22
IPM23
CRD16
CRD17
CRD18
CRD19
CRD20
CRD21
CRD22
CRD23
J4-17
J4-18
J4-20
J4-21
J4-23
J4-24
J4-26
J4-27

FIG. 20B

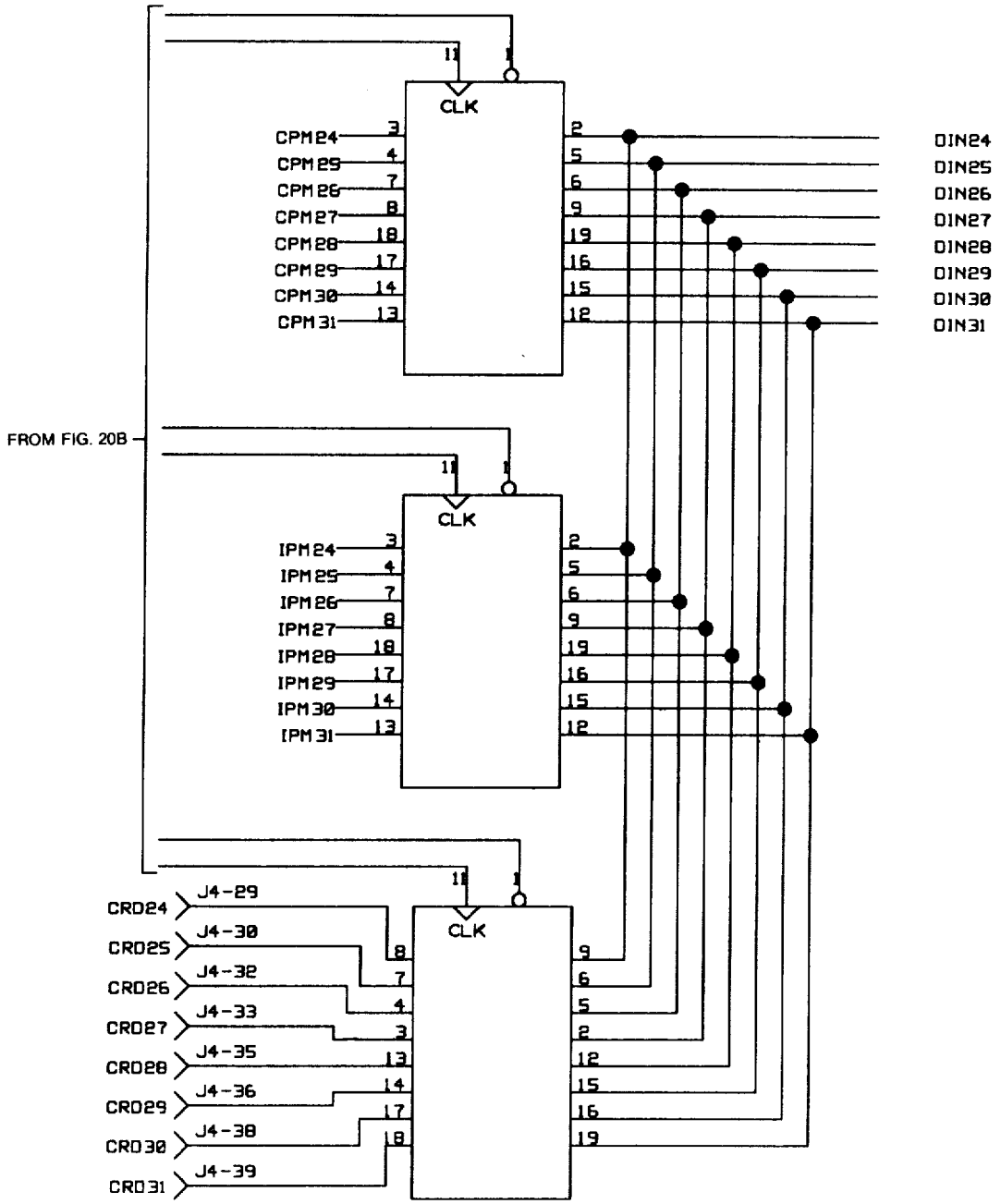
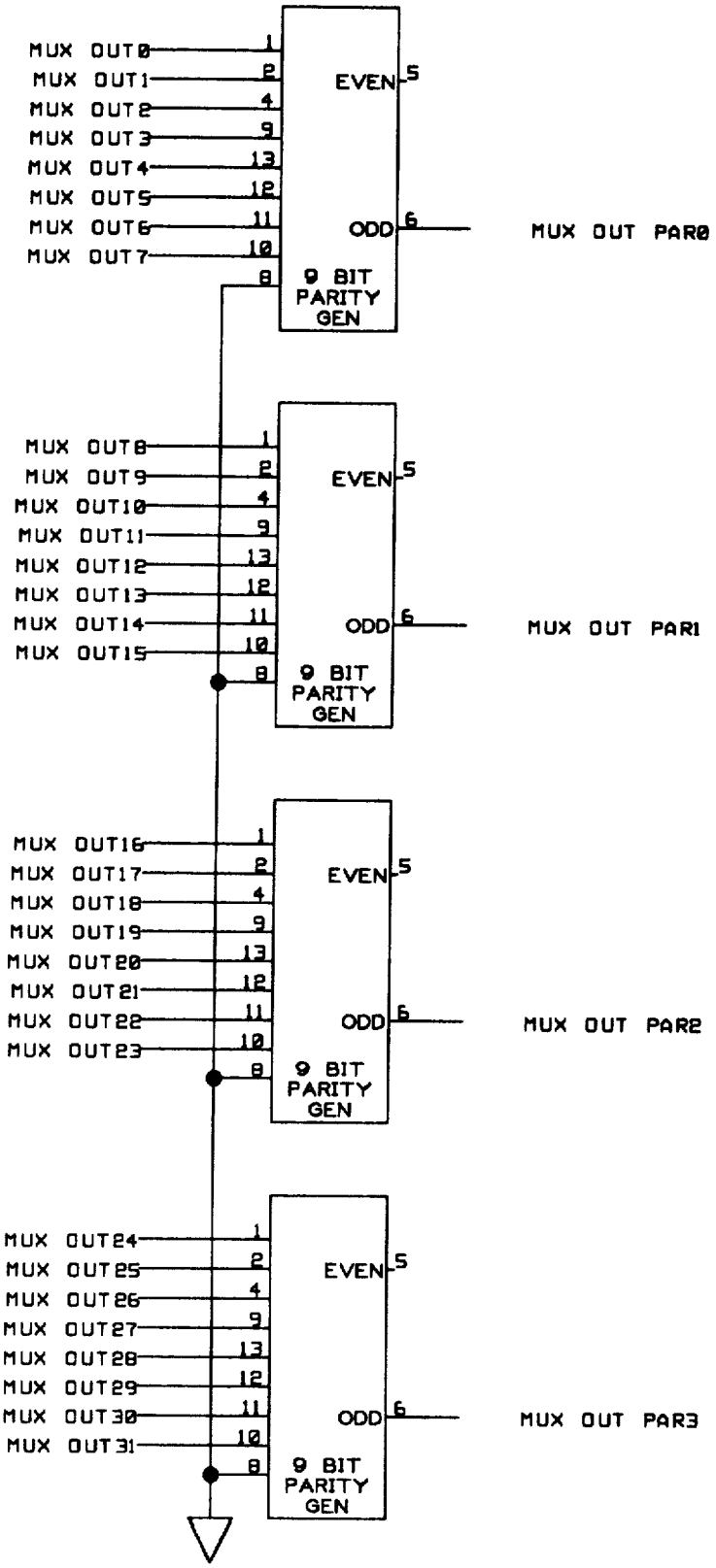
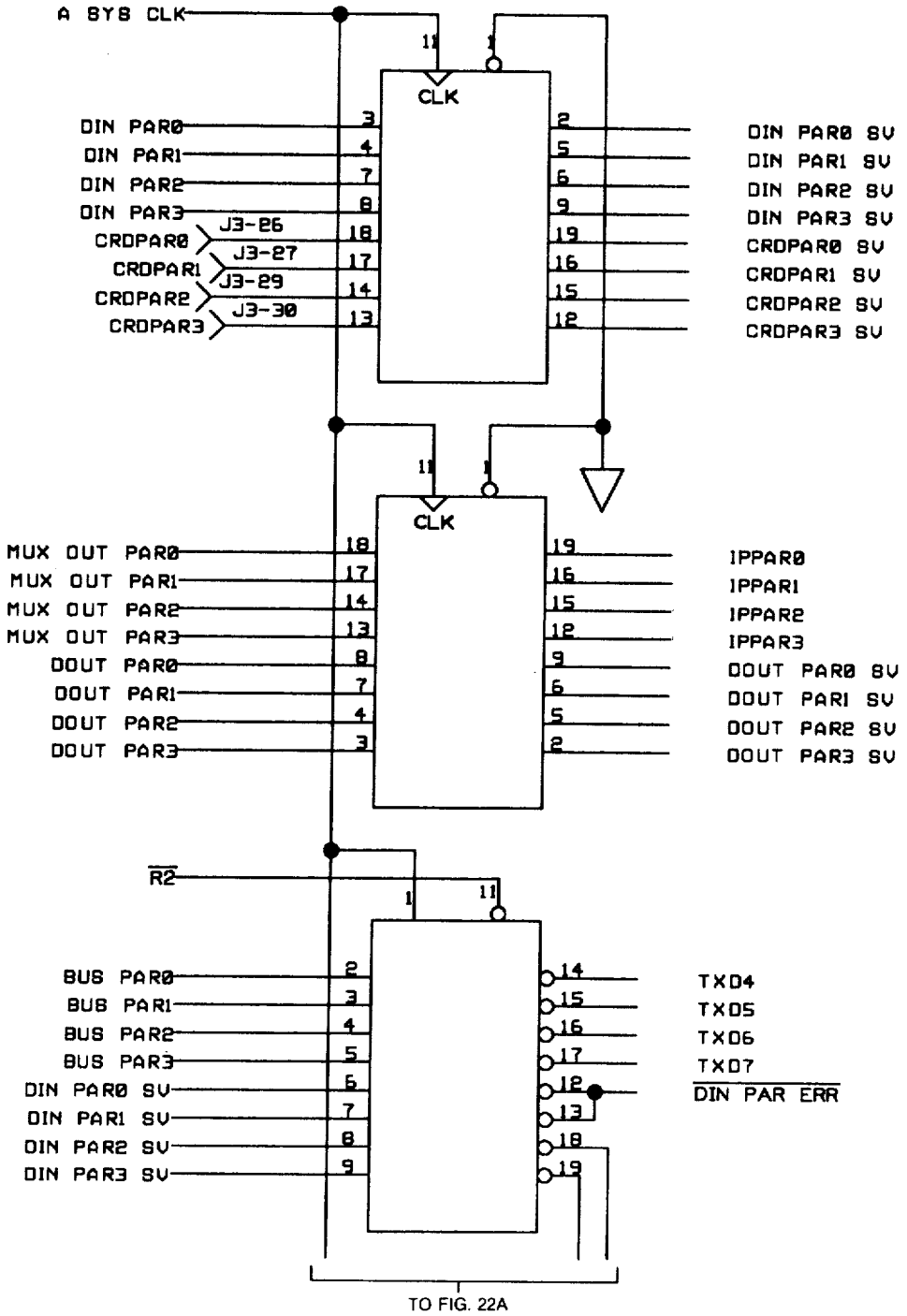

CLK
CPM24
CPM25
CPM26
CPM27
CPM28
CPM29
CPM30
CPM31
DIN24
DIN25
DIN26
DIN27
DIN28
DIN29
DIN30
DIN31
FROM FIG. 20B
IPM24
IPM25
IPM26
IPM27
IPM28
IPM29
IPM30
IPM31
CRD24
CRD25
CRD26
CRD27
CRD28
CRD29
CRD30
CRD31
J4-29
J4-30
J4-32
J4-33
J4-35
J4-36
J4-38
J4-39

FIG. 20C



MUX 48 PARITY GENERATION LOGIC

FIG. 21



SYSTEM CACHE PARITY LOGIC

FIG. 22

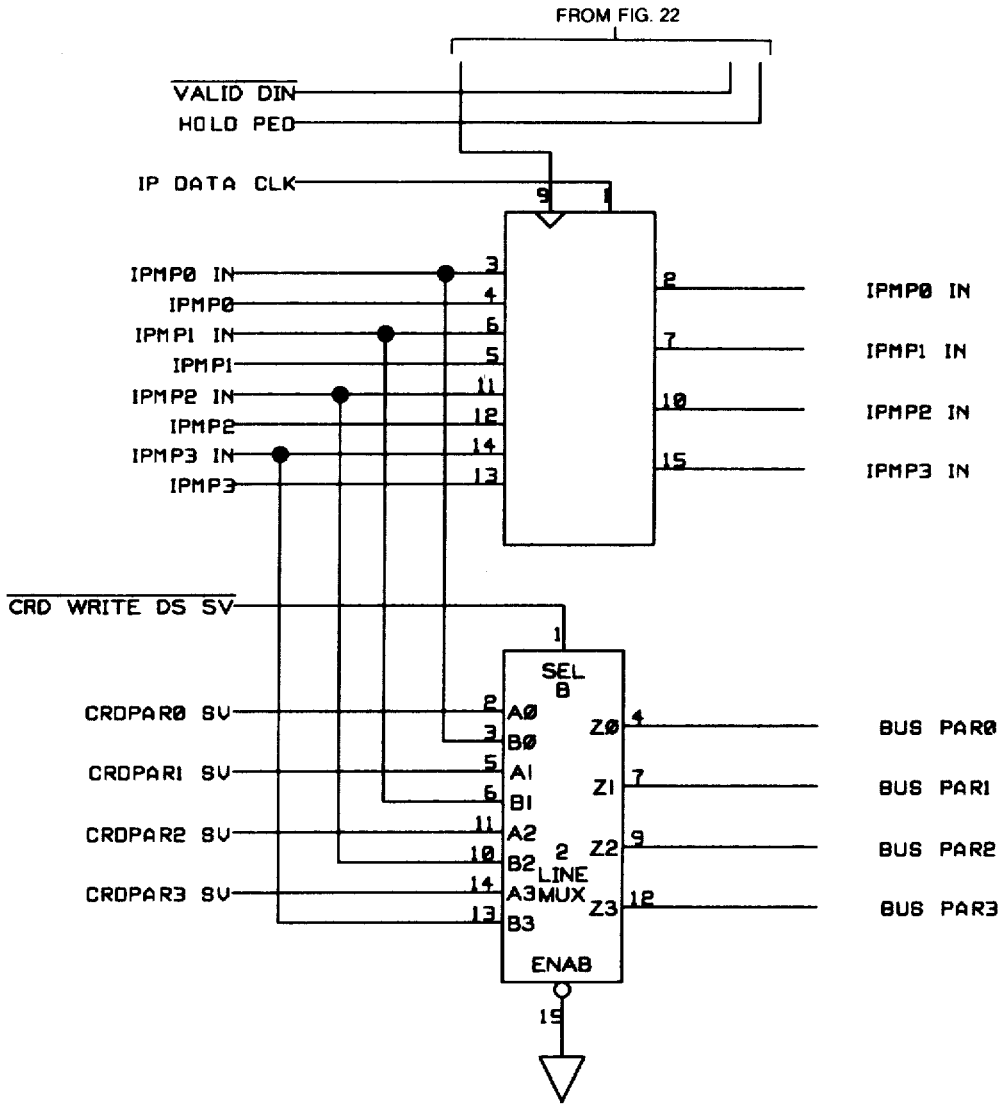
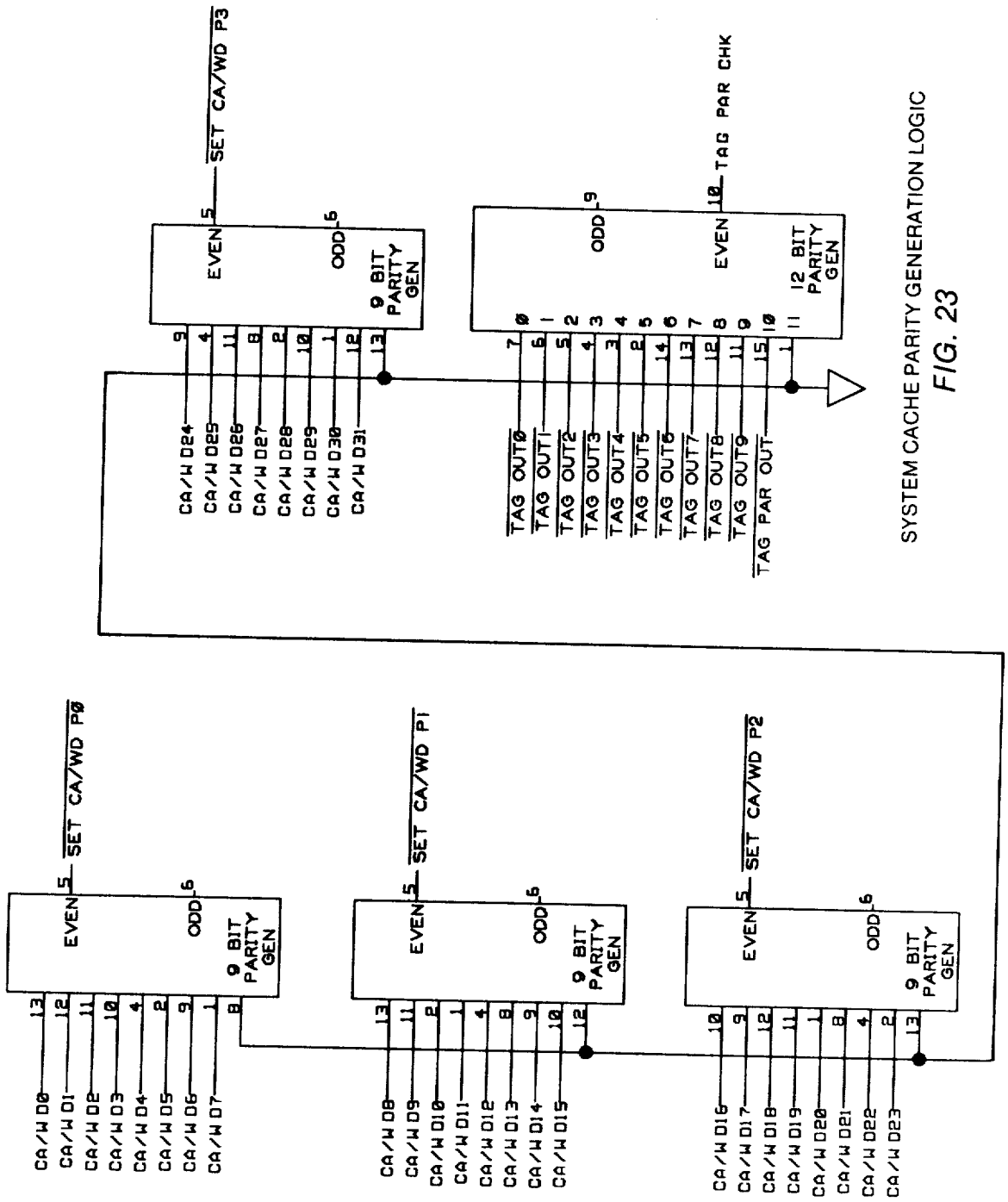
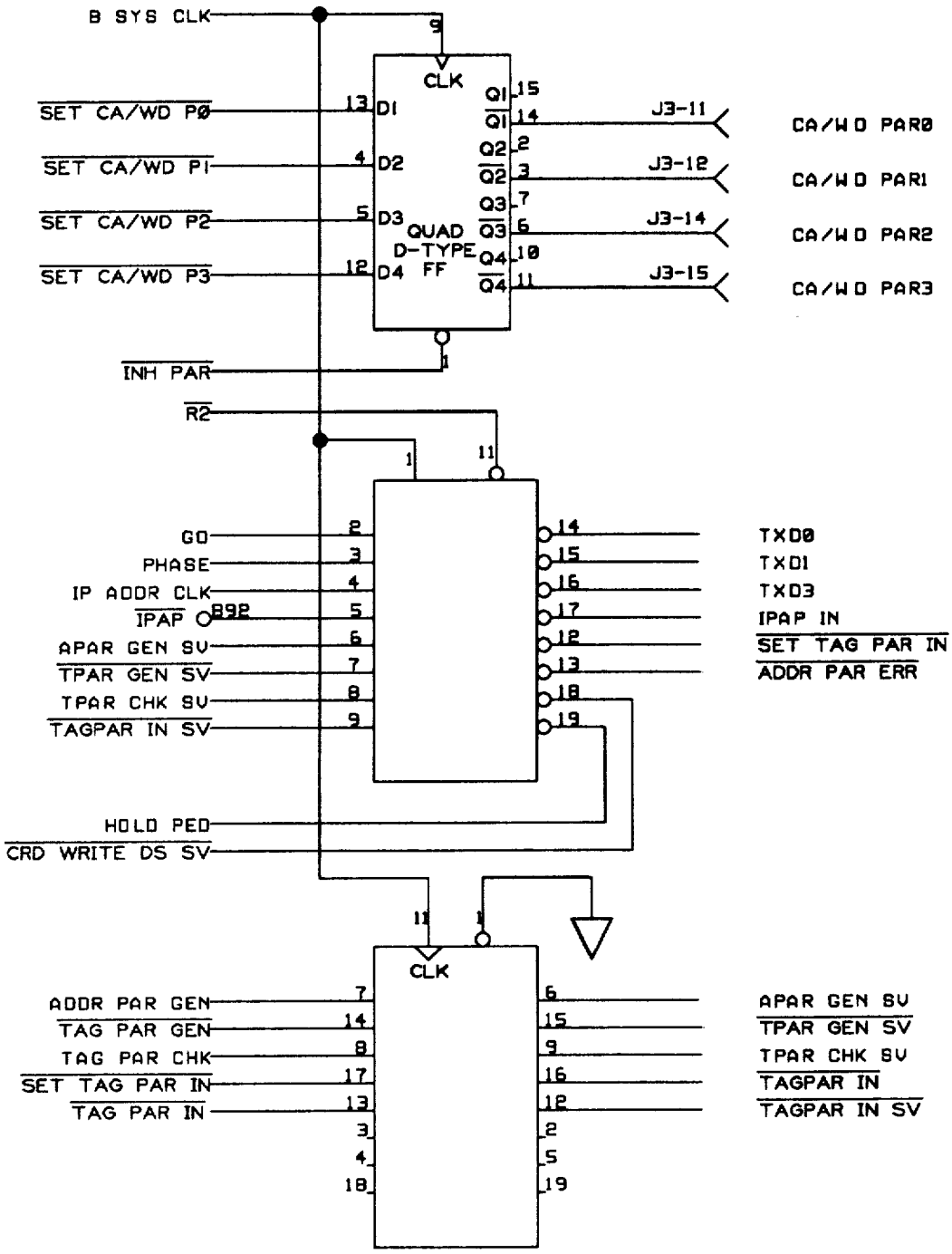

FROM FIG. 22
VALID DIN
HOLD PED
IP DATA CLK
IPMP0 IN
IPMP0
IPMP1 IN
IPMP1
IPMP2 IN
IPMP2
IPMP3 IN
IPMP3
IPMP0 IN
IPMP1 IN
IPMP2 IN
IPMP3 IN
CRD WRITE DS SV
SEL B
CRDPAR0 SV
CRDPAR1 SV
CRDPAR2 SV
CRDPAR3 SV
A0
B0
A1
B1
A2
B2
A3
B3
Z0
Z1
Z2
Z3
2 LINE MUX
ENAB
BUS PAR0
BUS PAR1
BUS PAR2
BUS PAR3

FIG. 22A



SYSTEM CACHE PARITY GENERATION LOGIC

FIG. 23



SYSTEM CACHE PARITY GENERATOR LOGIC

FIG. 24

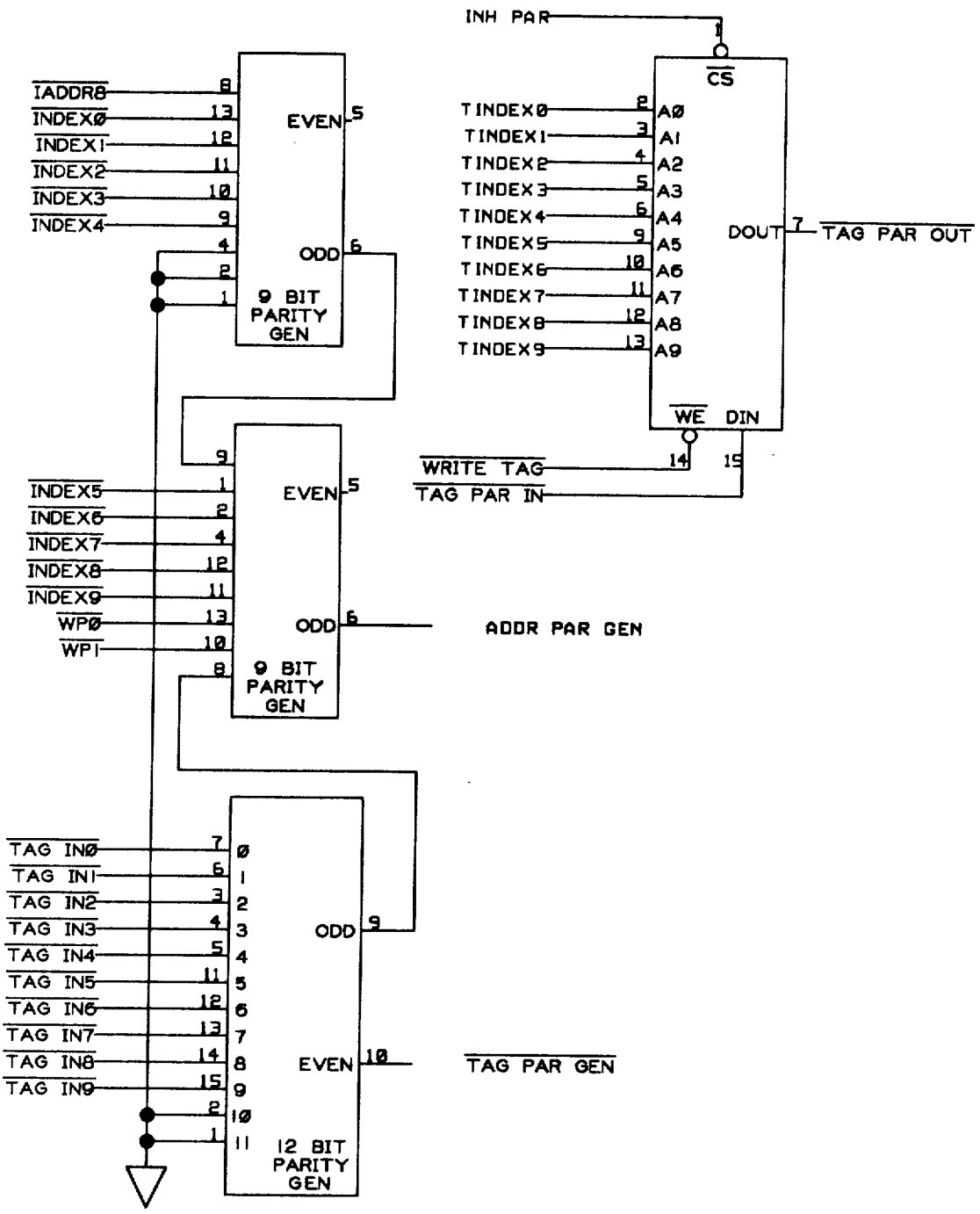
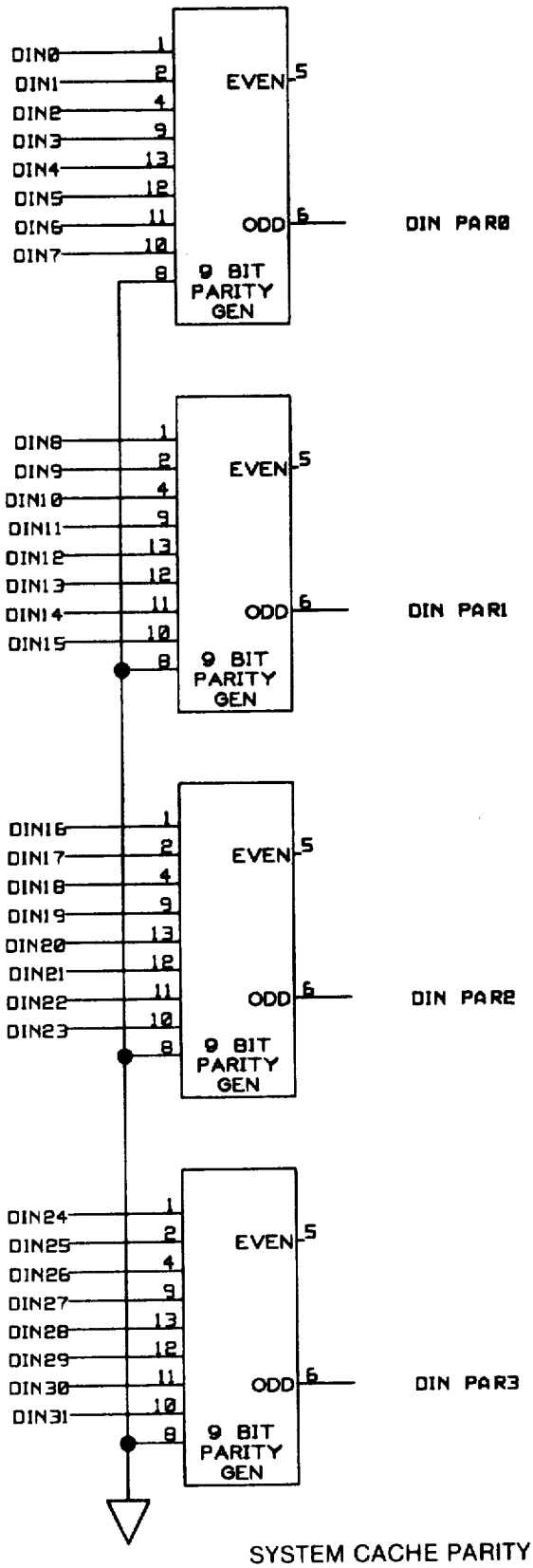
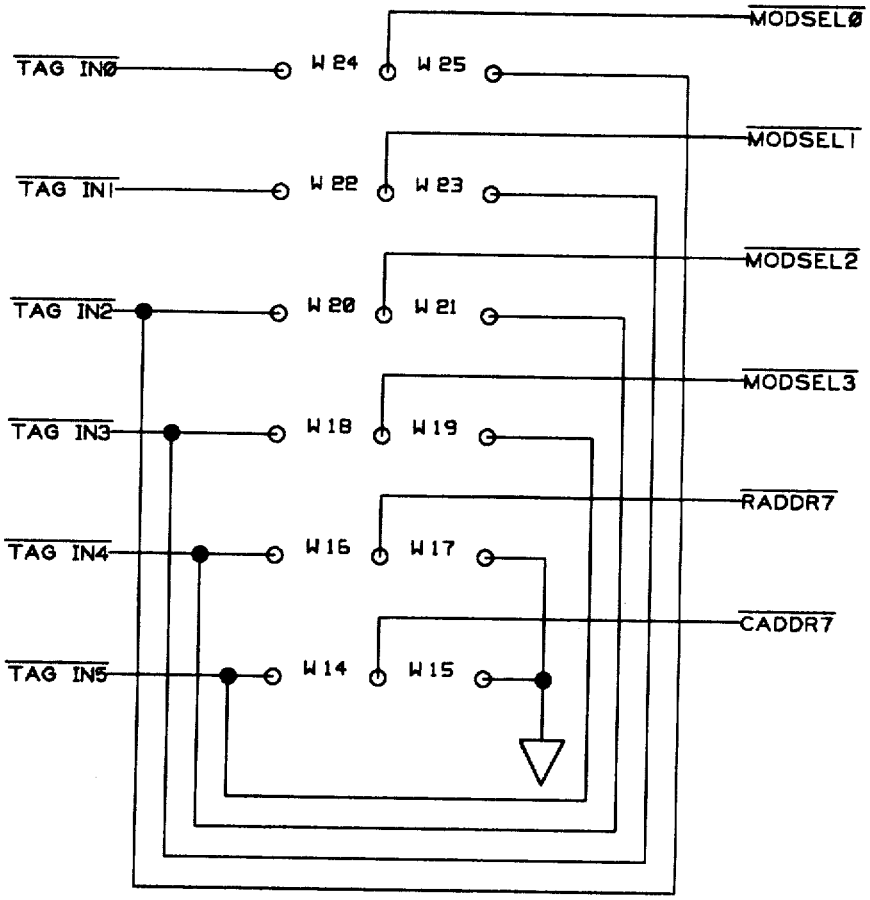

INH PAR
CS
IADDR8
INDEX0
INDEX1
INDEX2
INDEX3
INDEX4
EVEN
ODD
9 BIT PARITY GEN
TINDEX0
TINDEX1
TINDEX2
TINDEX3
TINDEX4
TINDEX5
TINDEX6
TINDEX7
TINDEX8
TINDEX9
A0
A1
A2
A3
A4
A5
A6
A7
A8
A9
DOUT
TAG PAR OUT
WE
DIN
WRITE TAG
TAG PAR IN
INDEX5
INDEX6
INDEX7
INDEX8
INDEX9
WP0
WP1
ADDR PAR GEN
TAG IN0
TAG IN1
TAG IN2
TAG IN3
TAG IN4
TAG IN5
TAG IN6
TAG IN7
TAG IN8
TAG IN9
12 BIT PARITY GEN
TAG PAR GEN

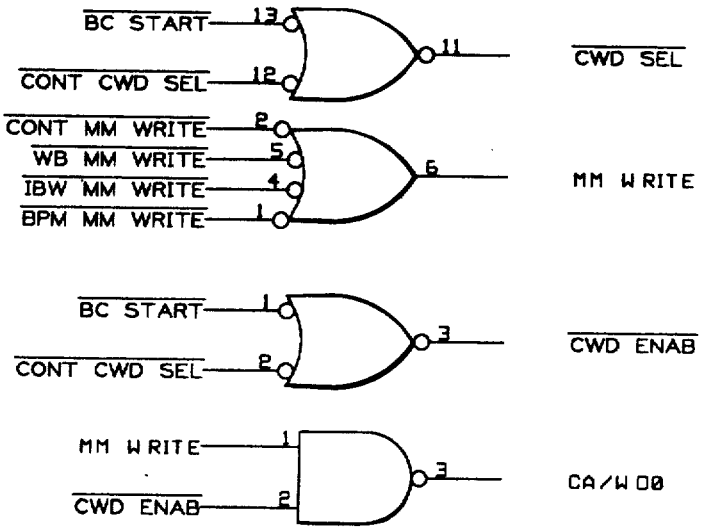
FIG. 24A



SYSTEM CACHE PARITY LOGIC

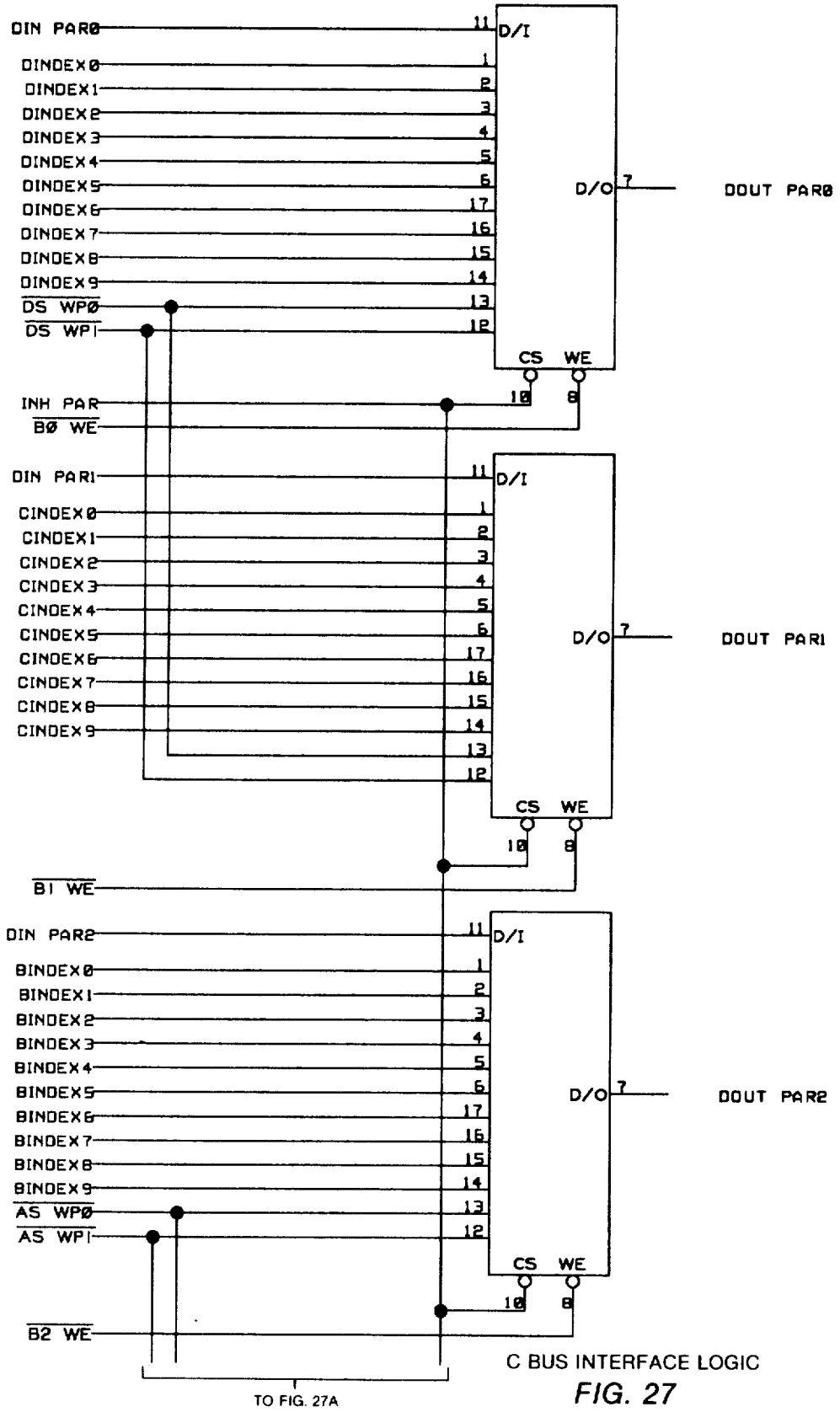
FIG. 25





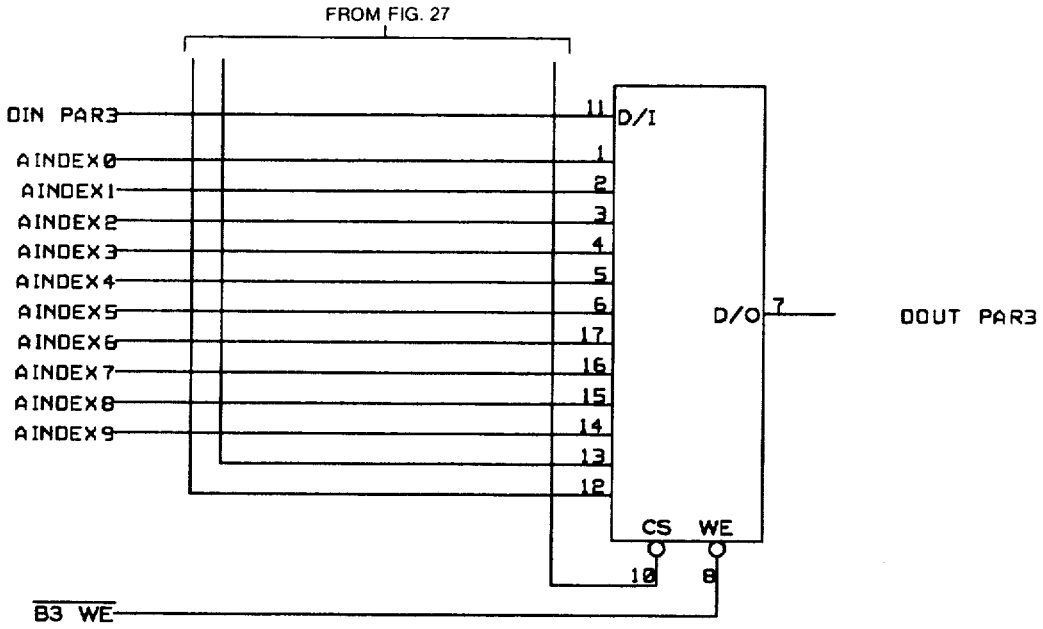
MAIN MEMORY INTERFACE CONTROL LOGIC

FIG. 26



C BUS INTERFACE LOGIC

FIG. 27


FROM FIG. 27
DIN PAR3
AINDEX0
AINDEX1
AINDEX2
AINDEX3
AINDEX4
AINDEX5
AINDEX6
AINDEX7
AINDEX8
AINDEX9
D/I
D/O
DOUT PAR3
CS
WE
B3 WE

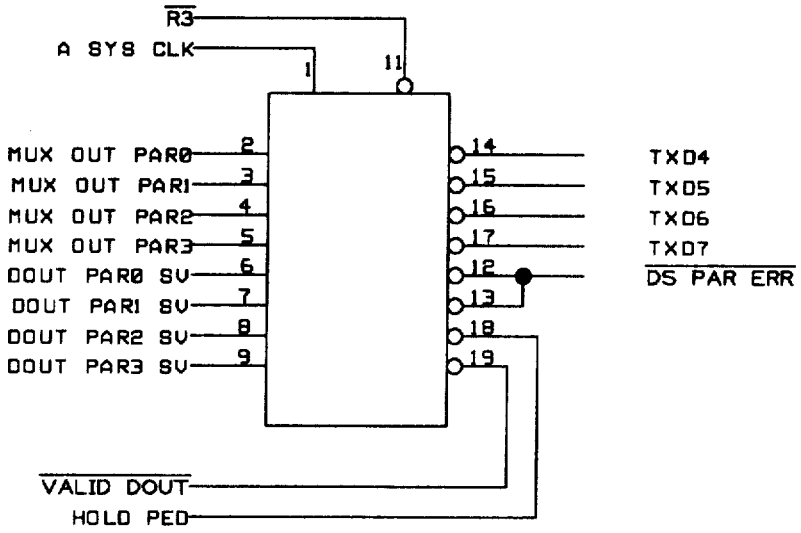
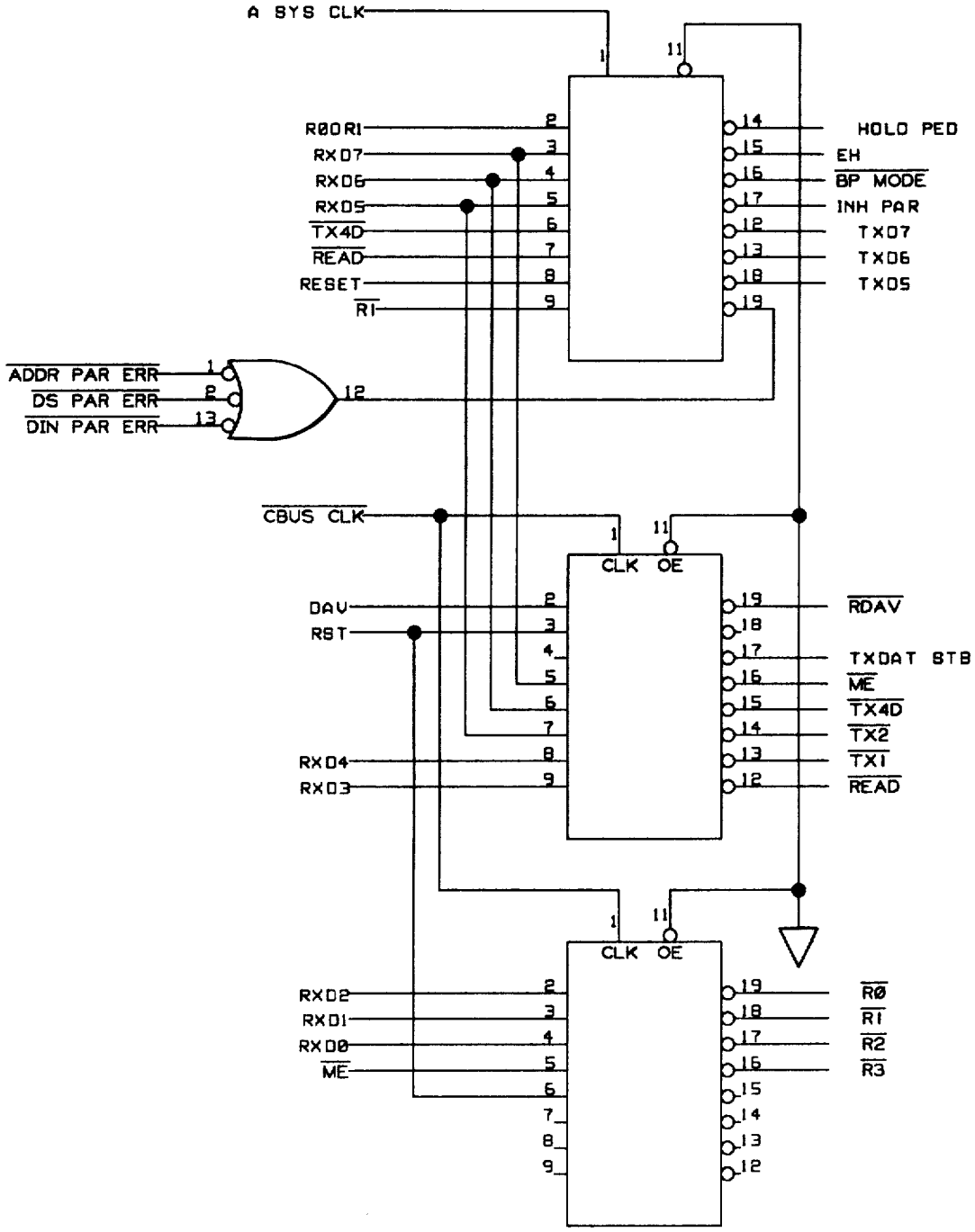

R3
A SYS CLK
MUX OUT PAR0
MUX OUT PAR1
MUX OUT PAR2
MUX OUT PAR3
DOUT PAR0 SV
DOUT PAR1 SV
DOUT PAR2 SV
DOUT PAR3 SV
TXD4
TXD5
TXD6
TXD7
DS PAR ERR
VALID DOUT
HOLD PED

FIG. 27A



C BUS INTERFACE LOGIC

FIG. 28

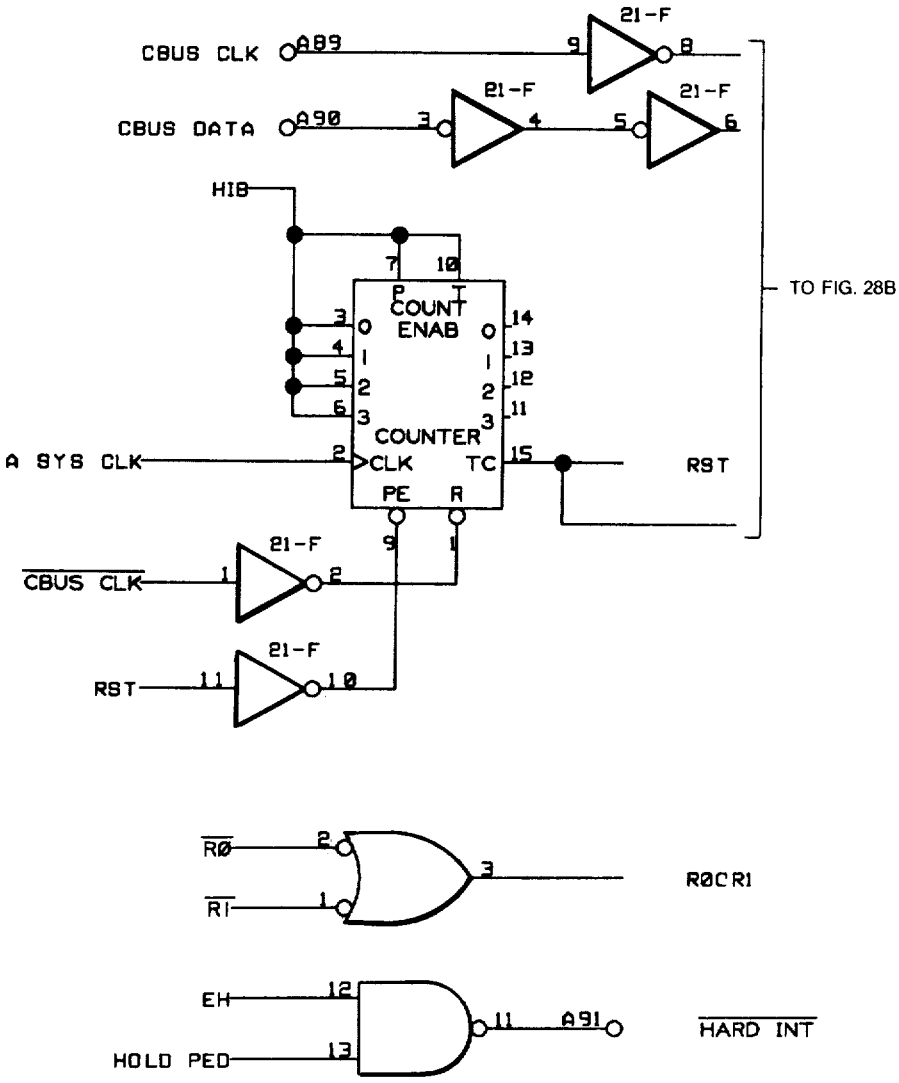

CBUS CLK
A89
21-F
CBUS DATA
A90
21-F
21-F
HIB
COUNT ENAB
COUNTER
A SYS CLK
CLK
TC
PE
R
RST
TO FIG. 28B
CBUS CLK
21-F
RST
21-F
RØ
RI
RØCRI
EH
HOLD PED
A91
HARD INT

FIG. 28A

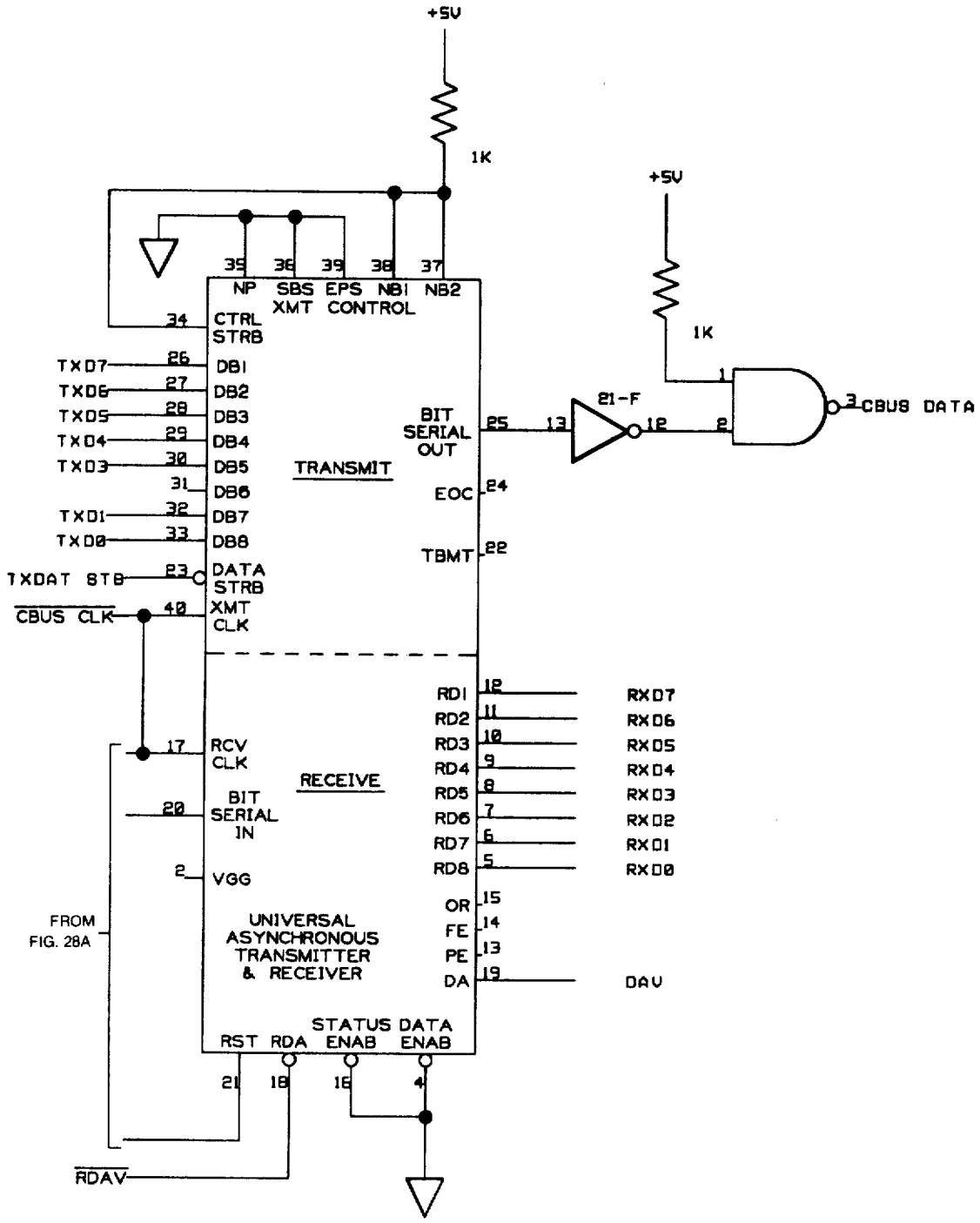
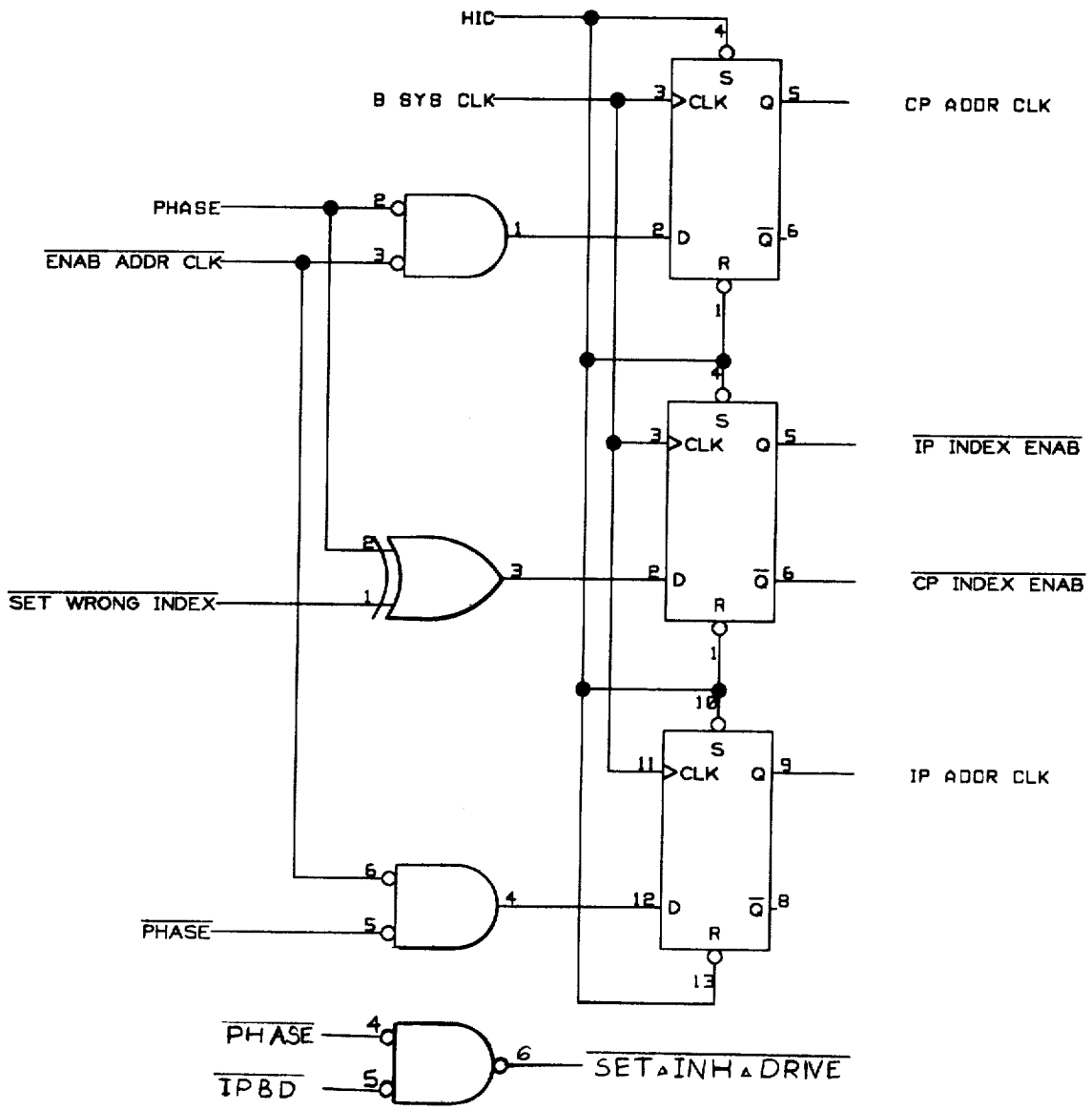

+5V
1K
+5V
1K
35
36
39
38
37
NP
SBS
EPS
NB1
NB2
XMT CONTROL
34
CTRL STRB
TXD7
26
DB1
TXD6
27
DB2
TXD5
28
DB3
TXD4
29
DB4
TXD3
30
DB5
31
DB6
TXD1
32
DB7
TXD0
33
DB8
TXDAT STB
23
DATA STRB
CBUS CLK
40
XMT CLK
TRANSMIT
BIT SERIAL OUT
25
13
21-F
12
1
2
3
CBUS DATA
EOC
24
TBMT
22
RD1
12
RD2
11
RD3
10
RD4
9
RD5
8
RD6
7
RD7
6
RD8
5
RXD7
RXD6
RXD5
RXD4
RXD3
RXD2
RXD1
RXD0
17
RCV CLK
RECEIVE
20
BIT SERIAL IN
2
VGG
FROM FIG. 28A
UNIVERSAL ASYNCHRONOUS TRANSMITTER & RECEIVER
OR
15
FE
14
PE
13
DA
19
DAV
STATUS ENAB
DATA ENAB
RST
RDA
21
18
16
4
RDAV

FIG. 28B



SYSTEM CACHE CONTROL LOGIC

FIG. 29

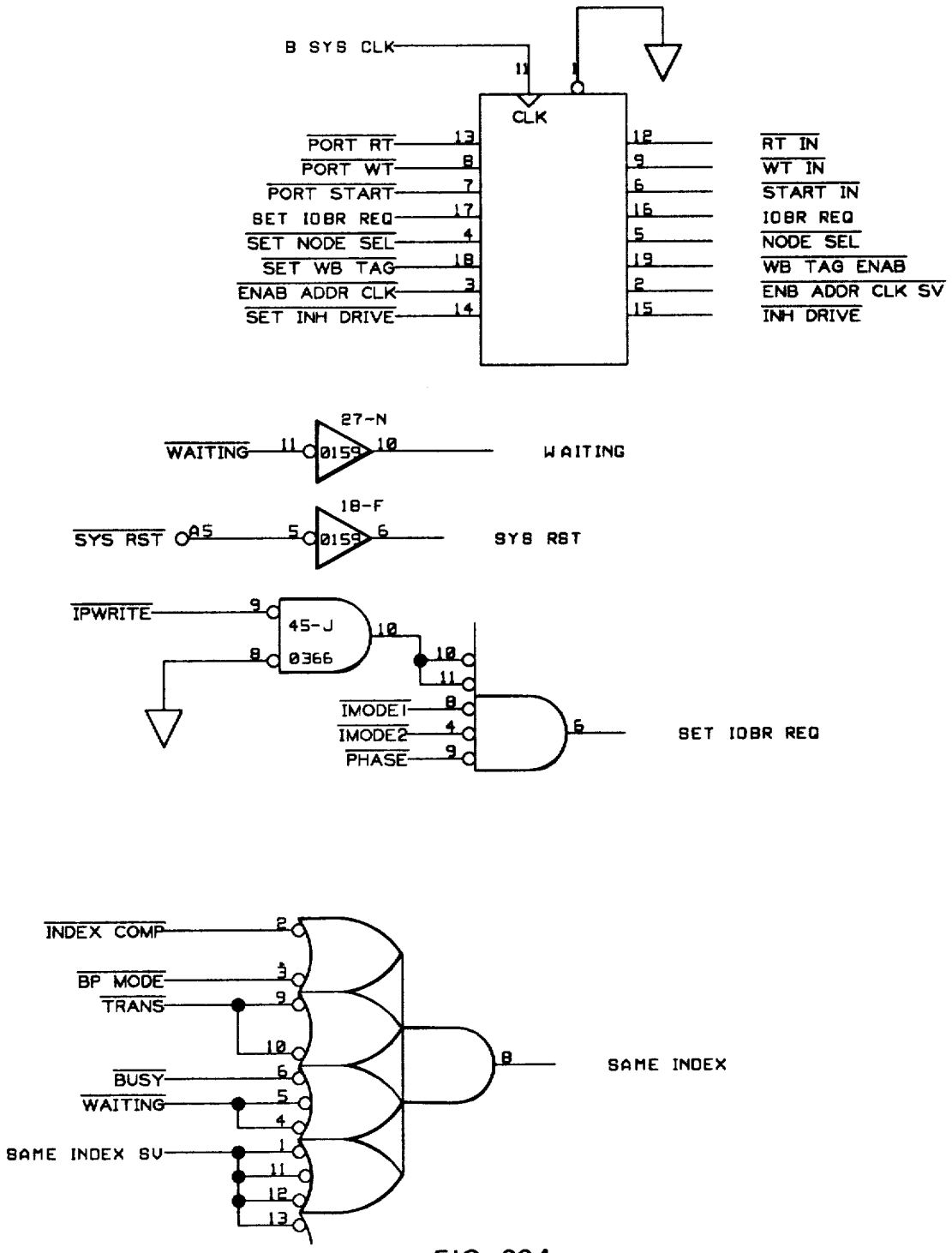
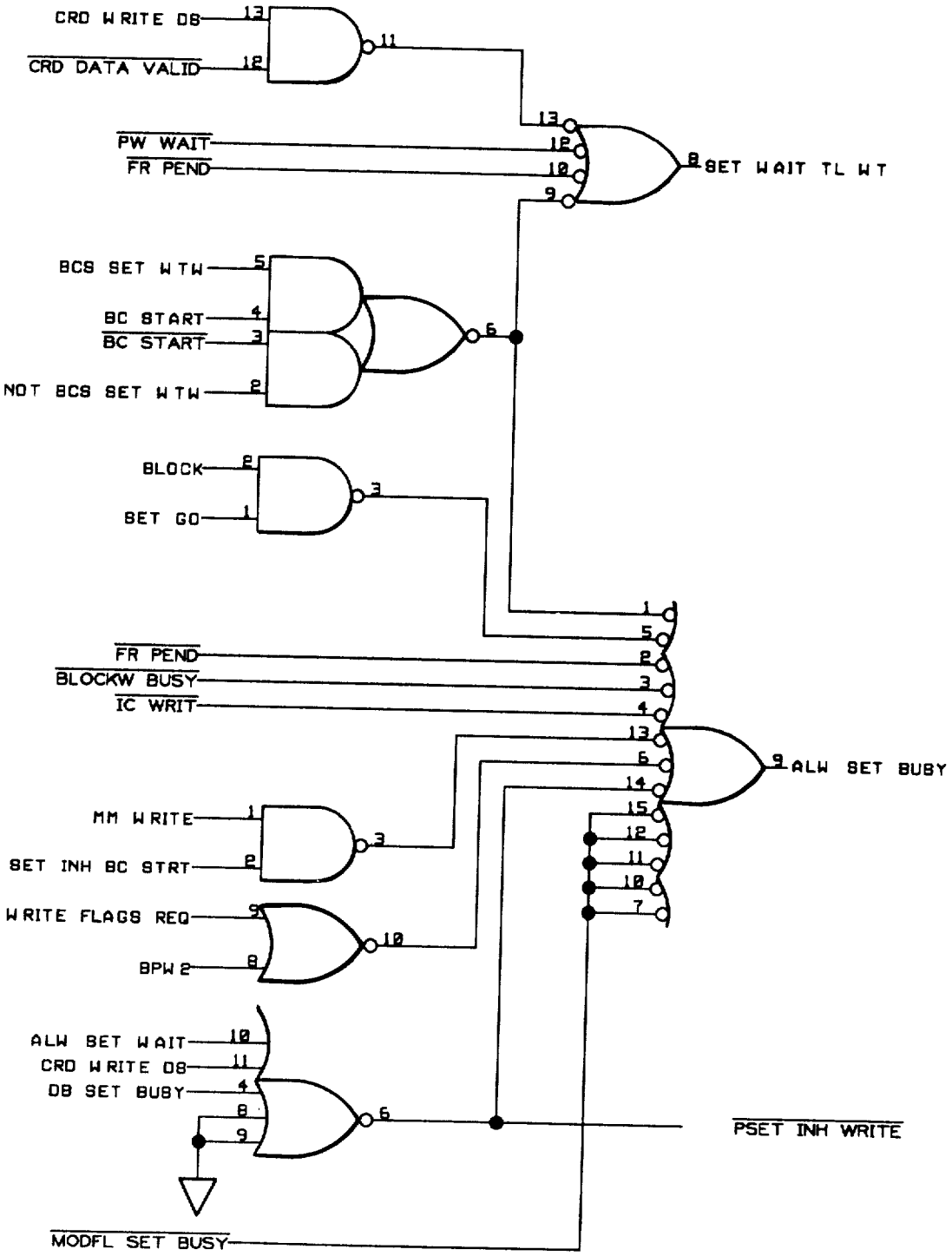

B SYS CLK
CLK
PORT RT
PORT WT
PORT START
SET IOBR REQ
SET NODE SEL
SET WB TAG
ENAB ADDR CLK
SET INH DRIVE
RT IN
WT IN
START IN
IOBR REQ
NODE SEL
WB TAG ENAB
ENB ADDR CLK SV
INH DRIVE
27-N
WAITING
WAITING
18-F
SYS RST
SYS RST
IPWRITE
45-J
0366
IMODE1
IMODE2
PHASE
SET IOBR REQ
INDEX COMP
BP MODE
TRANS
BUSY
WAITING
SAME INDEX SV
SAME INDEX

FIG. 29A



SYSTEM CACHE CONTROL LOGIC

FIG. 30

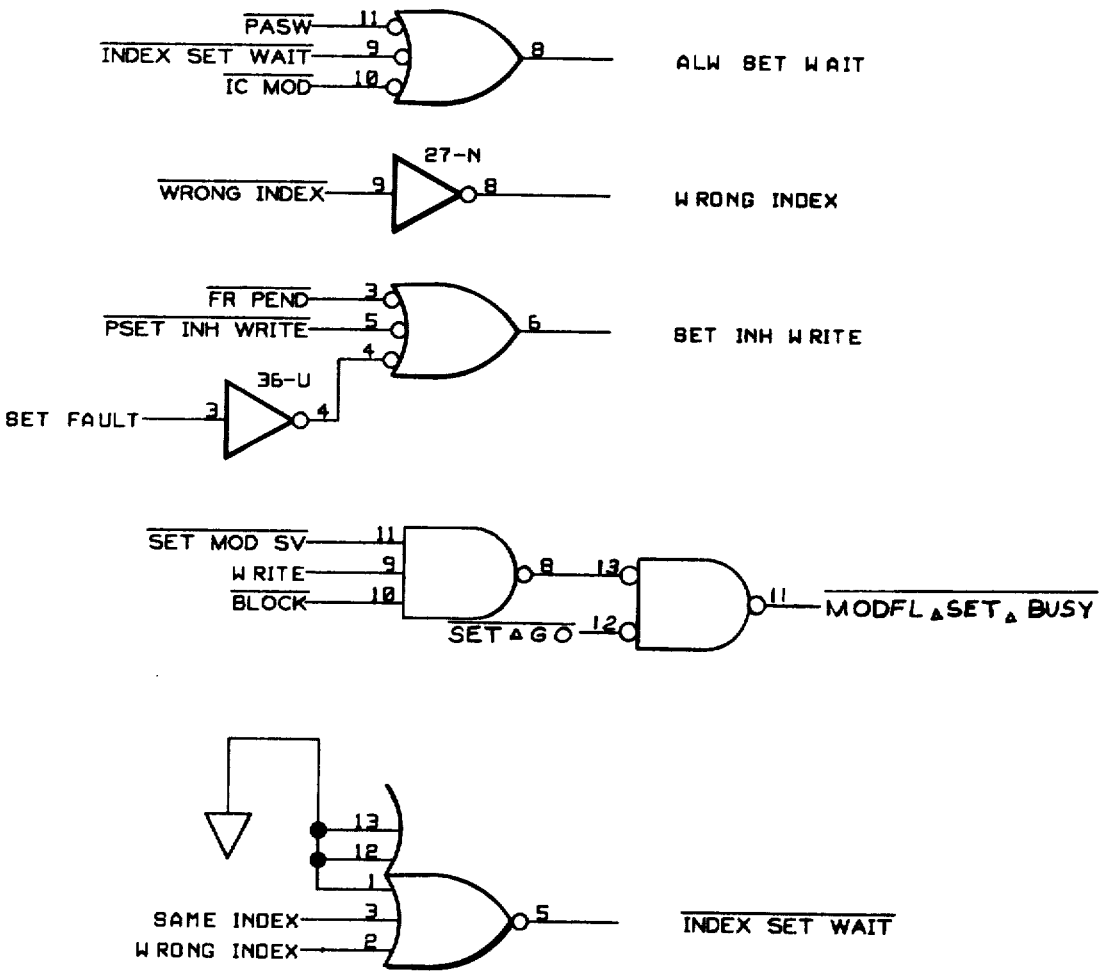
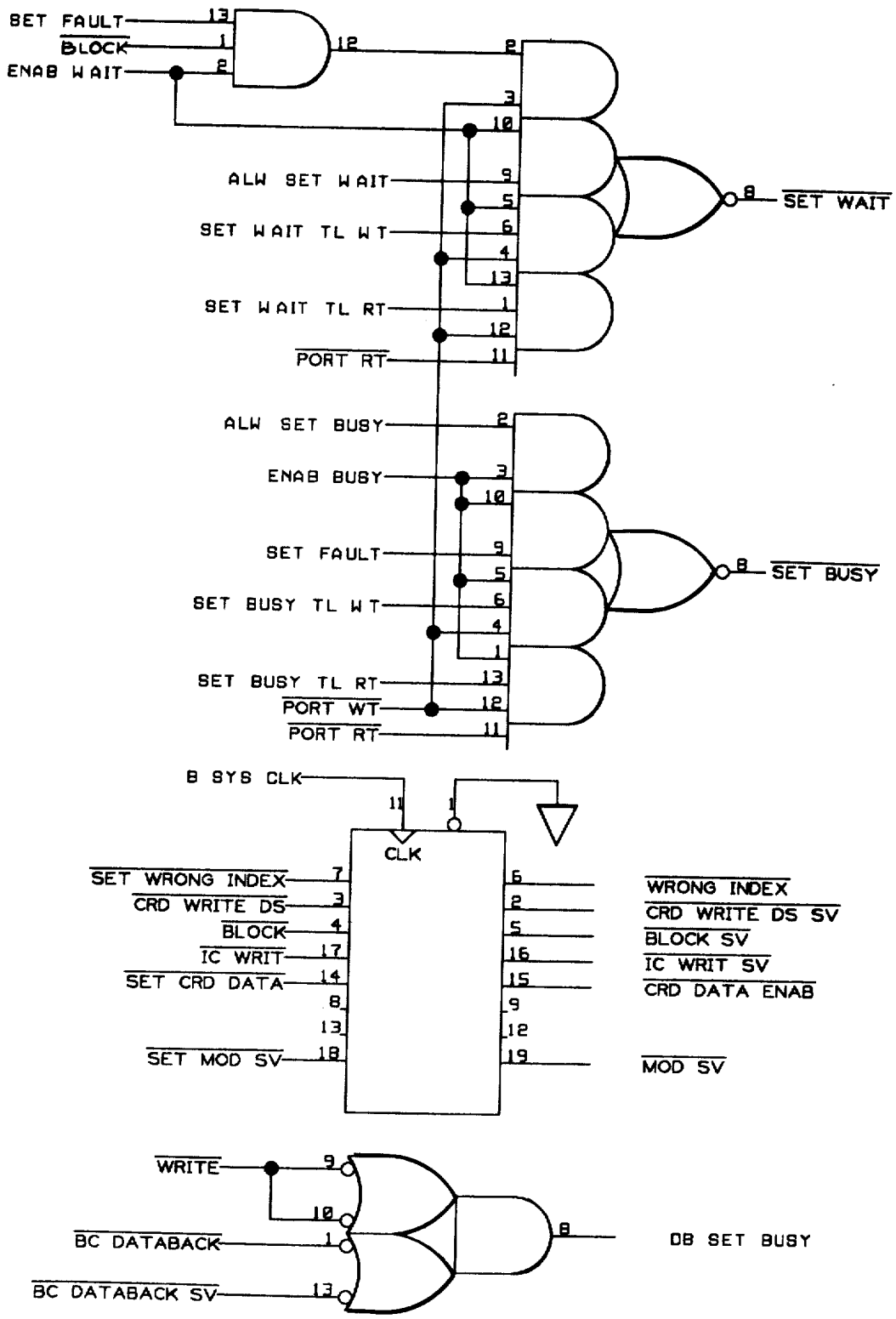

PASW
INDEX SET WAIT
IC MOD
ALW SET WAIT
27-N
WRONG INDEX
WRONG INDEX
FR PEND
PSET INH WRITE
36-U
SET FAULT
SET INH WRITE
SET MOD SV
WRITE
BLOCK
SET GO
MODFL SET BUSY
SAME INDEX
WRONG INDEX
INDEX SET WAIT

FIG. 30A



SYSTEM CACHE CONTROL LOGIC

FIG. 31

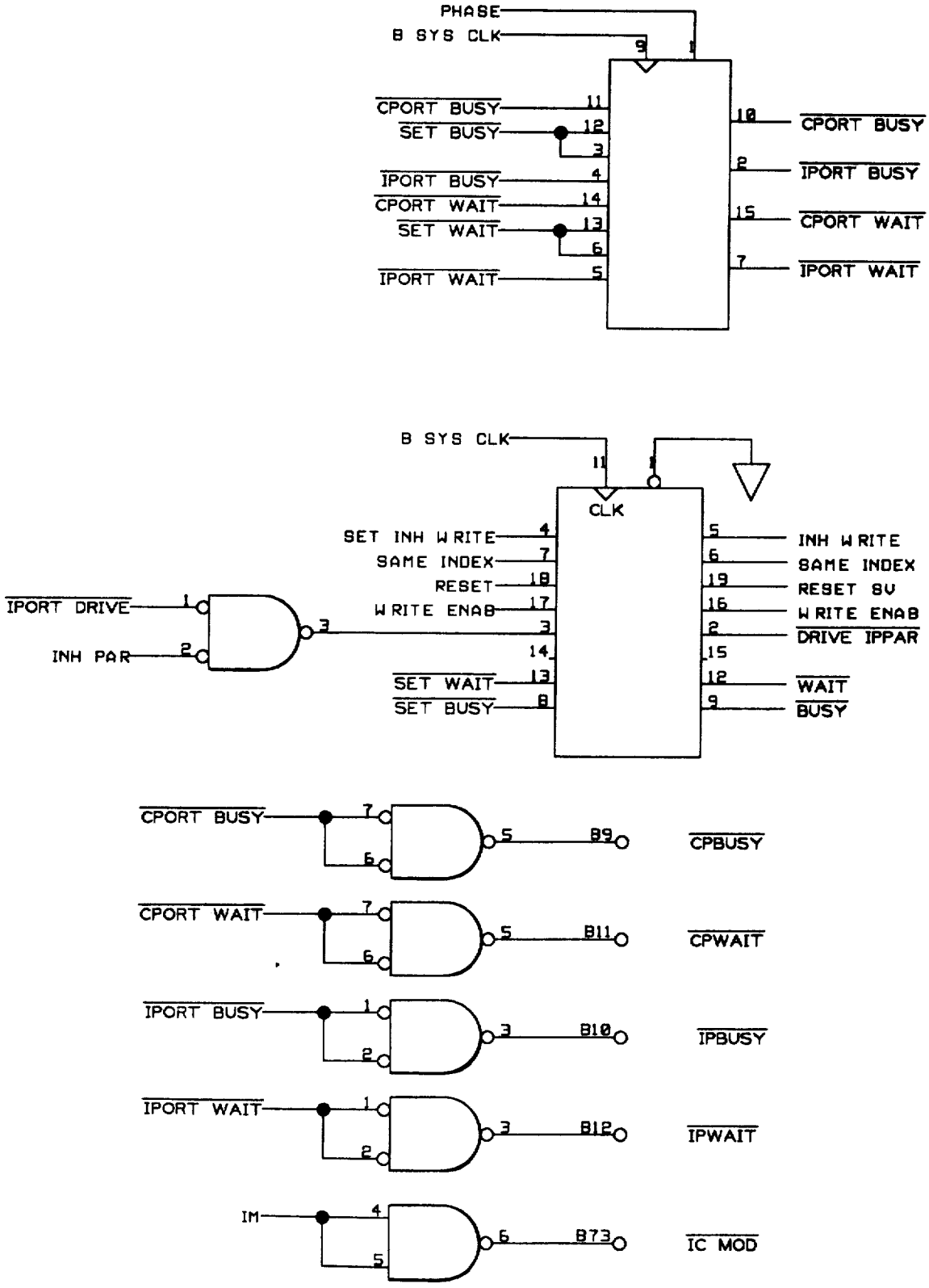
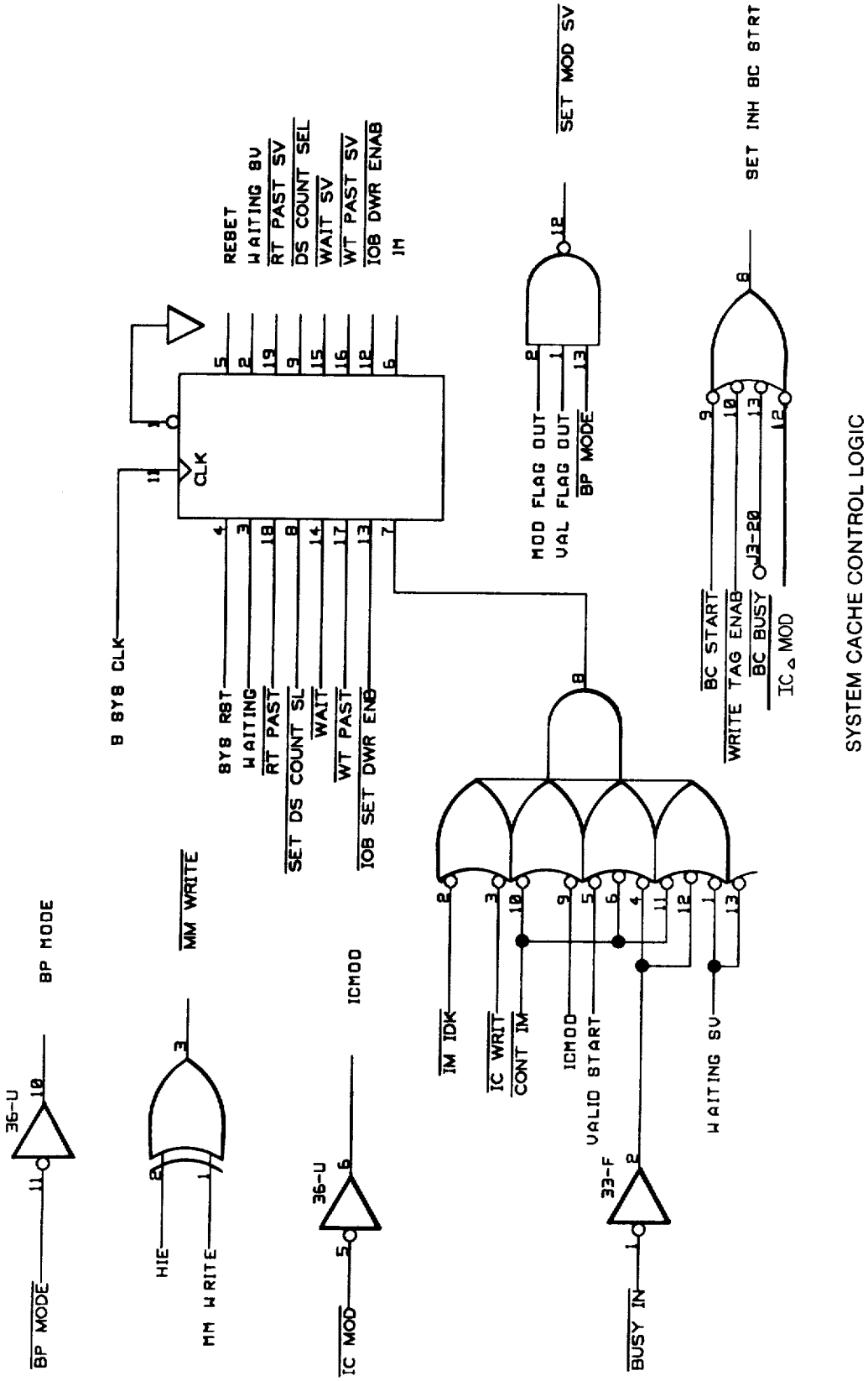

PHASE
B SYS CLK
CPORT BUSY
SET BUSY
IPORT BUSY
CPORT WAIT
SET WAIT
IPORT WAIT
CPORT BUSY
IPORT BUSY
CPORT WAIT
IPORT WAIT
B SYS CLK
CLK
SET INH WRITE
SAME INDEX
RESET
WRITE ENAB
SET WAIT
SET BUSY
INH WRITE
SAME INDEX
RESET SV
WRITE ENAB
DRIVE IPPAR
WAIT
BUSY
IPORT DRIVE
INH PAR
CPORT BUSY
B9
CPBUSY
CPORT WAIT
B11
CPWAIT
IPORT BUSY
B10
IPBUSY
IPORT WAIT
B12
IPWAIT
IM
B73
IC MOD

FIG. 31A



SYSTEM CACHE CONTROL LOGIC

FIG. 32

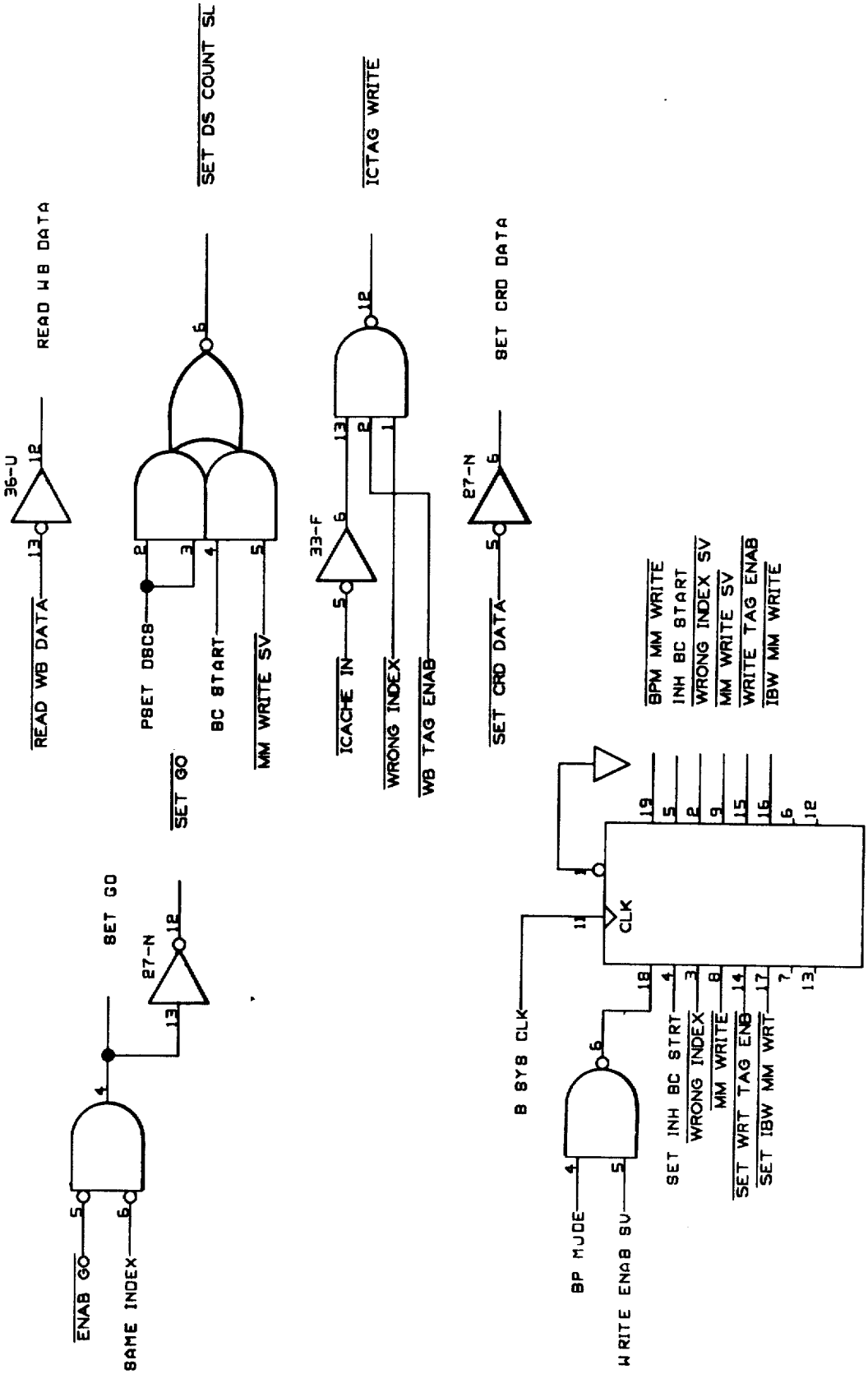
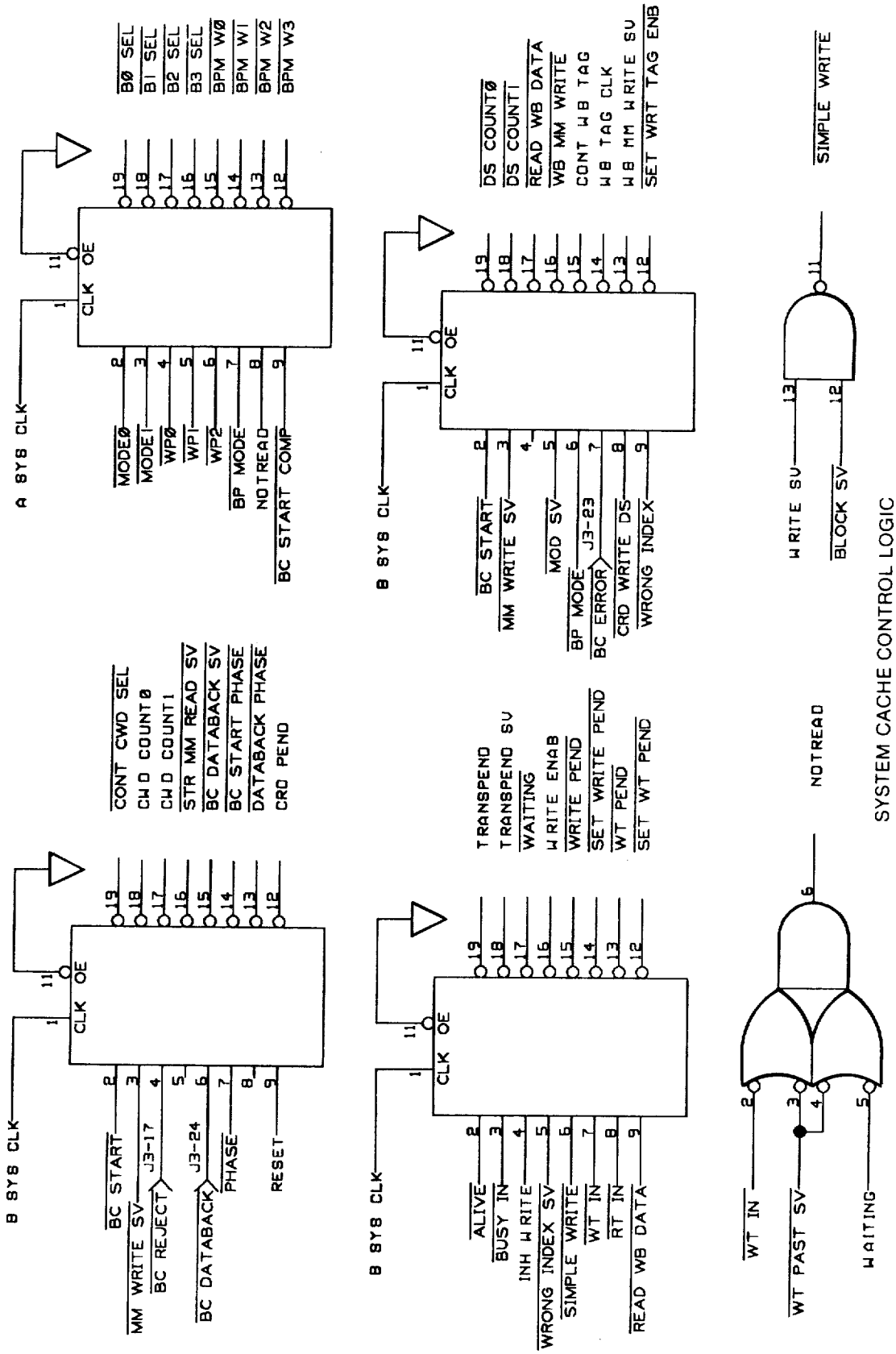
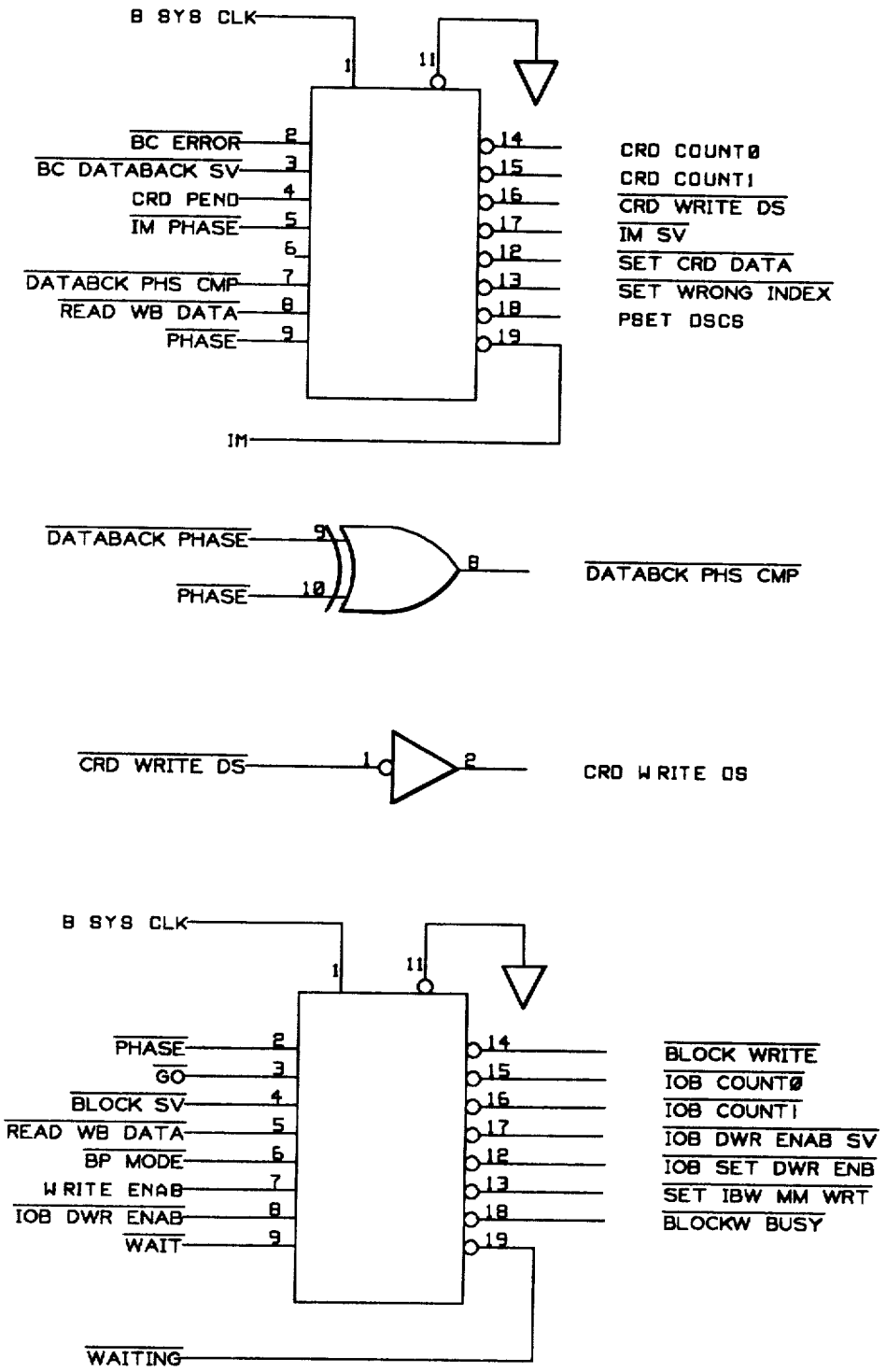

36-U
READ WB DATA
READ WB DATA
PSET DBCS
BC START
MM WRITE SV
SET DS COUNT SL
SET GO
33-F
ICACHE IN
WRONG INDEX
WB TAG ENAB
ICTAG WRITE
27-N
SET CRD DATA
SET CRD DATA
ENAB GO
BAME INDEX
27-N
SET GO
CLK
B BYB CLK
BP MODE
WRITE ENAB SV
SET INH BC STRT
WRONG INDEX
MM WRITE
SET WRT TAG ENB
SET IBW MM WRT
BPM MM WRITE
INH BC START
WRONG INDEX SV
MM WRITE SV
WRITE TAG ENAB
IBW MM WRITE

FIG. 32A



SYSTEM CACHE CONTROL LOGIC

FIG. 33



SYSTEM CACHE CONTROL LOGIC

FIG. 34

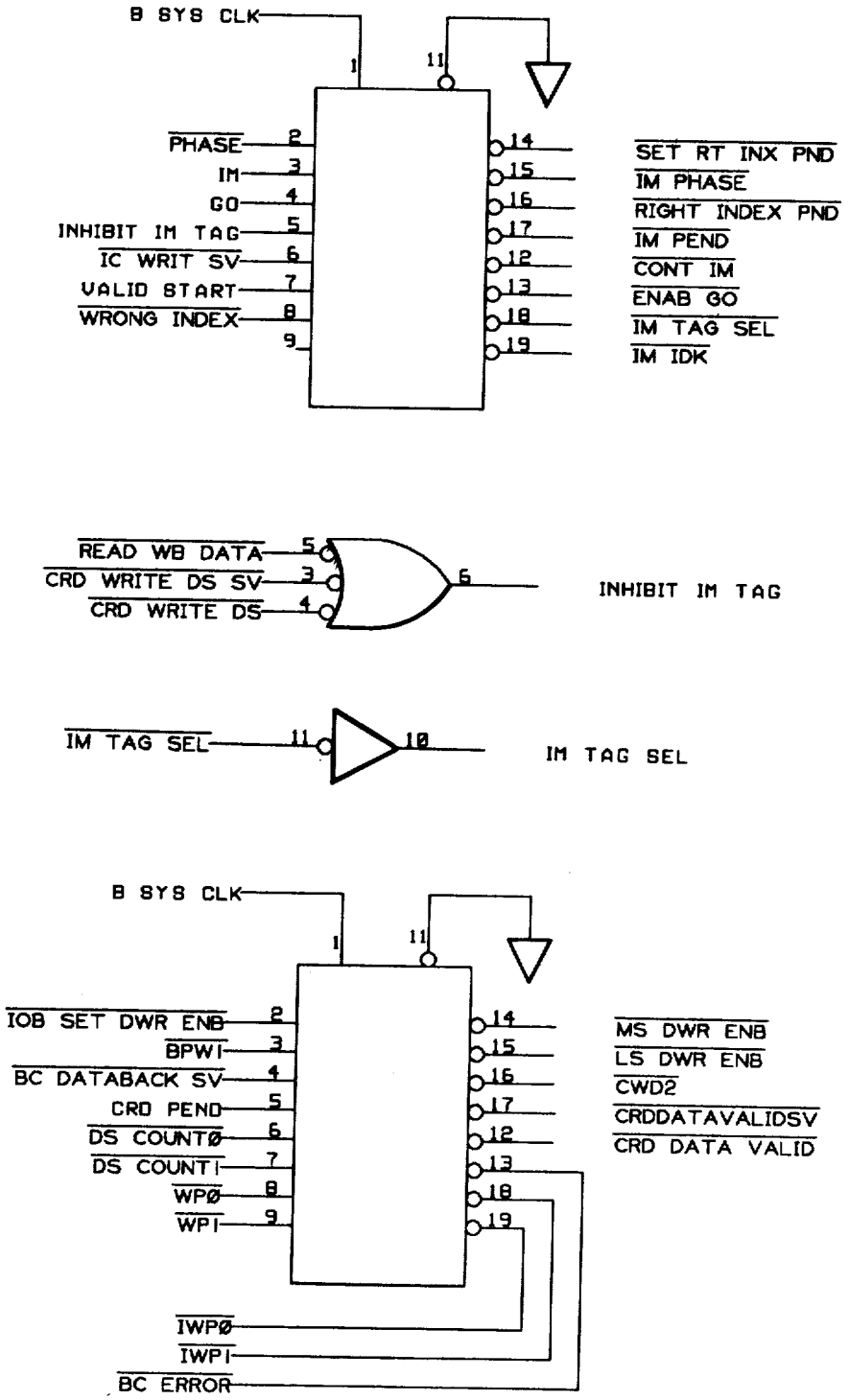
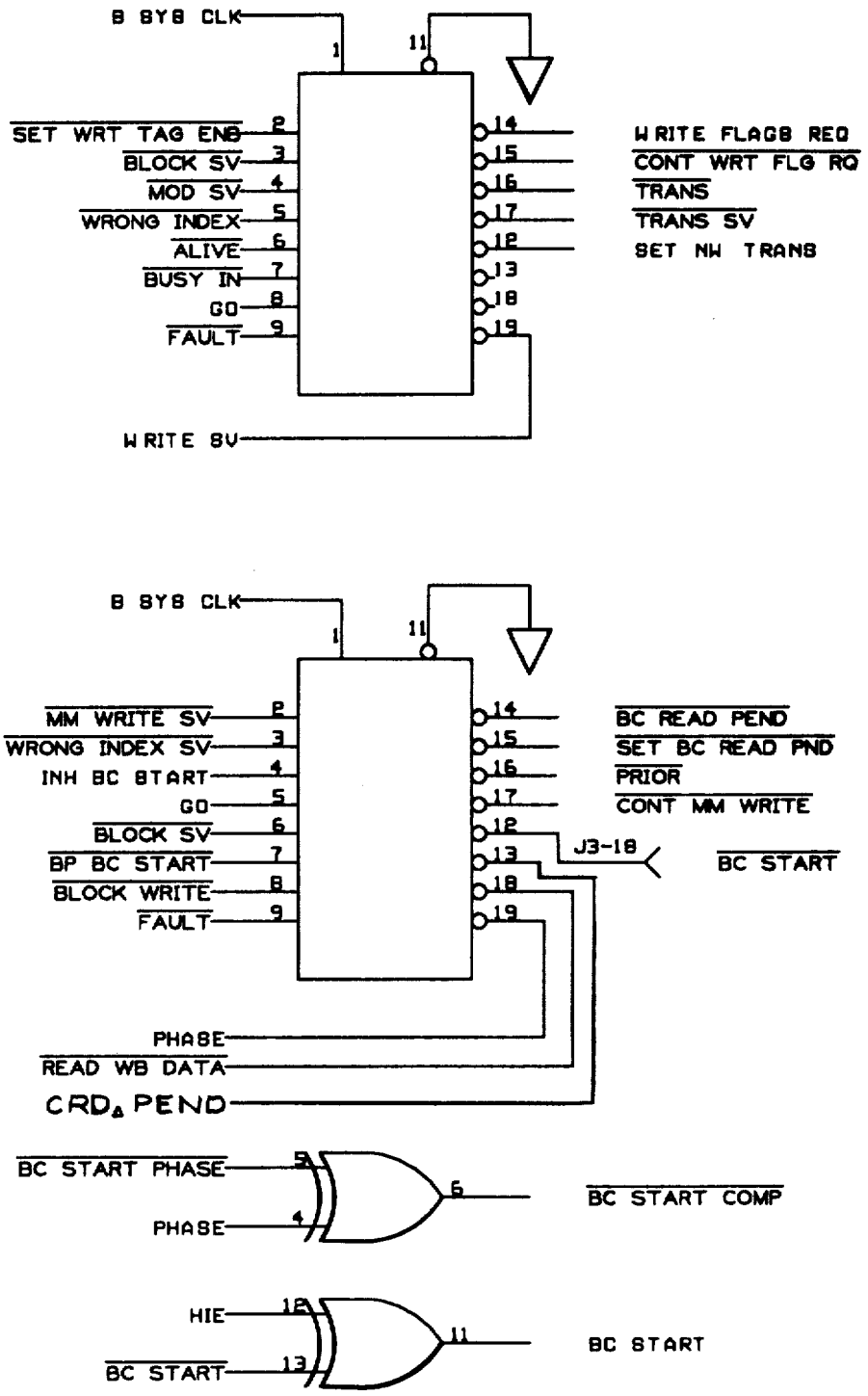
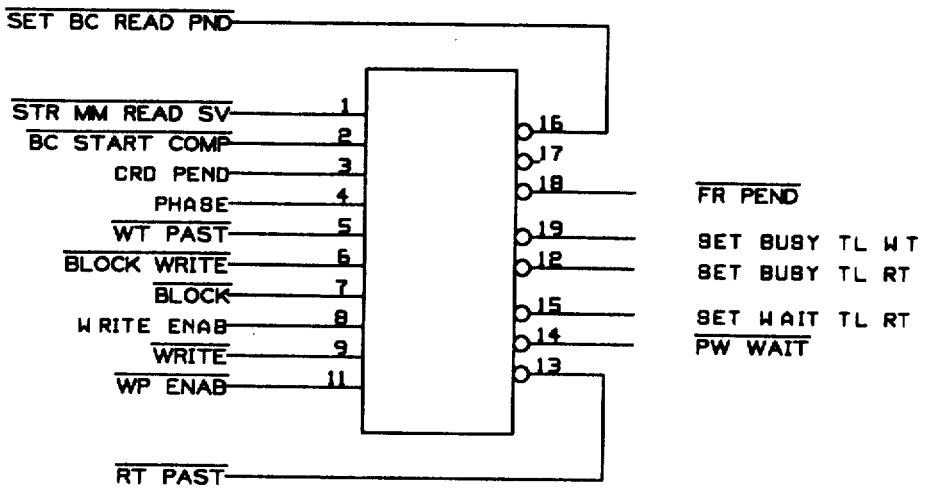

B SYS CLK
1
11
PHASE 2
IM 3
GO 4
INHIBIT IM TAG 5
IC WRIT SV 6
VALID START 7
WRONG INDEX 8
9
14 SET RT INX PND
15 IM PHASE
16 RIGHT INDEX PND
17 IM PEND
12 CONT IM
13 ENAB GO
18 IM TAG SEL
19 IM IDK
READ WB DATA 5
CRD WRITE DS SV 3
CRD WRITE DS 4
6 INHIBIT IM TAG
IM TAG SEL 11
10 IM TAG SEL
B SYS CLK
1
11
IOB SET DWR ENB 2
BPWI 3
BC DATABACK SV 4
CRD PEND 5
DS COUNTØ 6
DS COUNTI 7
WPØ 8
WPI 9
14 MS DWR ENB
15 LS DWR ENB
16 CWD2
17 CRDDATAVALIDSV
12 CRD DATA VALID
13
18
19
IWPØ
IWPI
BC ERROR

FIG. 34A



SYSTEM CACHE CONTROL LOGIC

FIG. 35


SET BC READ PND
STR MM READ SV
BC START COMP
CRD PEND
PHASE
WT PAST
BLOCK WRITE
BLOCK
WRITE ENAB
WRITE
WP ENAB
RT PAST
FR PEND
SET BUSY TL WT
SET BUSY TL RT
SET WAIT TL RT
PW WAIT

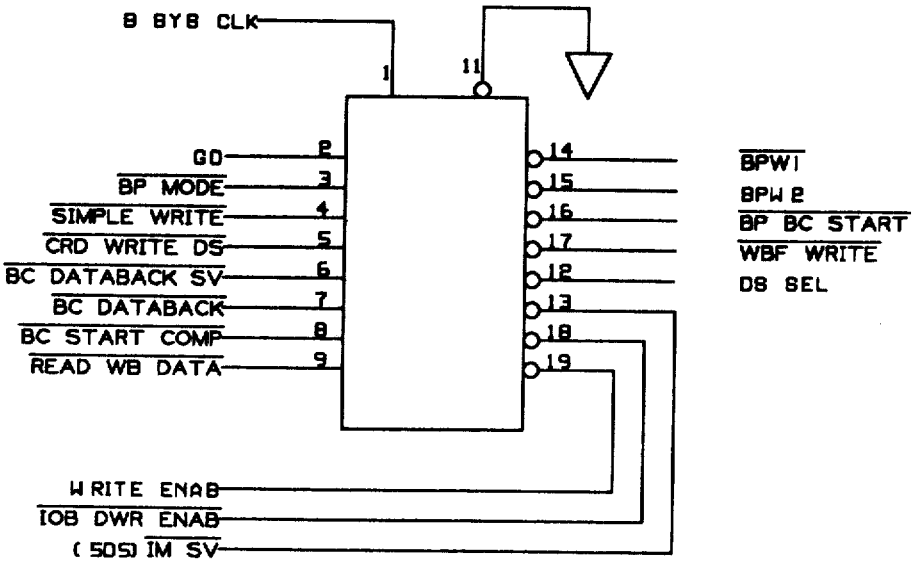
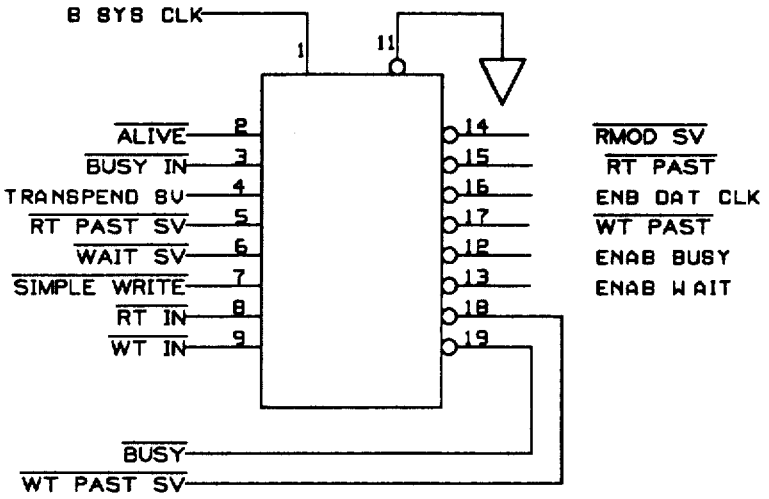
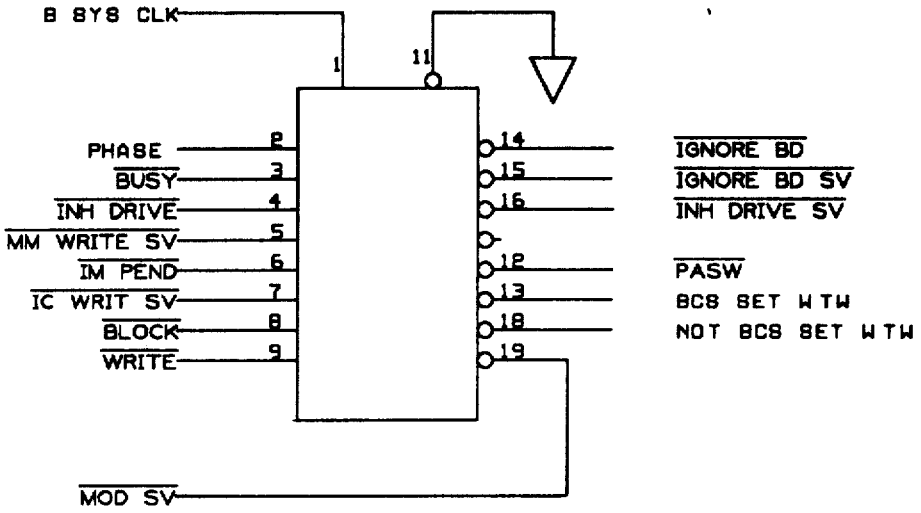

B BYB CLK
GO
BP MODE
SIMPLE WRITE
CRD WRITE DS
BC DATABACK SV
BC DATABACK
BC START COMP
READ WB DATA
WRITE ENAB
IOB DWR ENAB
(505) IM SV
BPW1
BPW 2
BP BC START
WBF WRITE
DS SEL

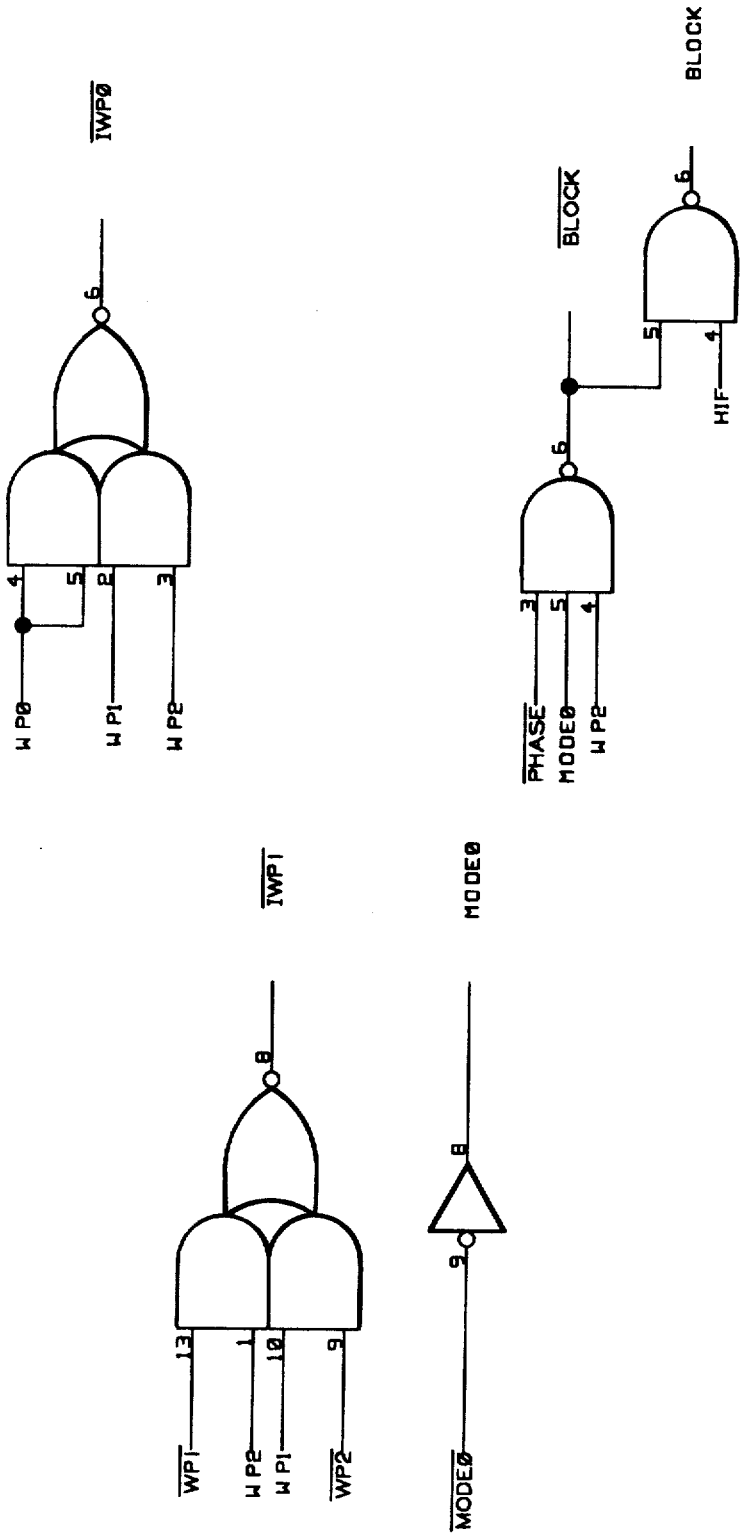
FIG. 35A





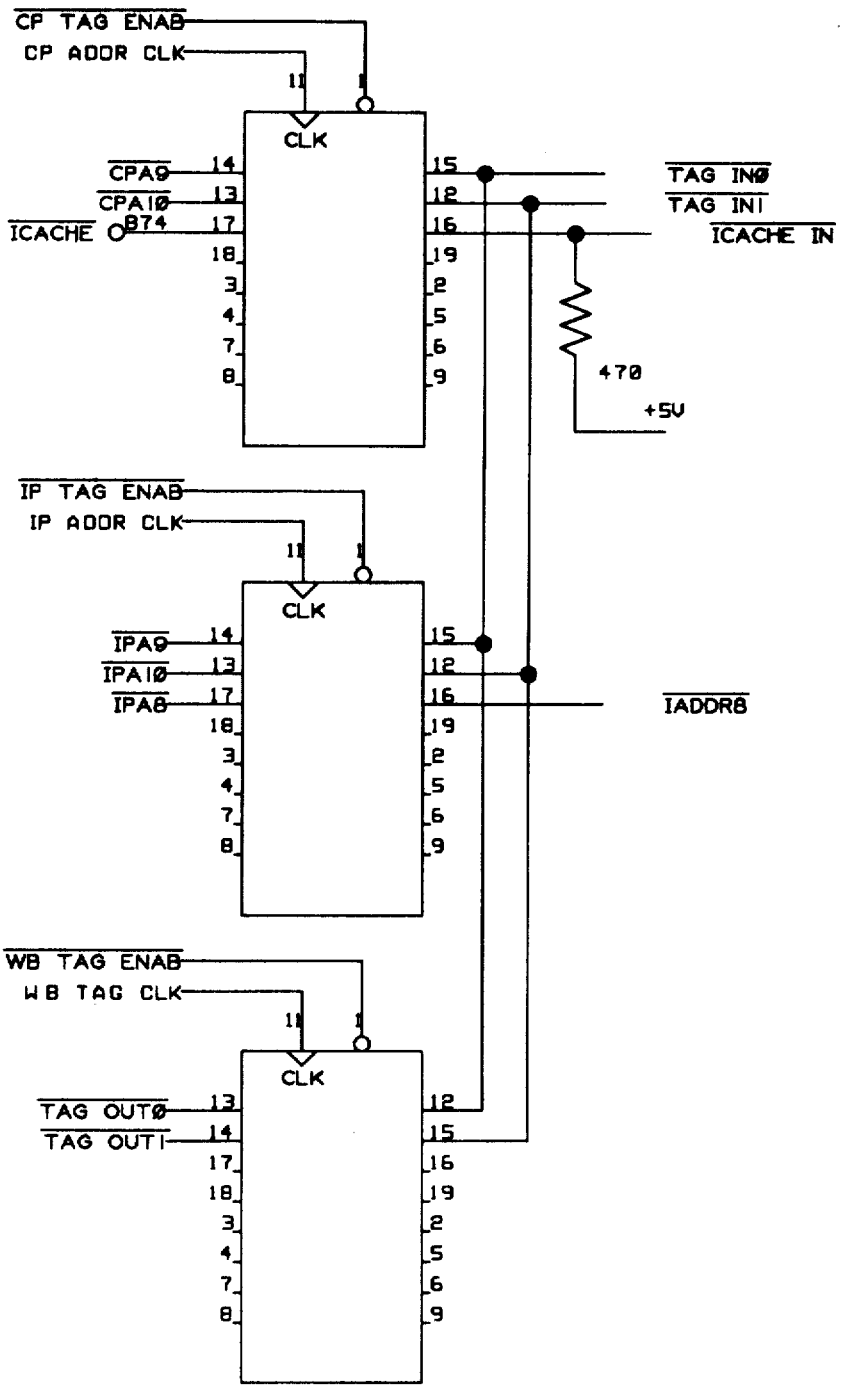
SYSTEM CACHE CONTROL LOGIC

FIG. 36



SYSTEM CACHE CONTROL LOGIC

FIG. 37



SYSTEM CACHE CONTROL LOGIC

FIG. 38

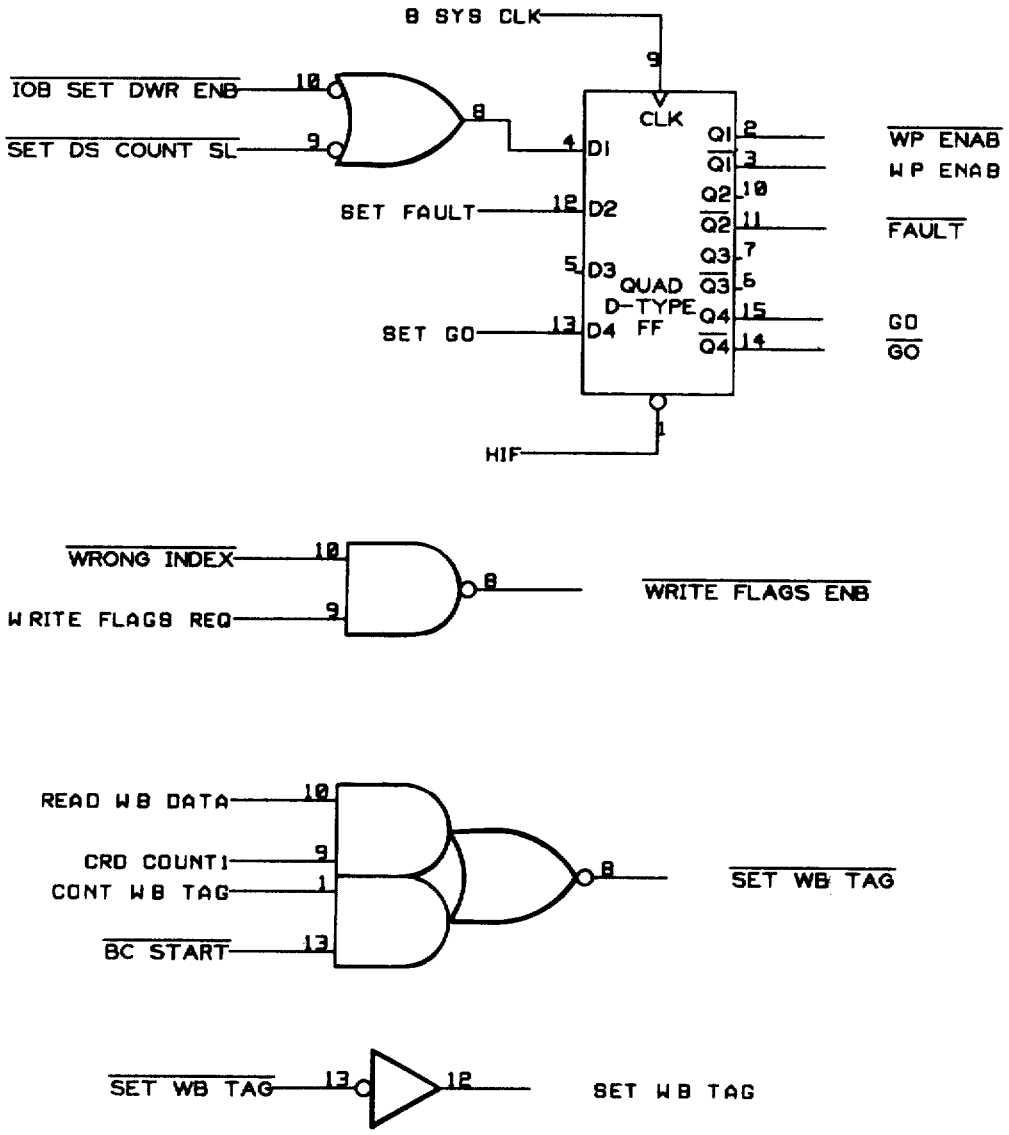
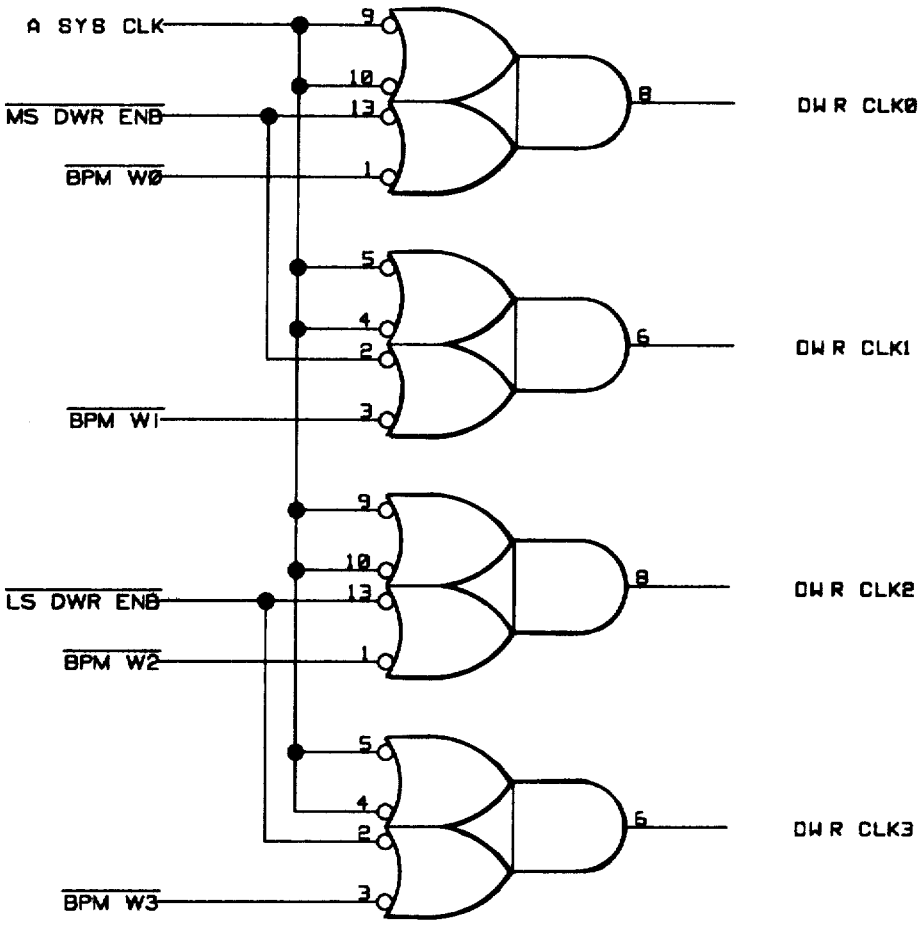
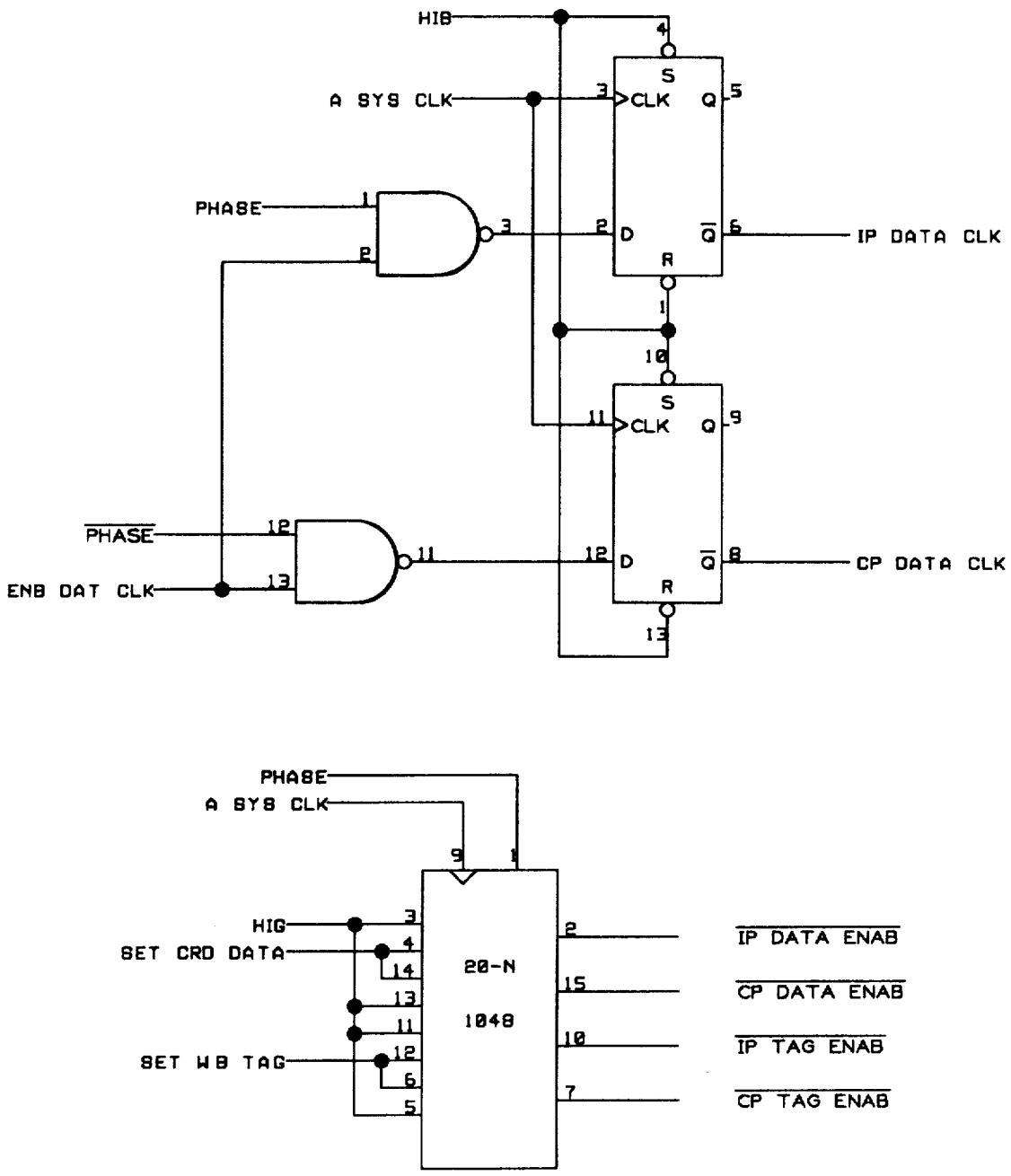

B SYS CLK
IOB SET DWR ENB
SET DS COUNT SL
SET FAULT
SET GO
HIF
CLK
D1
D2
D3
D4
QUAD
D-TYPE
FF
Q1
Q1
Q2
Q2
Q3
Q3
Q4
Q4
WP ENAB
WP ENAB
FAULT
GO
GO
WRONG INDEX
WRITE FLAGS REQ
WRITE FLAGS ENB
READ WB DATA
CRD COUNT1
CONT WB TAG
BC START
SET WB TAG
SET WB TAG
SET WB TAG

FIG. 38A



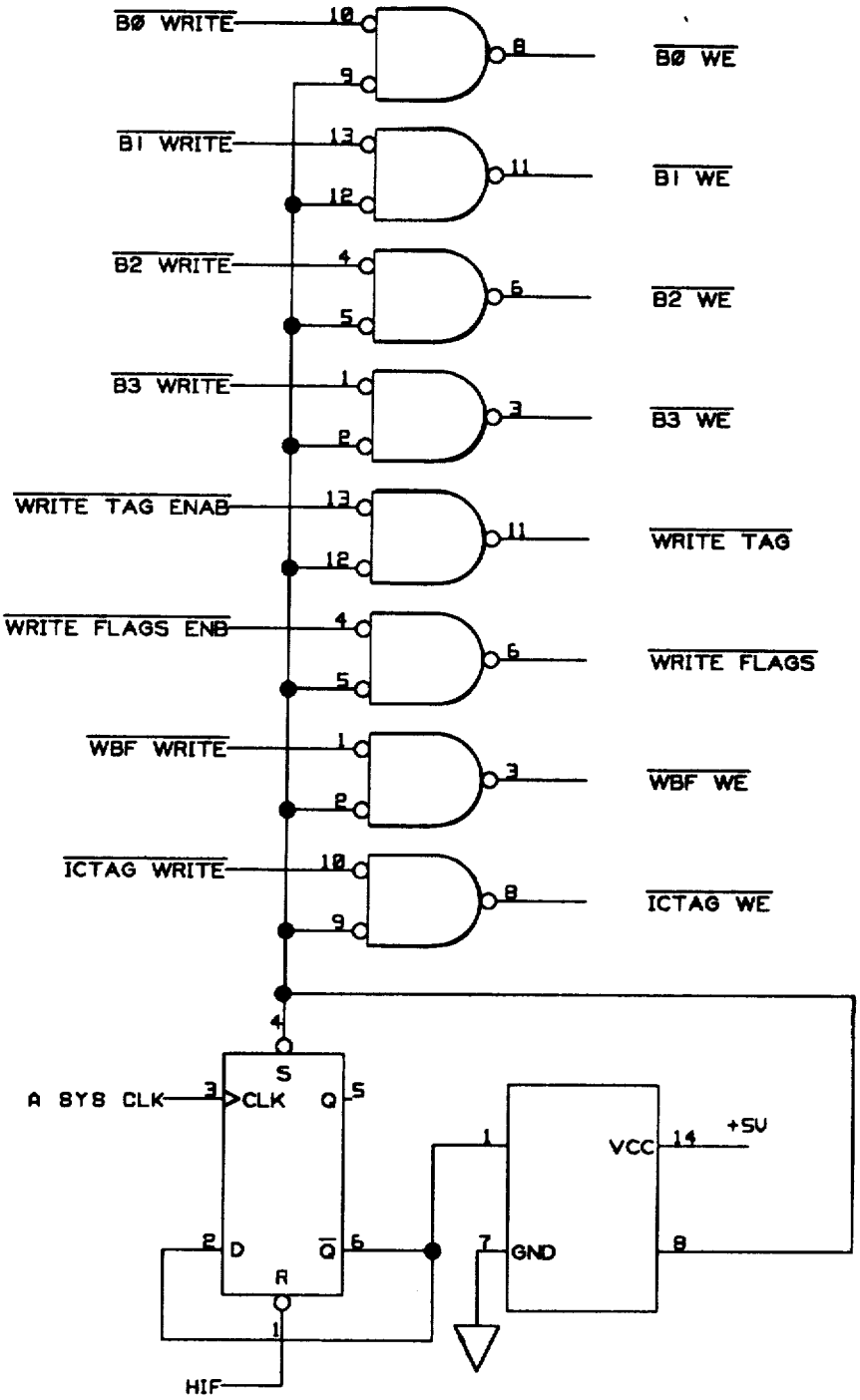
SYSTEM CACHE CONTROL LOGIC

FIG. 39



SYSTEM CACHE CONTROL LOGIC

FIG. 40



SYSTEM CACHE CONTROL LOGIC

FIG. 41

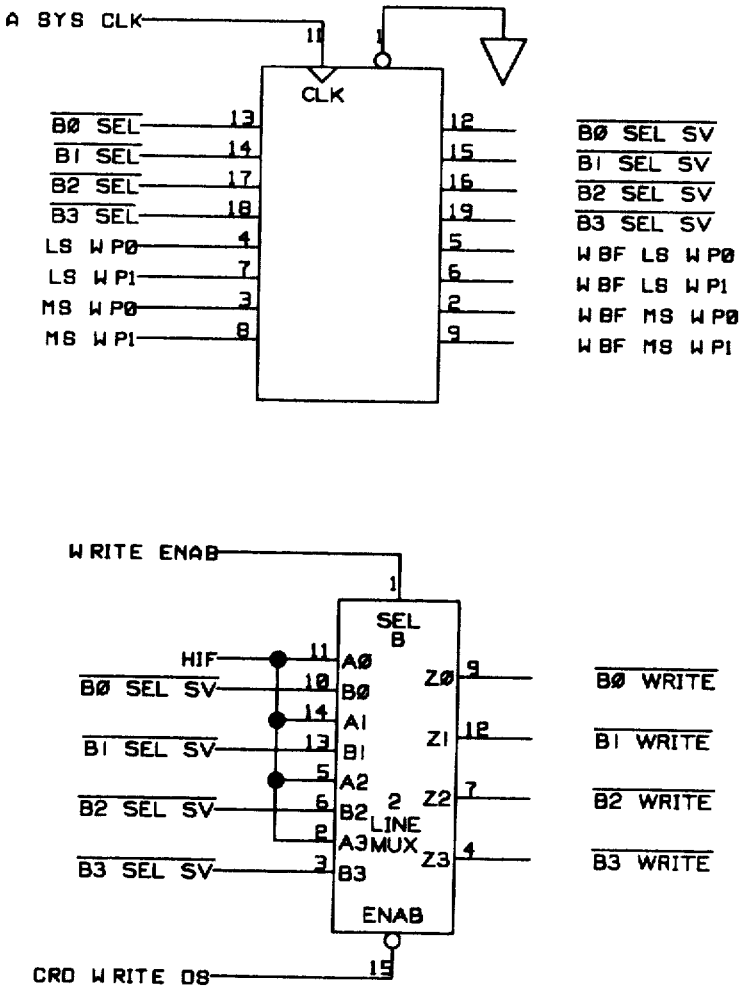
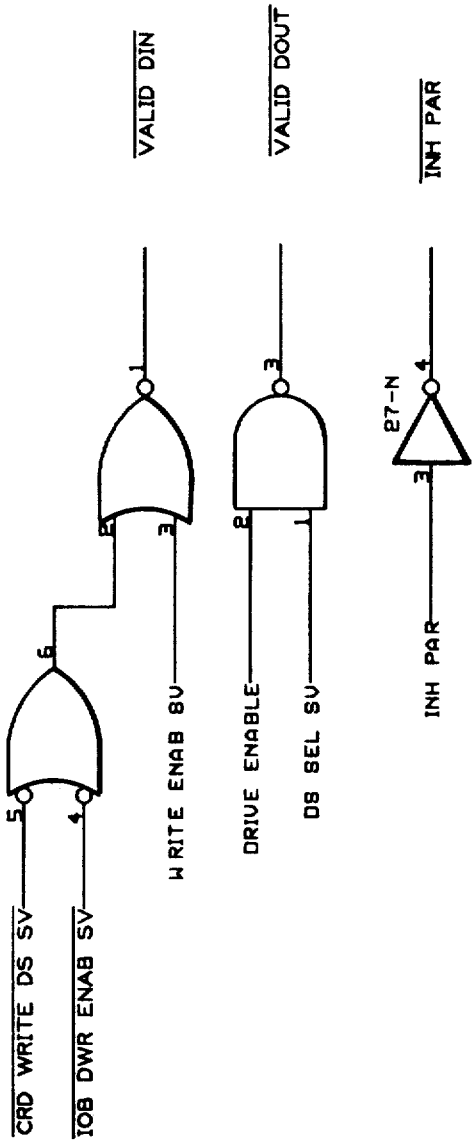
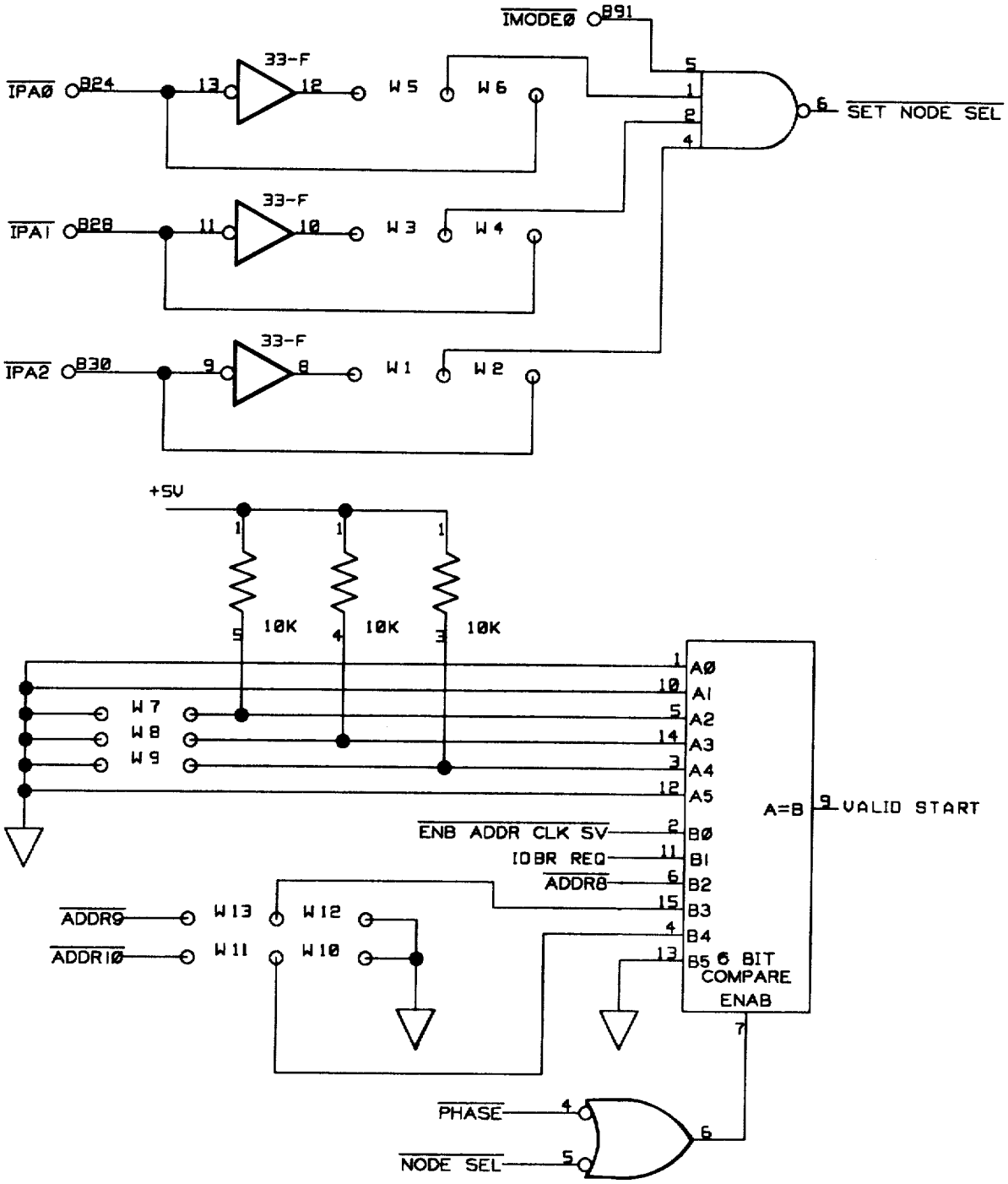

A SYS CLK
CLK
B0 SEL
B1 SEL
B2 SEL
B3 SEL
LS WP0
LS WP1
MS WP0
MS WP1
B0 SEL SV
B1 SEL SV
B2 SEL SV
B3 SEL SV
WBF LS WP0
WBF LS WP1
WBF MS WP0
WBF MS WP1
WRITE ENAB
SEL B
HIF
B0 SEL SV
B1 SEL SV
B2 SEL SV
B3 SEL SV
A0
B0
A1
B1
A2
B2
A3
B3
2 LINE MUX
Z0
Z1
Z2
Z3
B0 WRITE
B1 WRITE
B2 WRITE
B3 WRITE
ENAB
CRD WRITE DS

FIG. 41A



SYSTEM CACHE CONTROL LOGIC

FIG. 42



SYSTEM CACHE CONTROL LOGIC

FIG. 43

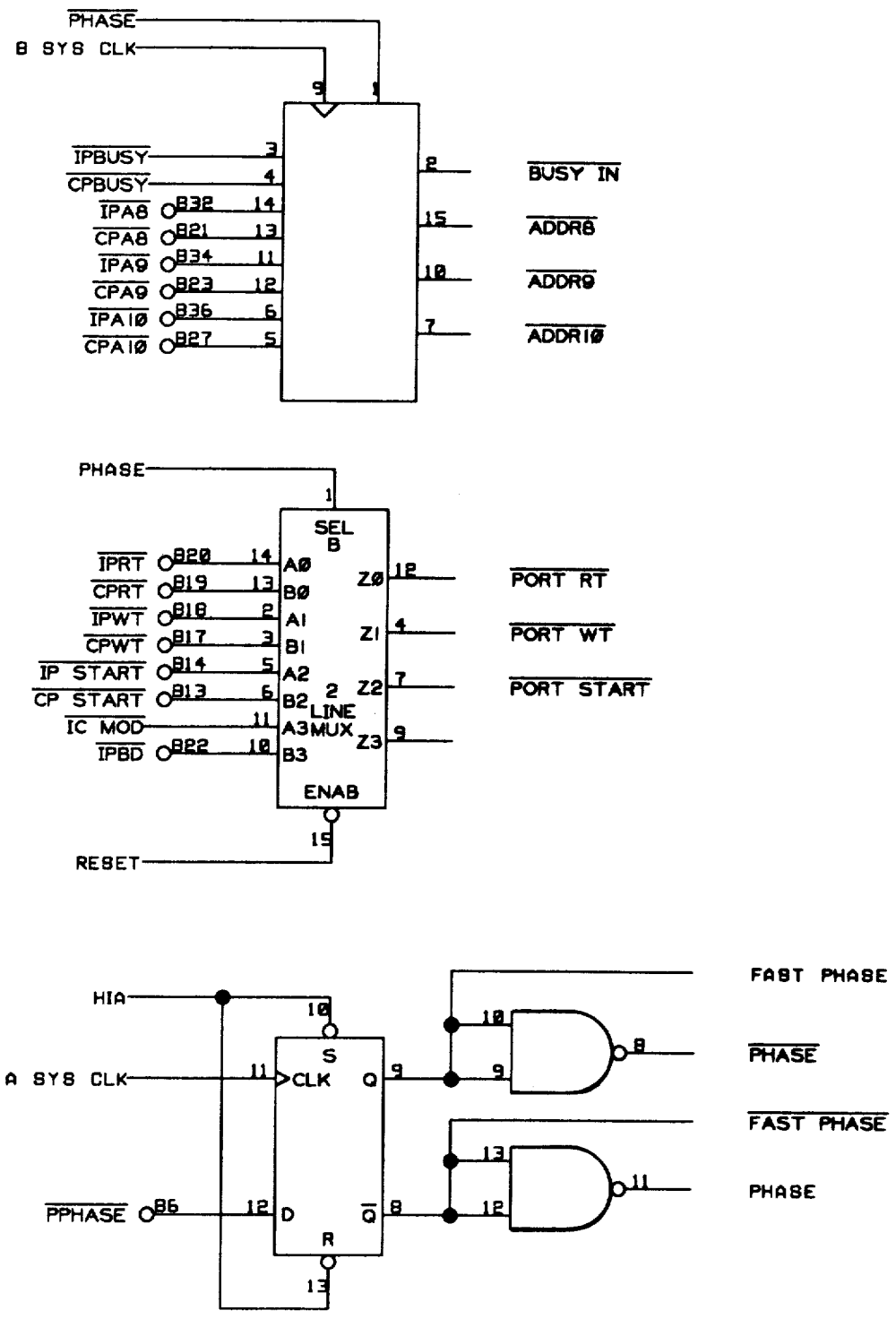

PHASE
B SYS CLK
IPBUSY
CPBUSY
IPA8
CPA8
IPA9
CPA9
IPA10
CPA10
BUSY IN
ADDR8
ADDR9
ADDR10
PHASE
SEL
B
IPRT
CPRT
IPWT
CPWT
IP START
CP START
IC MOD
IPBD
A0
B0
A1
B1
A2
B2
A3
B3
Z0
Z1
Z2
Z3
2 LINE MUX
ENAB
RESET
PORT RT
PORT WT
PORT START
HIA
A SYS CLK
PPHASE
S
CLK
D
Q
R
FAST PHASE
PHASE
FAST PHASE
PHASE

FIG. 43A

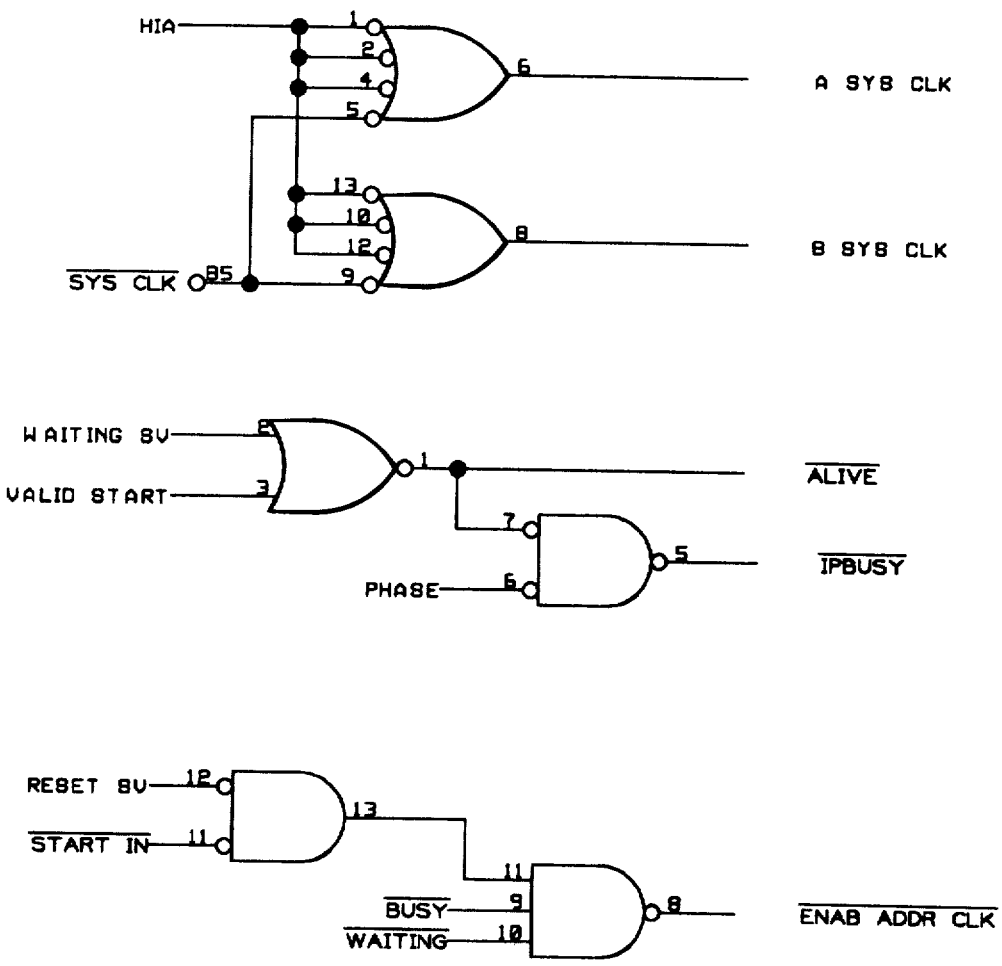
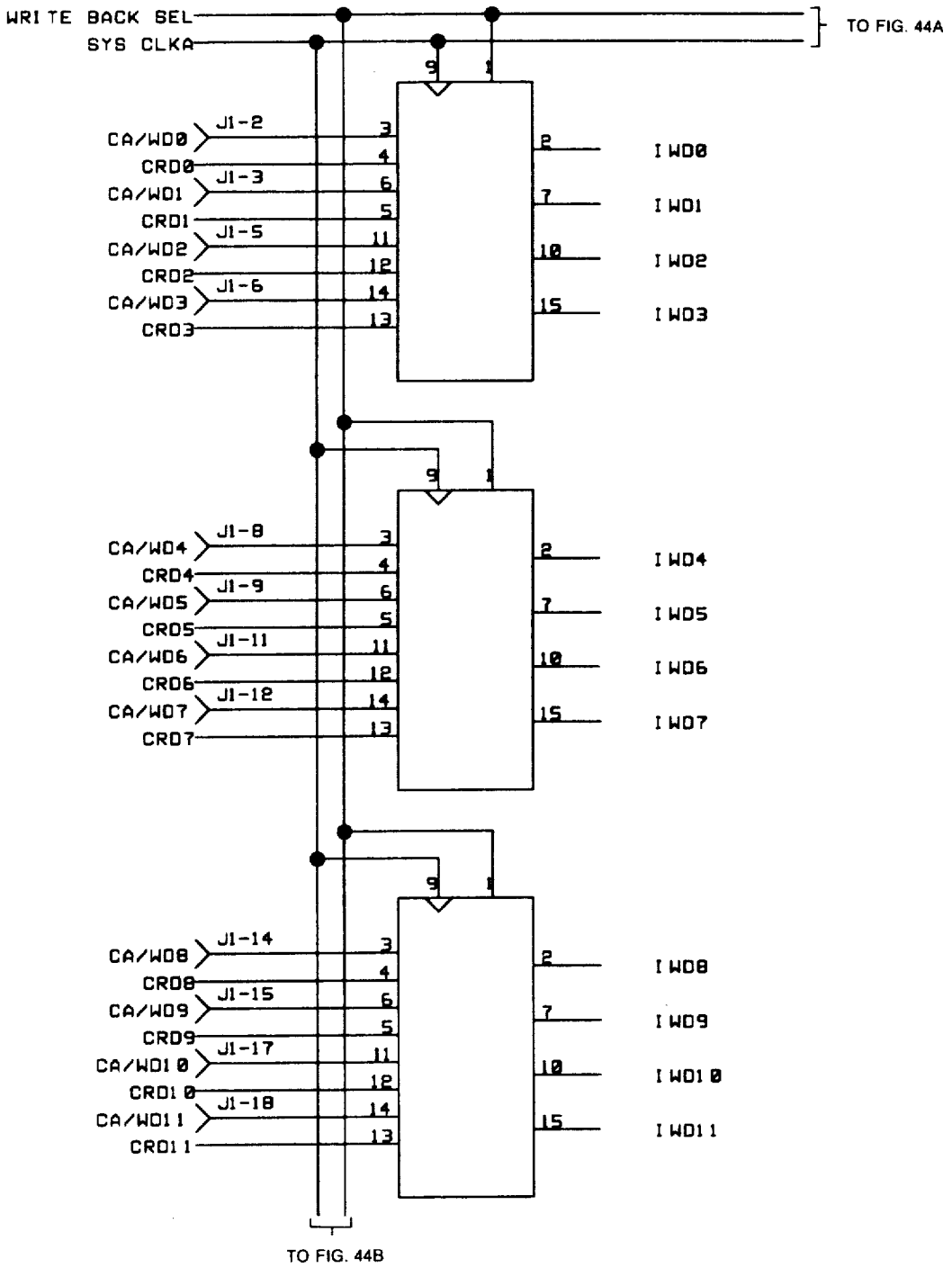

HIA
1
2
4
5
6
A SYS CLK
13
10
12
9
8
B SYS CLK
SYS CLK
B5
WAITING SU
VALID START
3
1
ALIVE
7
5
IPBUSY
PHASE
6
RESET SU
12
START IN
11
13
11
BUSY
9
WAITING
10
8
ENAB ADDR CLK

FIG. 43B



MUX STORE UNIT

FIG. 44

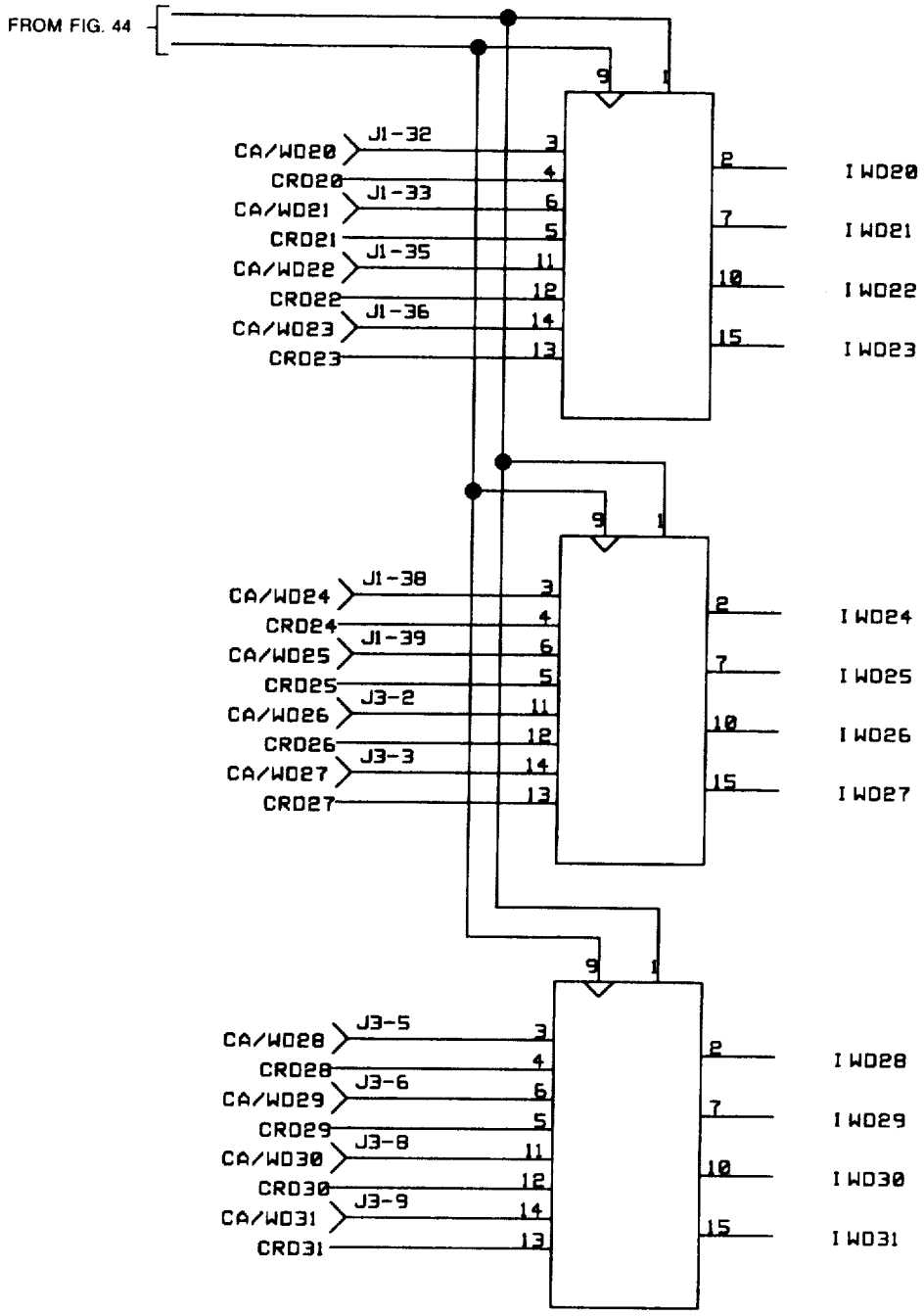

FROM FIG. 44
CA/WD20
J1-32
CRD20
CA/WD21
J1-33
CRD21
CA/WD22
J1-35
CRD22
CA/WD23
J1-36
CRD23
I WD20
I WD21
I WD22
I WD23
CA/WD24
J1-38
CRD24
CA/WD25
J1-39
CRD25
CA/WD26
J3-2
CRD26
CA/WD27
J3-3
CRD27
I WD24
I WD25
I WD26
I WD27
CA/WD28
J3-5
CRD28
CA/WD29
J3-6
CRD29
CA/WD30
J3-8
CRD30
CA/WD31
J3-9
CRD31
I WD28
I WD29
I WD30
I WD31

FIG. 44A

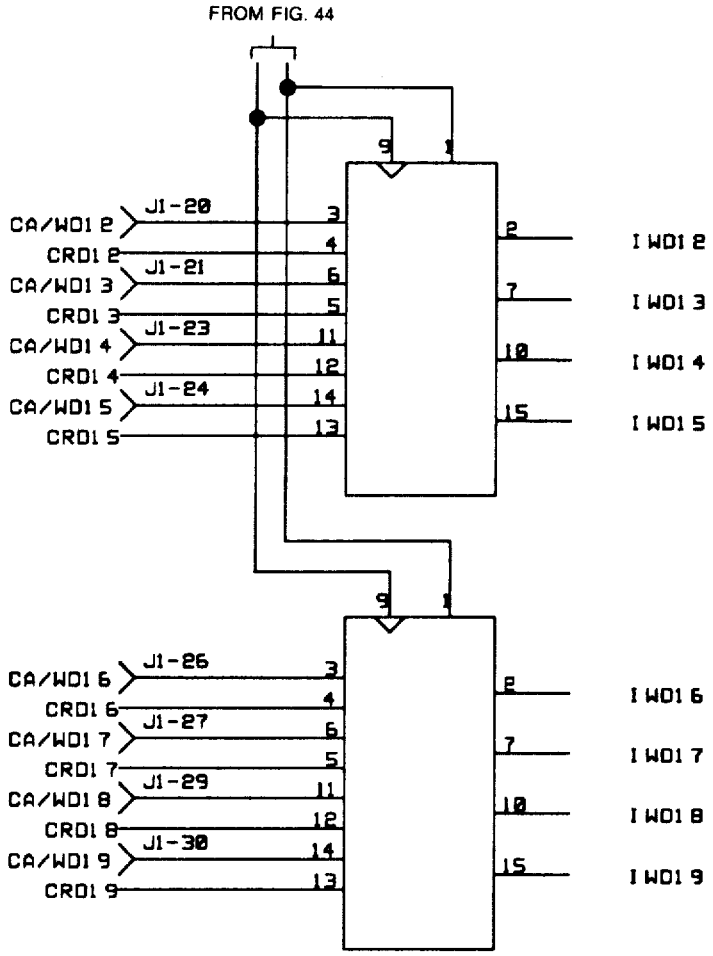
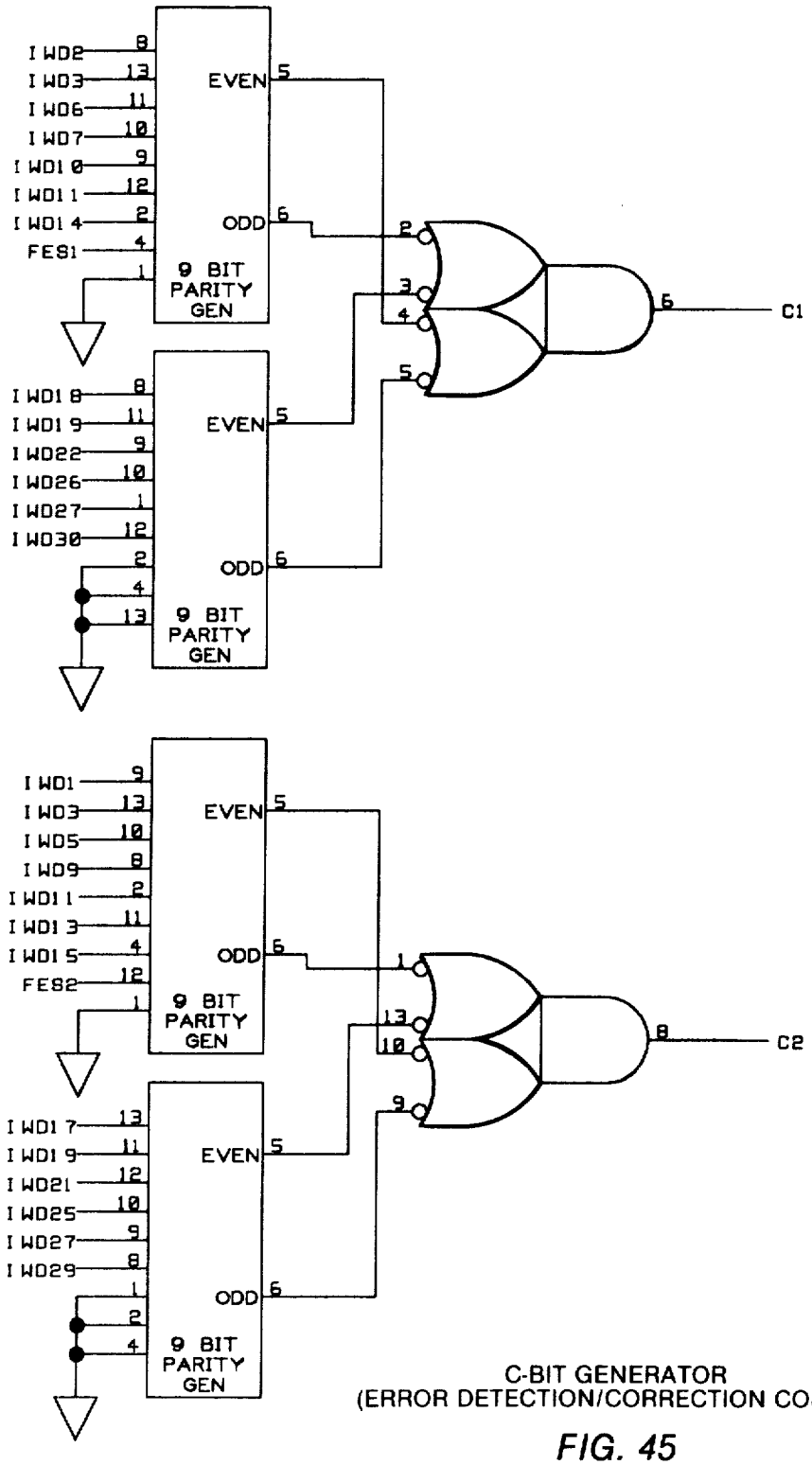

FROM FIG. 44
CA/WD1 2
J1-20
CRD1 2
CA/WD1 3
J1-21
CRD1 3
CA/WD1 4
J1-23
CRD1 4
CA/WD1 5
J1-24
CRD1 5
I WD1 2
I WD1 3
I WD1 4
I WD1 5
CA/WD1 6
J1-26
CRD1 6
CA/WD1 7
J1-27
CRD1 7
CA/WD1 8
J1-29
CRD1 8
CA/WD1 9
J1-30
CRD1 9
I WD1 6
I WD1 7
I WD1 8
I WD1 9

FIG. 44B



C-BIT GENERATOR
(ERROR DETECTION/CORRECTION CODE)

FIG. 45

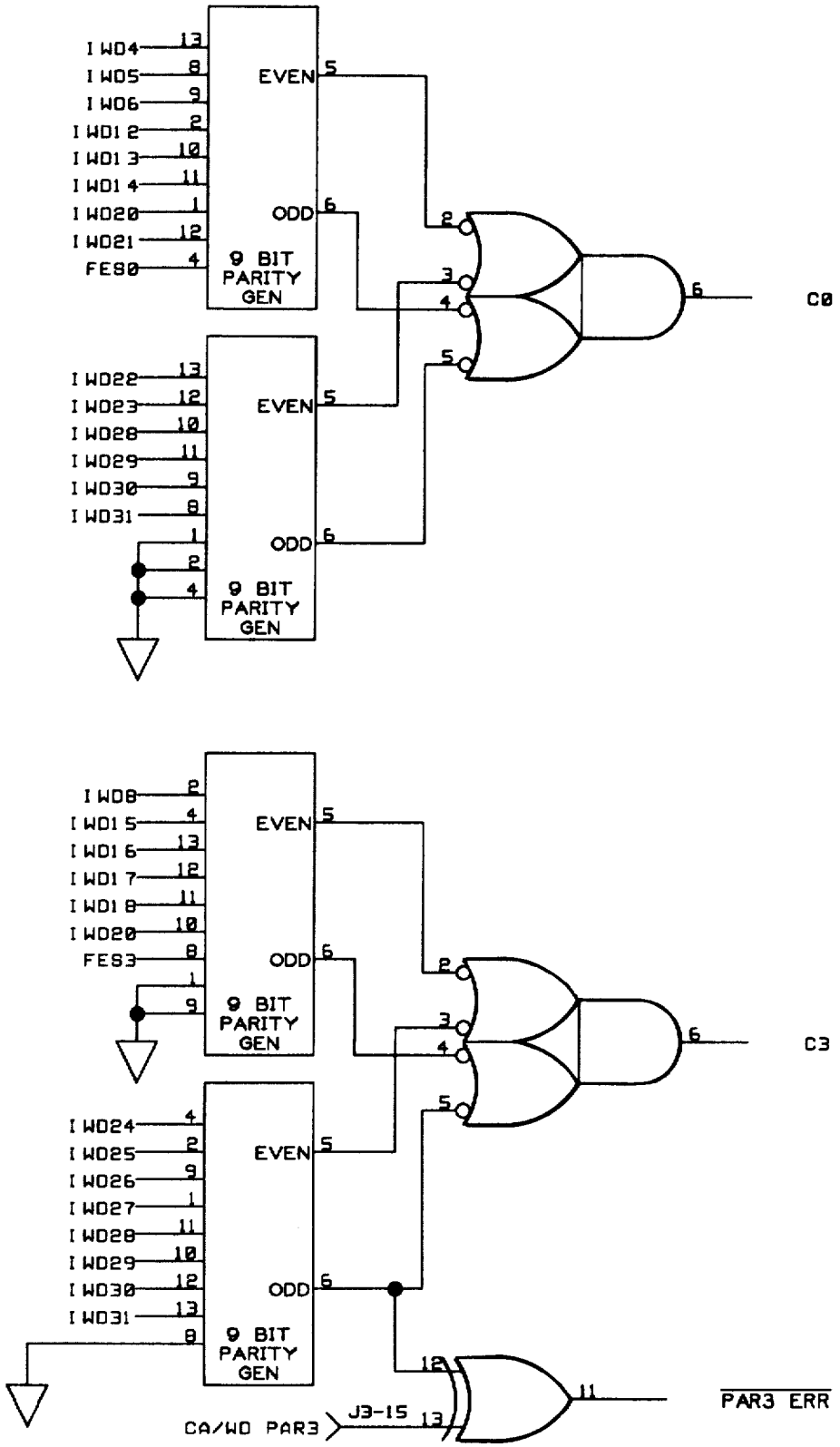

I WD4
I WD5
I WD6
I WD12
I WD13
I WD14
I WD20
I WD21
FES0
EVEN
ODD
9 BIT PARITY GEN
I WD22
I WD23
I WD28
I WD29
I WD30
I WD31
EVEN
ODD
9 BIT PARITY GEN
C0
I WD8
I WD15
I WD16
I WD17
I WD18
I WD20
FES3
EVEN
ODD
9 BIT PARITY GEN
I WD24
I WD25
I WD26
I WD27
I WD28
I WD29
I WD30
I WD31
EVEN
ODD
9 BIT PARITY GEN
C3
CA/WD PAR3
J3-15
PAR3 ERR

FIG. 45A

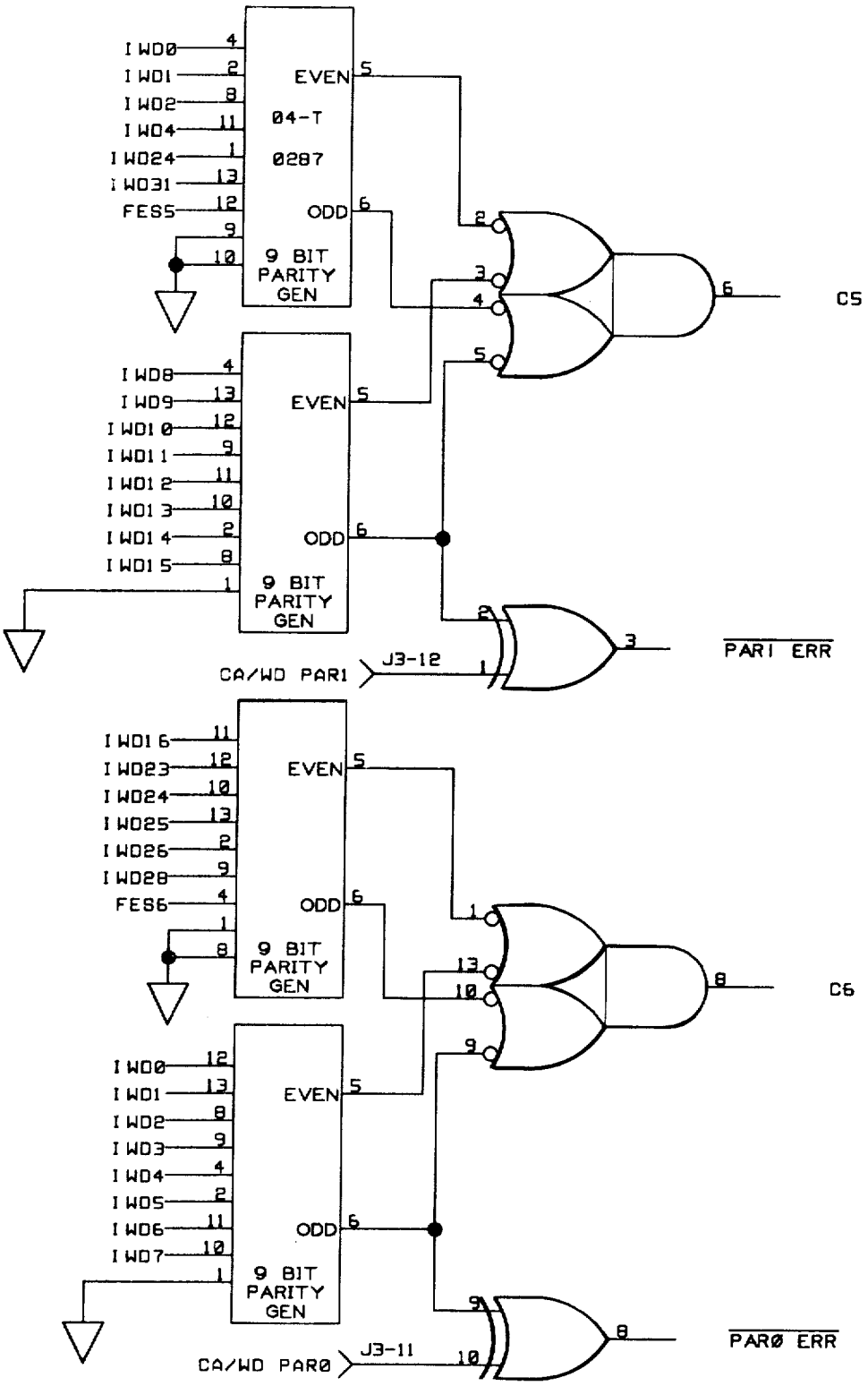

I WD0
I WD1
I WD2
I WD4
I WD24
I WD31
FES5
EVEN
04-T
0287
ODD
9 BIT PARITY GEN
C5
I WD8
I WD9
I WD10
I WD11
I WD12
I WD13
I WD14
I WD15
EVEN
ODD
9 BIT PARITY GEN
CA/WD PAR1
J3-12
PAR1 ERR
I WD16
I WD23
I WD24
I WD25
I WD26
I WD28
FES6
EVEN
ODD
9 BIT PARITY GEN
C6
I WD0
I WD1
I WD2
I WD3
I WD4
I WD5
I WD6
I WD7
EVEN
ODD
9 BIT PARITY GEN
CA/WD PAR0
J3-11
PAR0 ERR

FIG. 45B

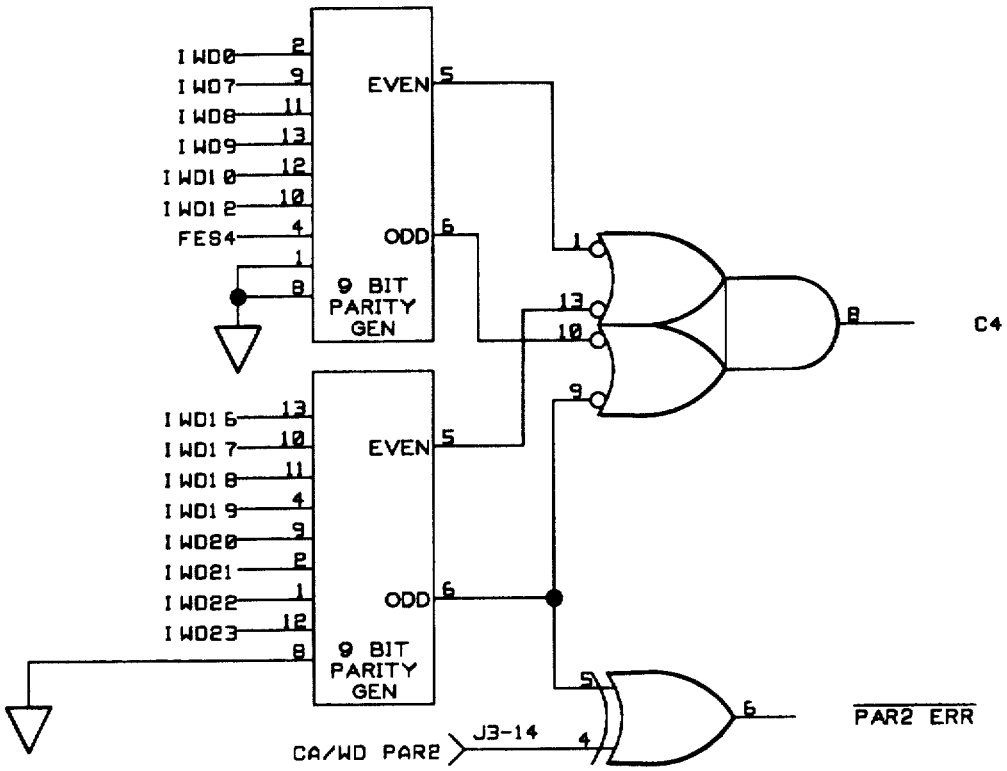
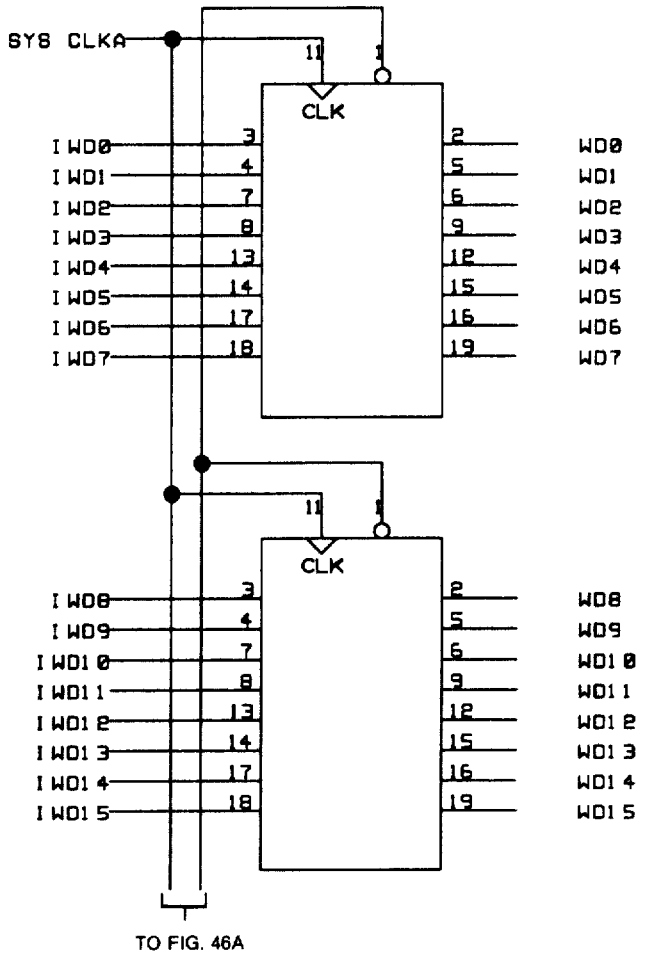

I WD0
I WD7
I WD8
I WD9
I WD10
I WD12
FES4
EVEN
ODD
9 BIT PARITY GEN
I WD16
I WD17
I WD18
I WD19
I WD20
I WD21
I WD22
I WD23
EVEN
ODD
9 BIT PARITY GEN
C4
PAR2 ERR
CA/WD PAR2
J3-14

FIG. 45C



(32 BITS) REGISTER AND, (8 BITS) REGISTER

FIG. 46

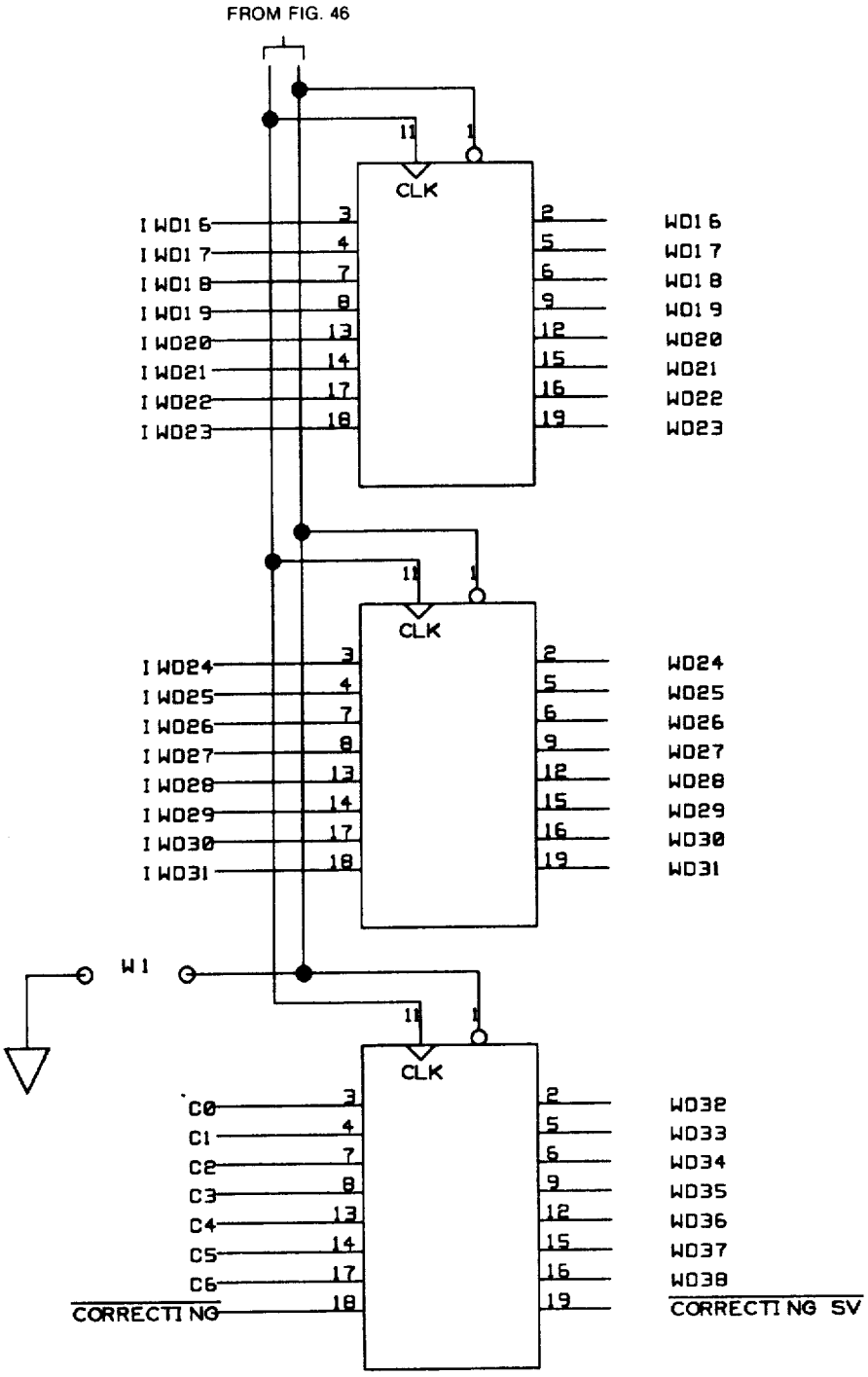
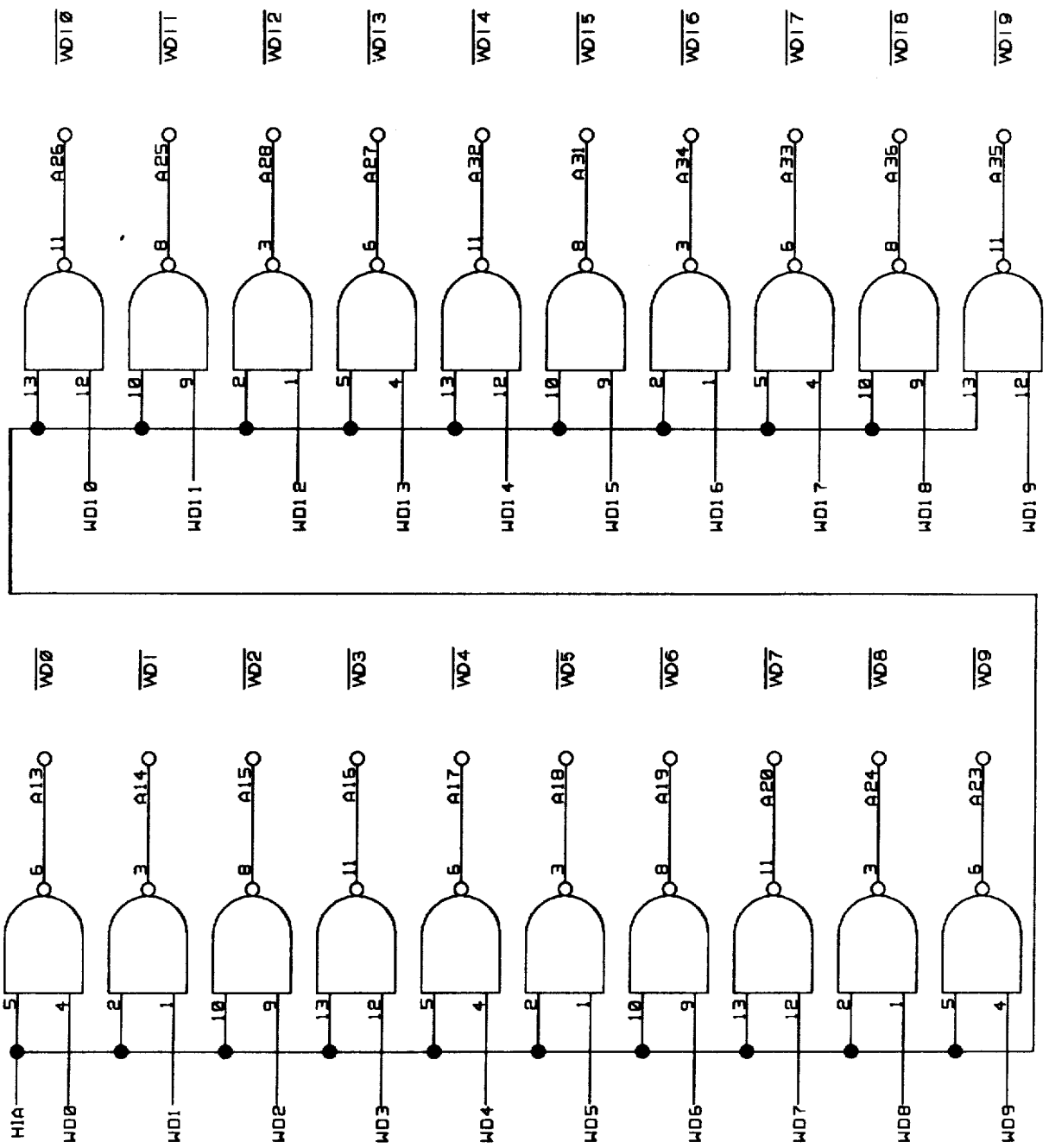

FROM FIG. 46
CLK
I WD16
I WD17
I WD18
I WD19
I WD20
I WD21
I WD22
I WD23
WD16
WD17
WD18
WD19
WD20
WD21
WD22
WD23
CLK
I WD24
I WD25
I WD26
I WD27
I WD28
I WD29
I WD30
I WD31
WD24
WD25
WD26
WD27
WD28
WD29
WD30
WD31
W1
CLK
C0
C1
C2
C3
C4
C5
C6
CORRECTING
WD32
WD33
WD34
WD35
WD36
WD37
WD38
CORRECTING SV

FIG. 46A



WRITE DATA BUS DRIVERS

FIG. 47

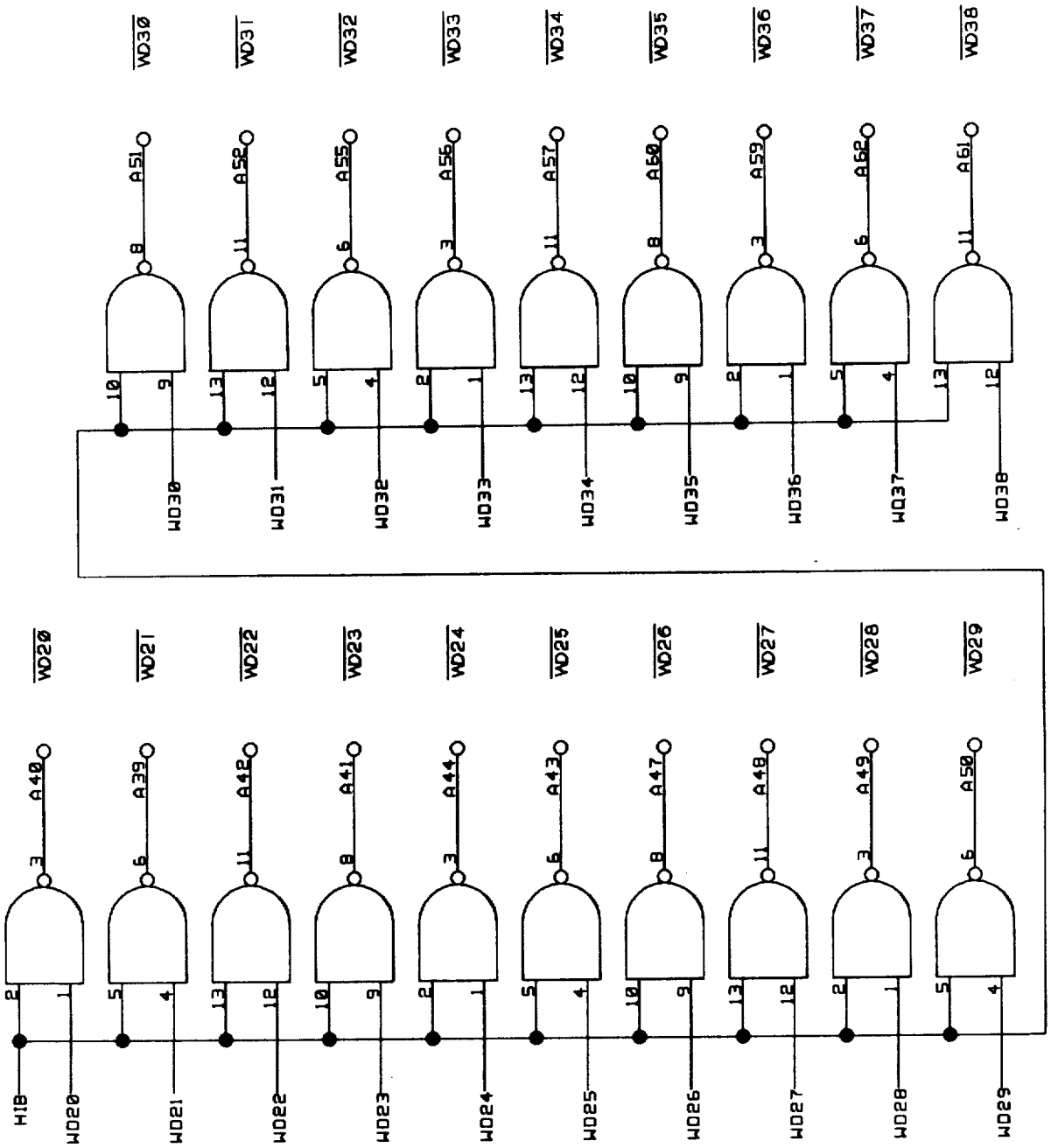
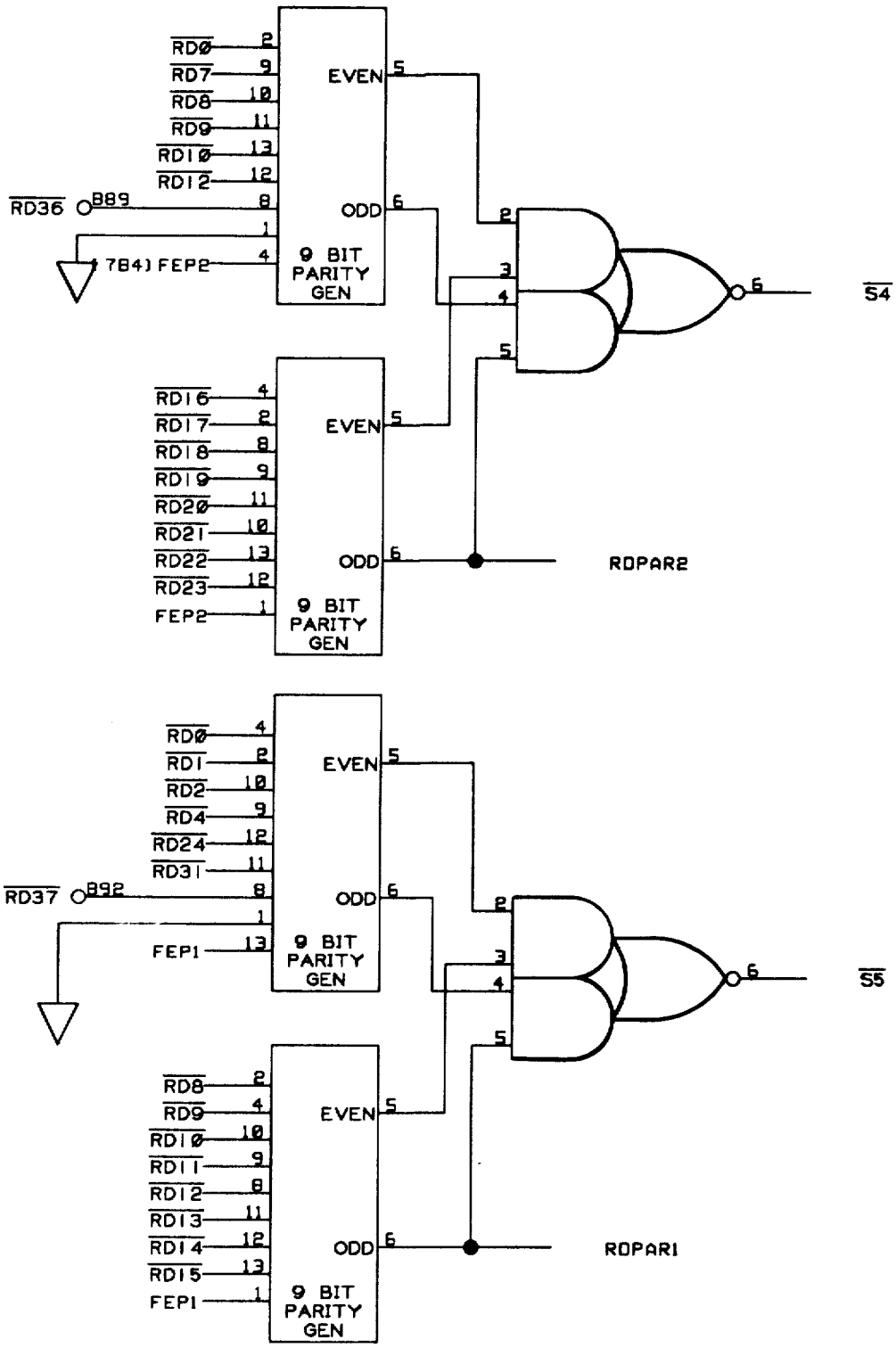

WD30
WD31
WD32
WD33
WD34
WD35
WD36
WD37
WD38
A51
A52
A55
A56
A57
A60
A59
A62
A61
WD30
WD31
WD32
WD33
WD34
WD35
WD36
WD37
WD38
WD20
WD21
WD22
WD23
WD24
WD25
WD26
WD27
WD28
WD29
A40
A39
A42
A41
A44
A43
A47
A48
A49
A50
H18
WD20
WD21
WD22
WD23
WD24
WD25
WD26
WD27
WD28
WD29

FIG. 47A



S-BIT GENERATOR

FIG. 48

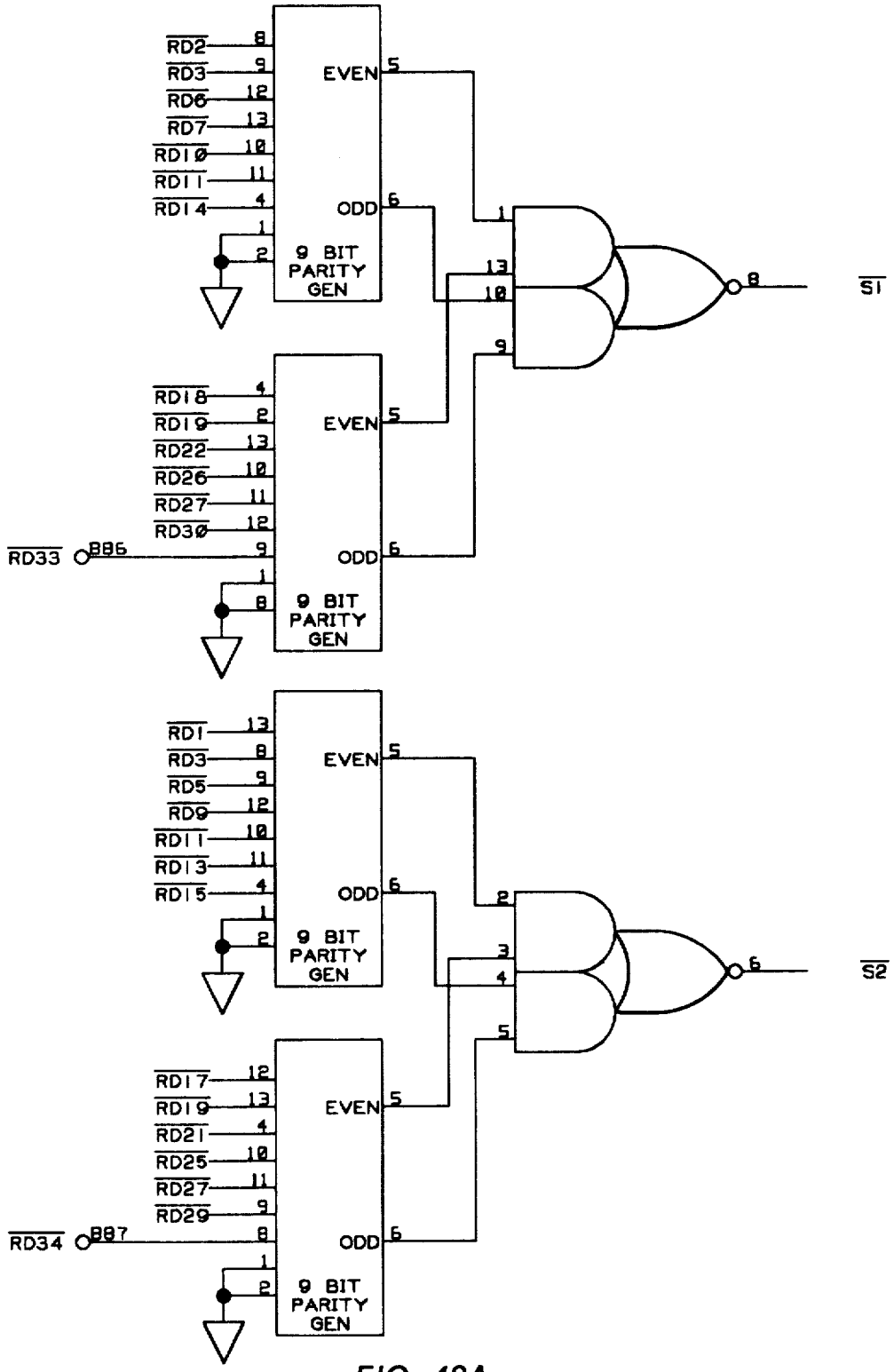

RD2
RD3
RD6
RD7
RD10
RD11
RD14
EVEN
ODD
9 BIT PARITY GEN
RD18
RD19
RD22
RD26
RD27
RD30
RD33
B86
RD1
RD3
RD5
RD9
RD11
RD13
RD15
RD17
RD19
RD21
RD25
RD27
RD29
RD34
B87
S1
S2

FIG. 48A

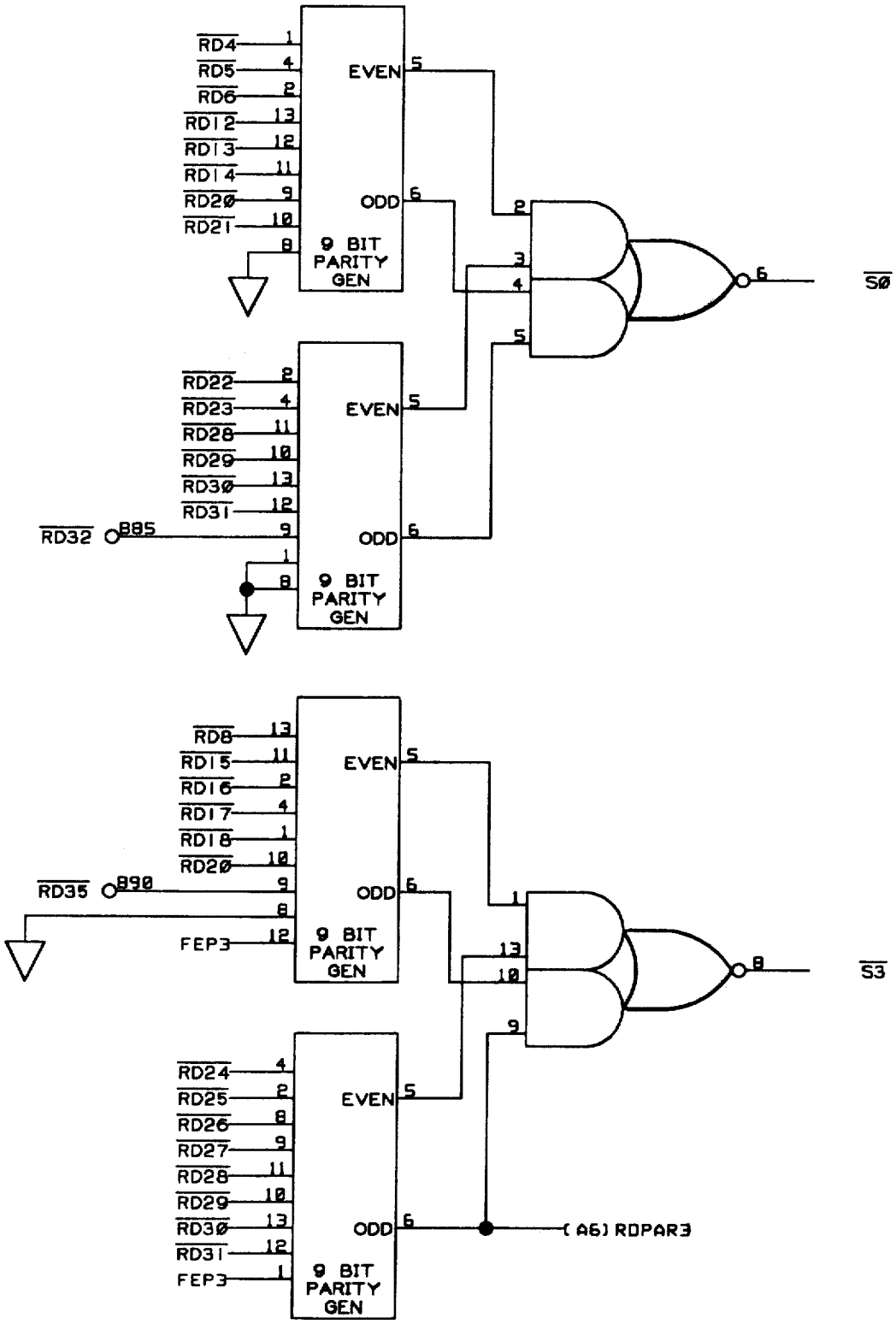

RD4
RD5
RD6
RD12
RD13
RD14
RD20
RD21
EVEN
ODD
9 BIT PARITY GEN
S0
RD22
RD23
RD28
RD29
RD30
RD31
RD32
B85
RD8
RD15
RD16
RD17
RD18
RD20
RD35
B90
FEP3
S3
RD24
RD25
RD26
RD27
RD28
RD29
RD30
RD31
FEP3
(A6) RDPAR3

FIG. 48B

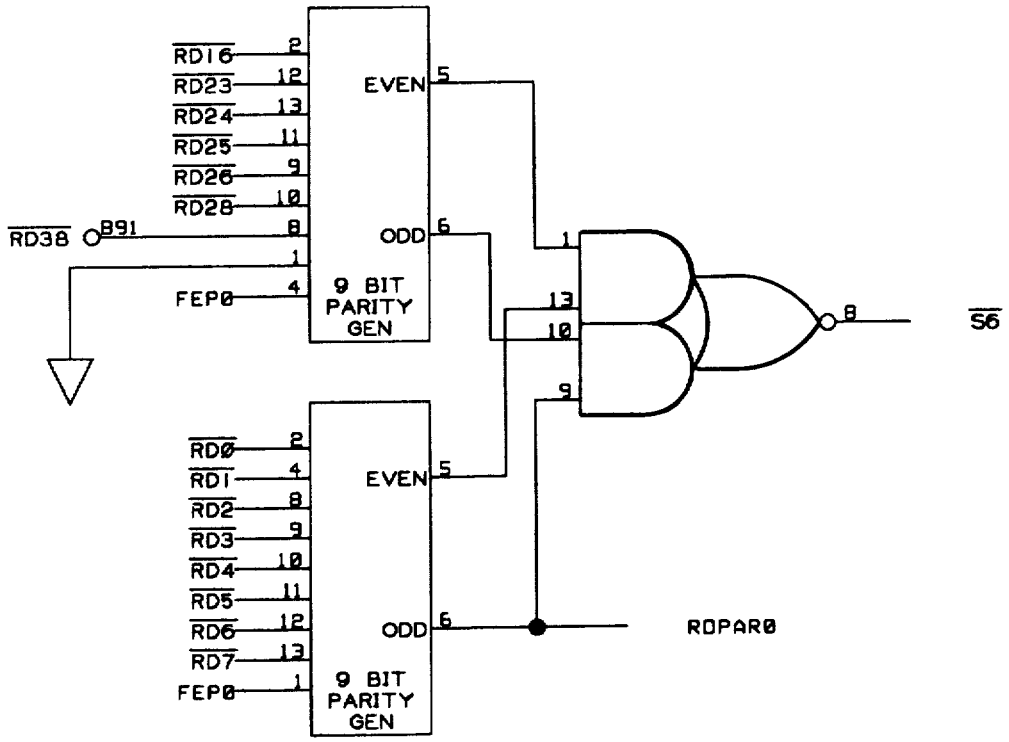
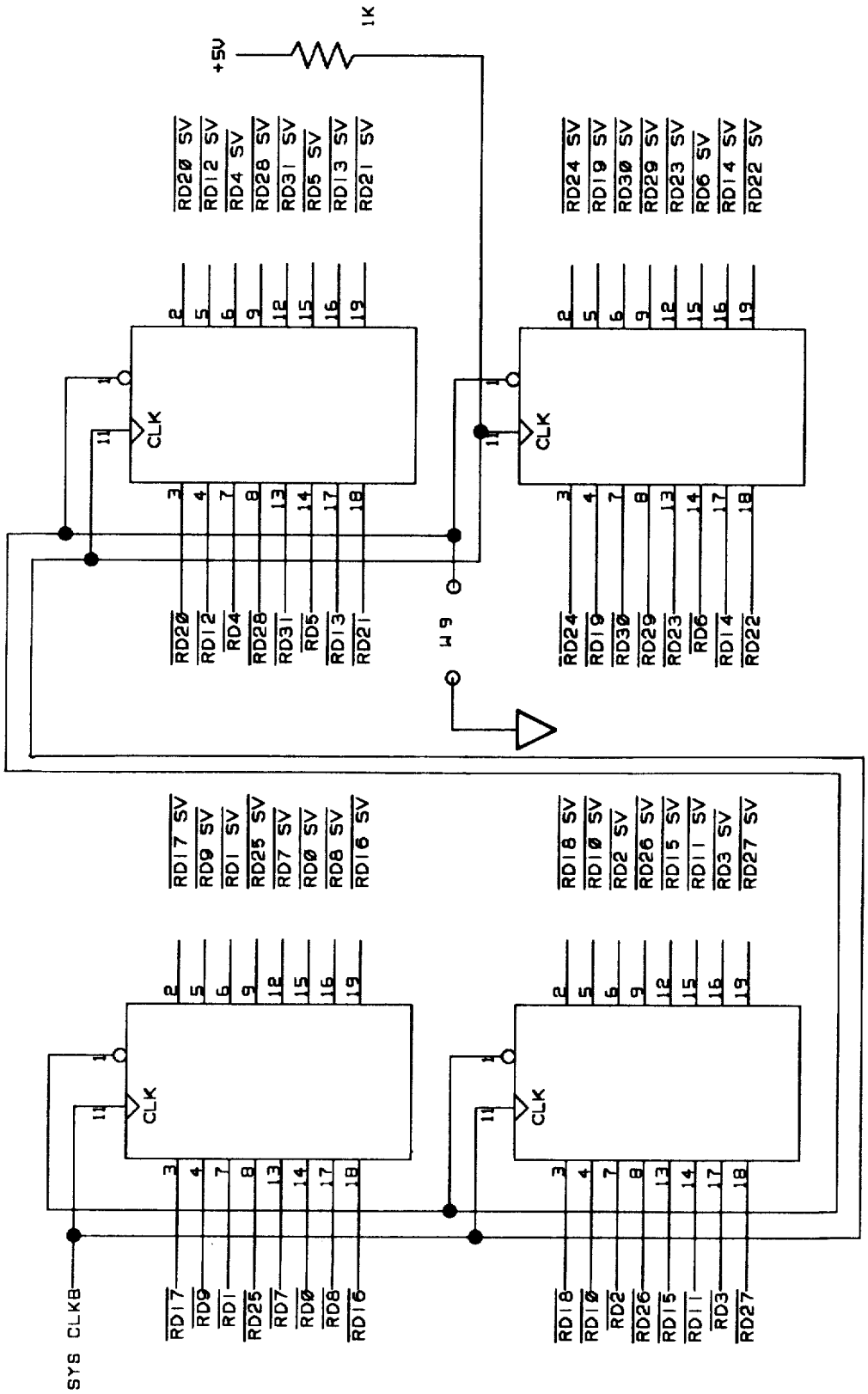
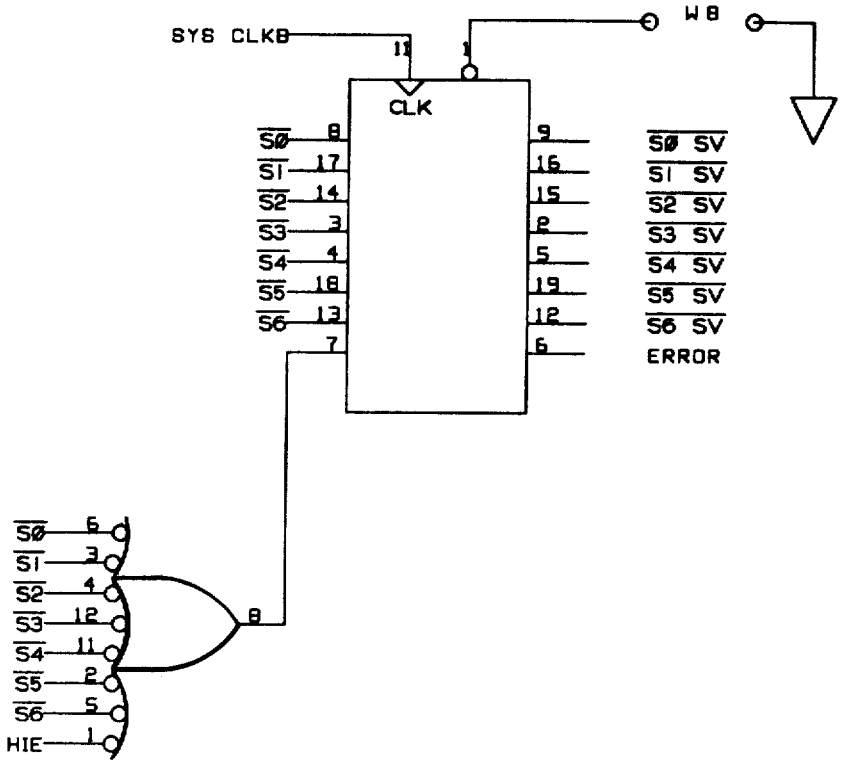

RD16
RD23
RD24
RD25
RD26
RD28
RD38
B91
FEP0
EVEN
ODD
9 BIT PARITY GEN
RD0
RD1
RD2
RD3
RD4
RD5
RD6
RD7
FEP0
EVEN
ODD
9 BIT PARITY GEN
RDPAR0
S6

FIG. 48C



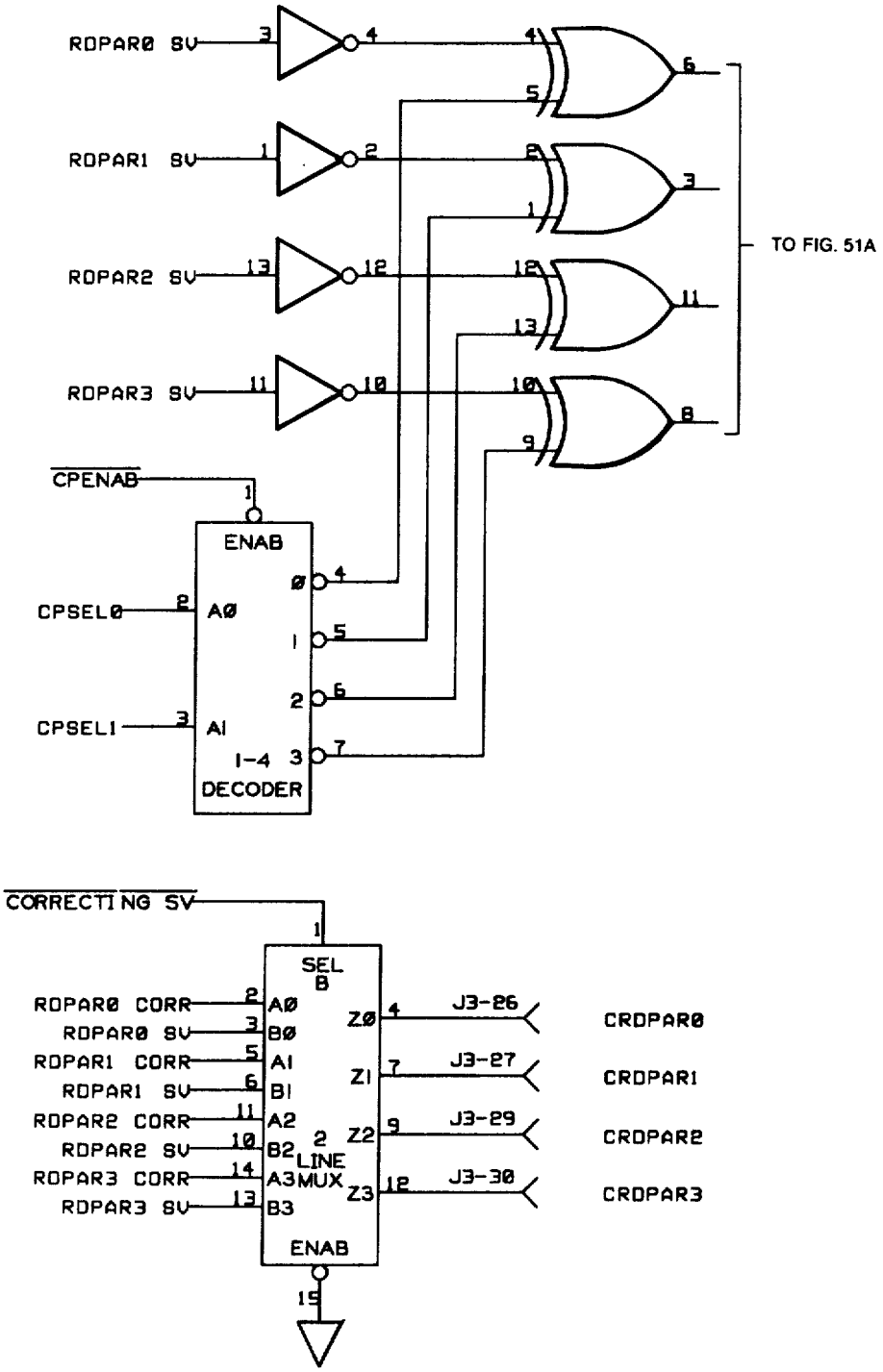
RD SV REGISTER

FIG. 49



S-SV REGISTER

FIG. 50



PARITY LOGIC

FIG. 51

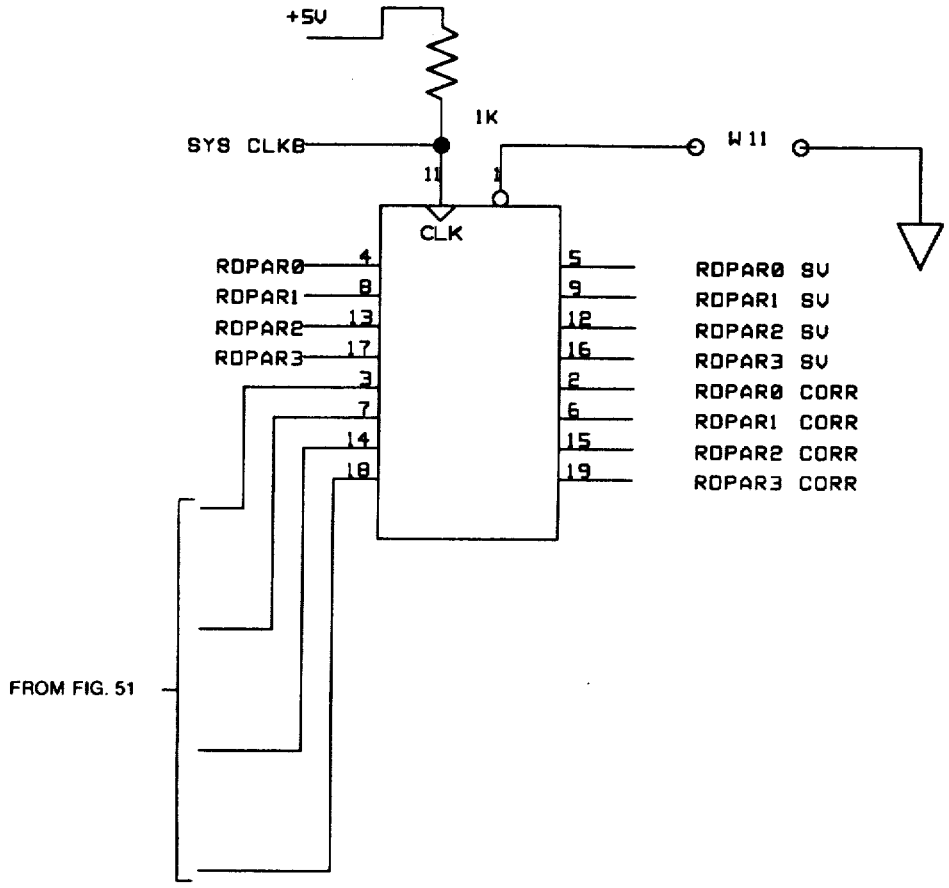
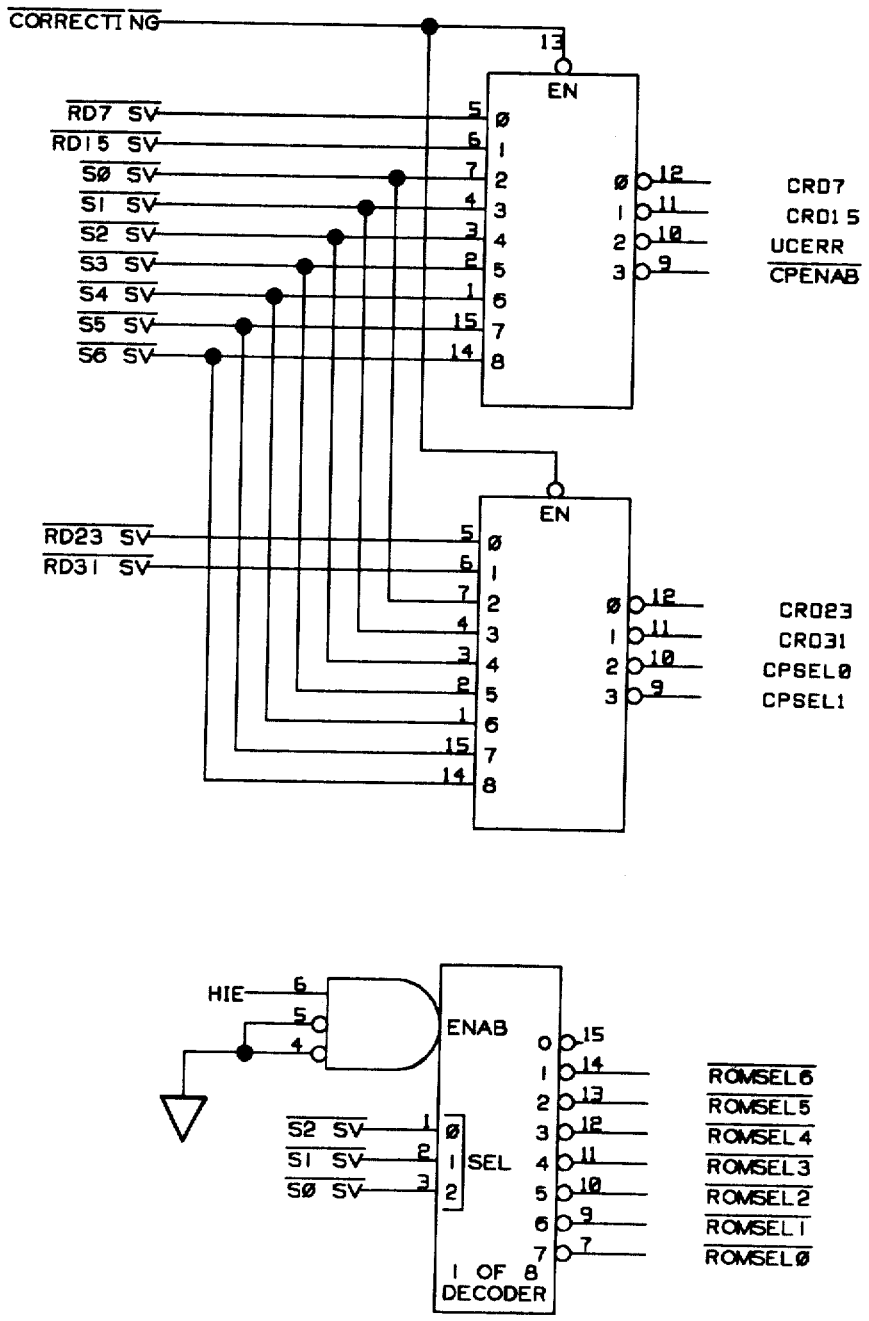

+5V
1K
SYS CLKB
11
1
W11
CLK
RDPAR0
RDPAR1
RDPAR2
RDPAR3
4
8
13
17
3
7
14
18
5
9
12
16
2
6
15
19
RDPAR0 SV
RDPAR1 SV
RDPAR2 SV
RDPAR3 SV
RDPAR0 CORR
RDPAR1 CORR
RDPAR2 CORR
RDPAR3 CORR
FROM FIG. 51

FIG. 51A



CORRECTION LOGIC

FIG. 52

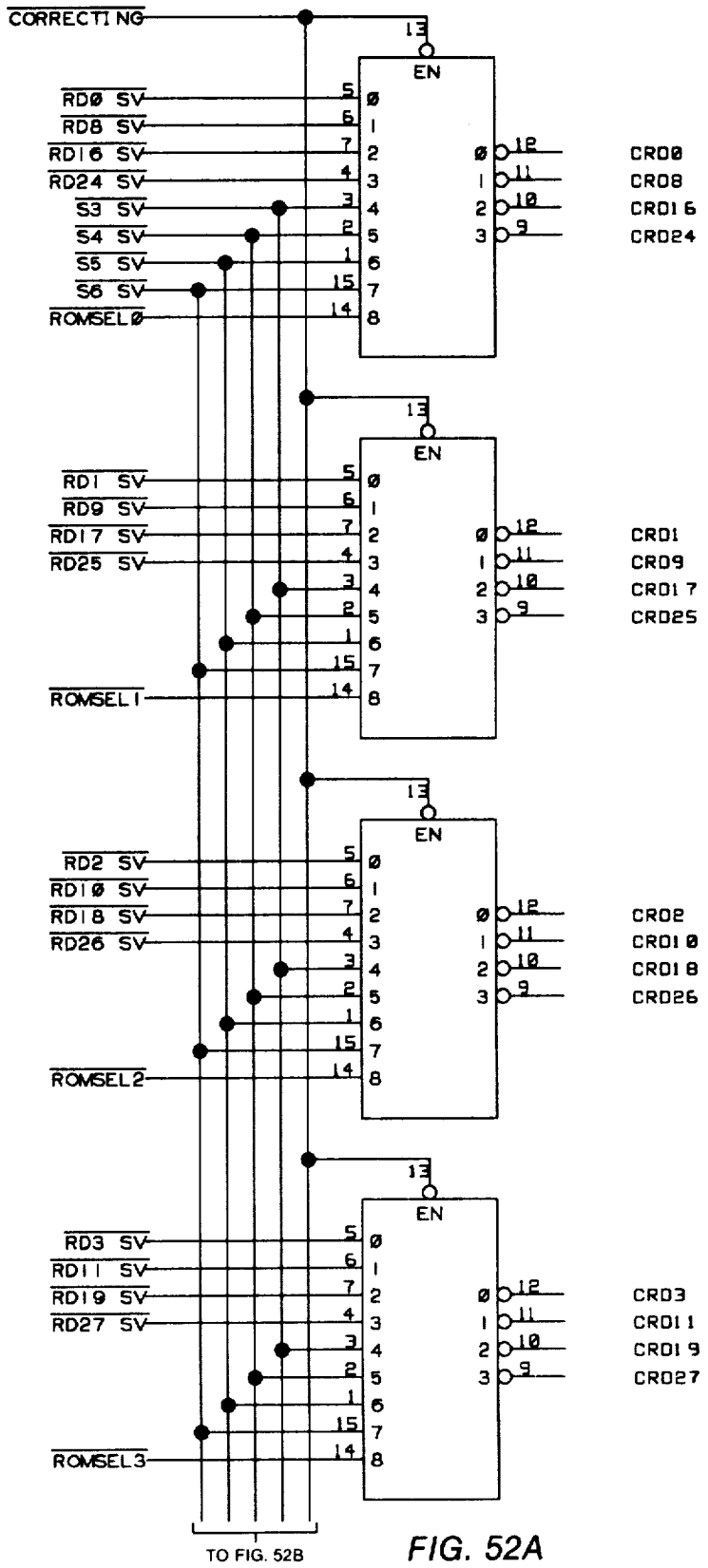

CORRECTING
RD0 SV
RD8 SV
RD16 SV
RD24 SV
S3 SV
S4 SV
S5 SV
S6 SV
ROMSEL0
EN
CR00
CR08
CR016
CR024
RD1 SV
RD9 SV
RD17 SV
RD25 SV
ROMSEL1
CR01
CR09
CR017
CR025
RD2 SV
RD10 SV
RD18 SV
RD26 SV
ROMSEL2
CR02
CR010
CR018
CR026
RD3 SV
RD11 SV
RD19 SV
RD27 SV
ROMSEL3
CR03
CR011
CR019
CR027
TO FIG. 52B

FIG. 52A

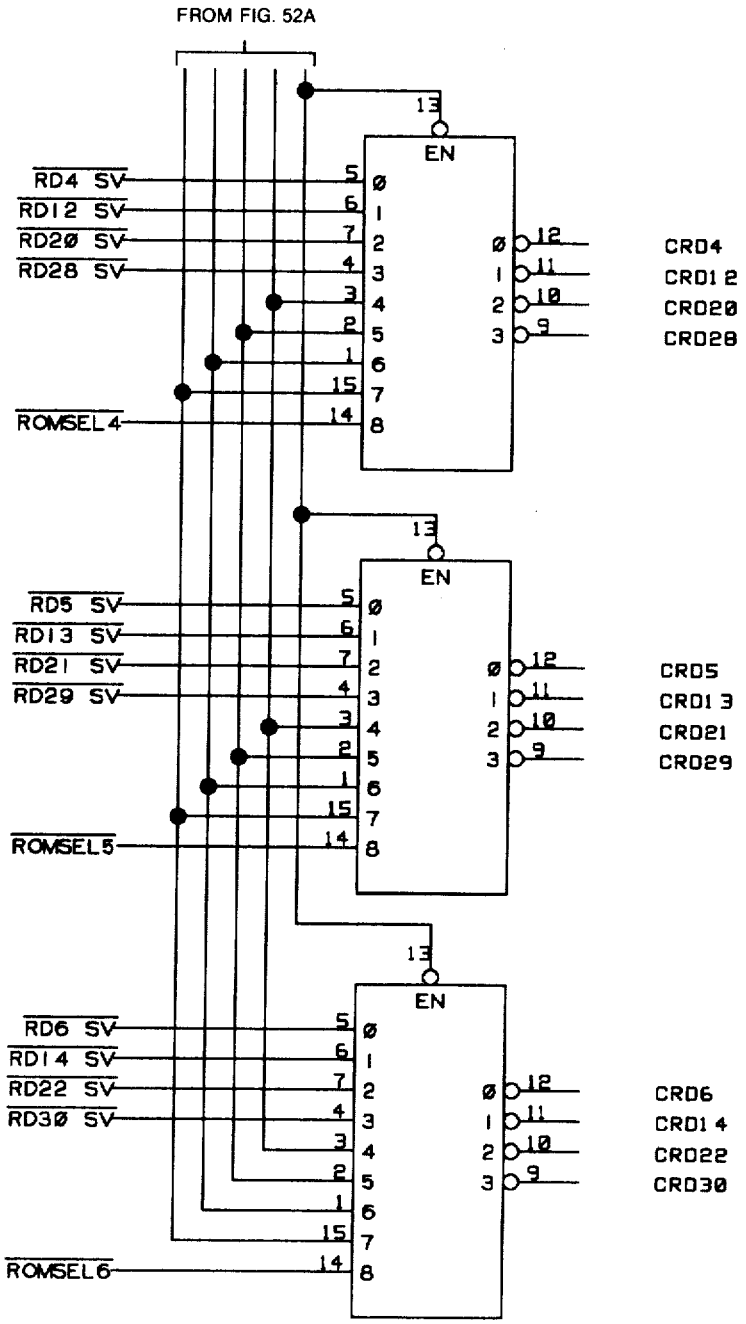
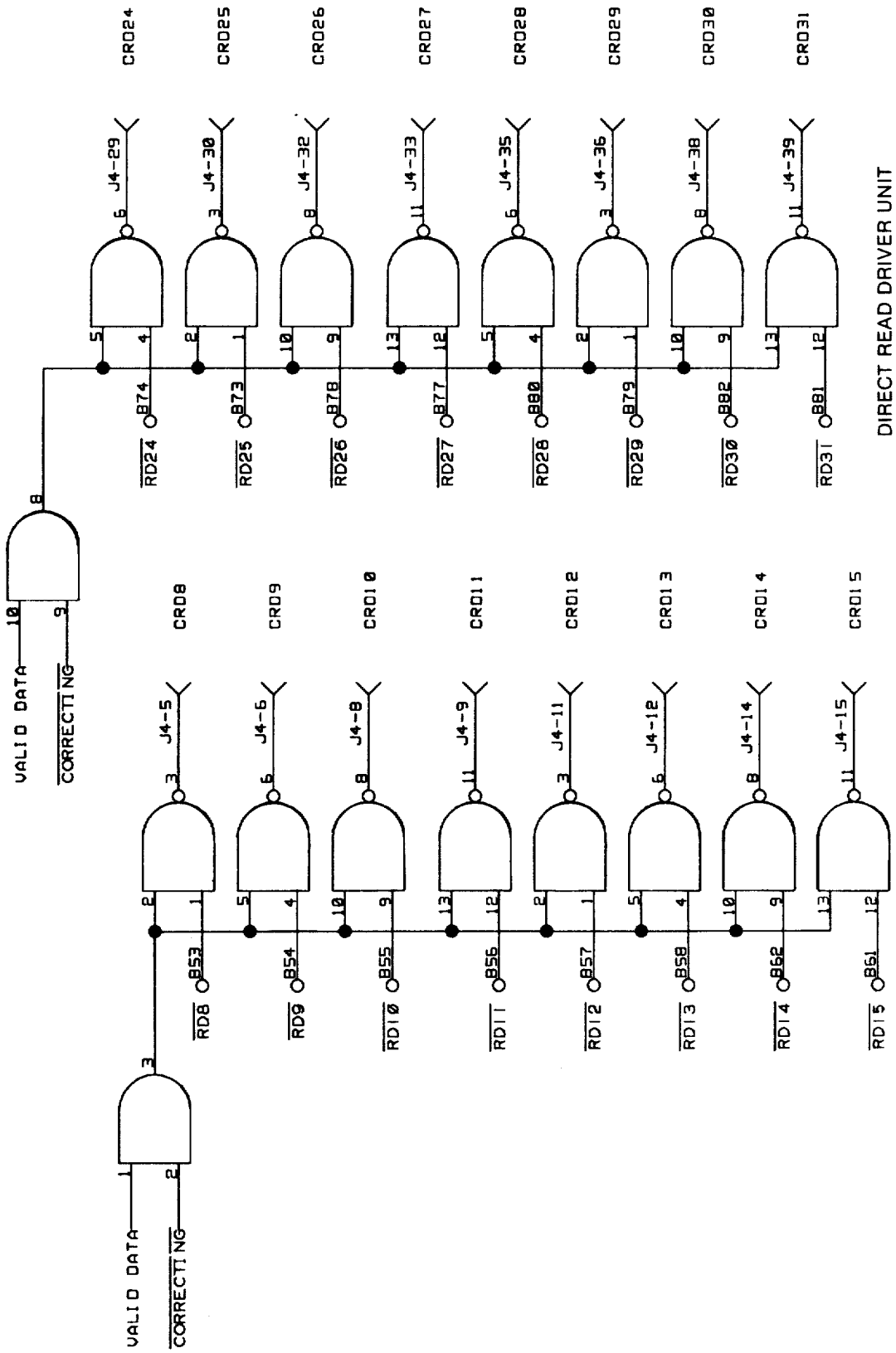

FROM FIG. 52A
EN
RD4 SV
RD12 SV
RD20 SV
RD28 SV
ROMSEL4
CRD4
CRD12
CRD20
CRD28
RD5 SV
RD13 SV
RD21 SV
RD29 SV
ROMSEL5
CRD5
CRD13
CRD21
CRD29
RD6 SV
RD14 SV
RD22 SV
RD30 SV
ROMSEL6
CRD6
CRD14
CRD22
CRD30

FIG. 52B



DIRECT READ DRIVER UNIT

FIG. 53

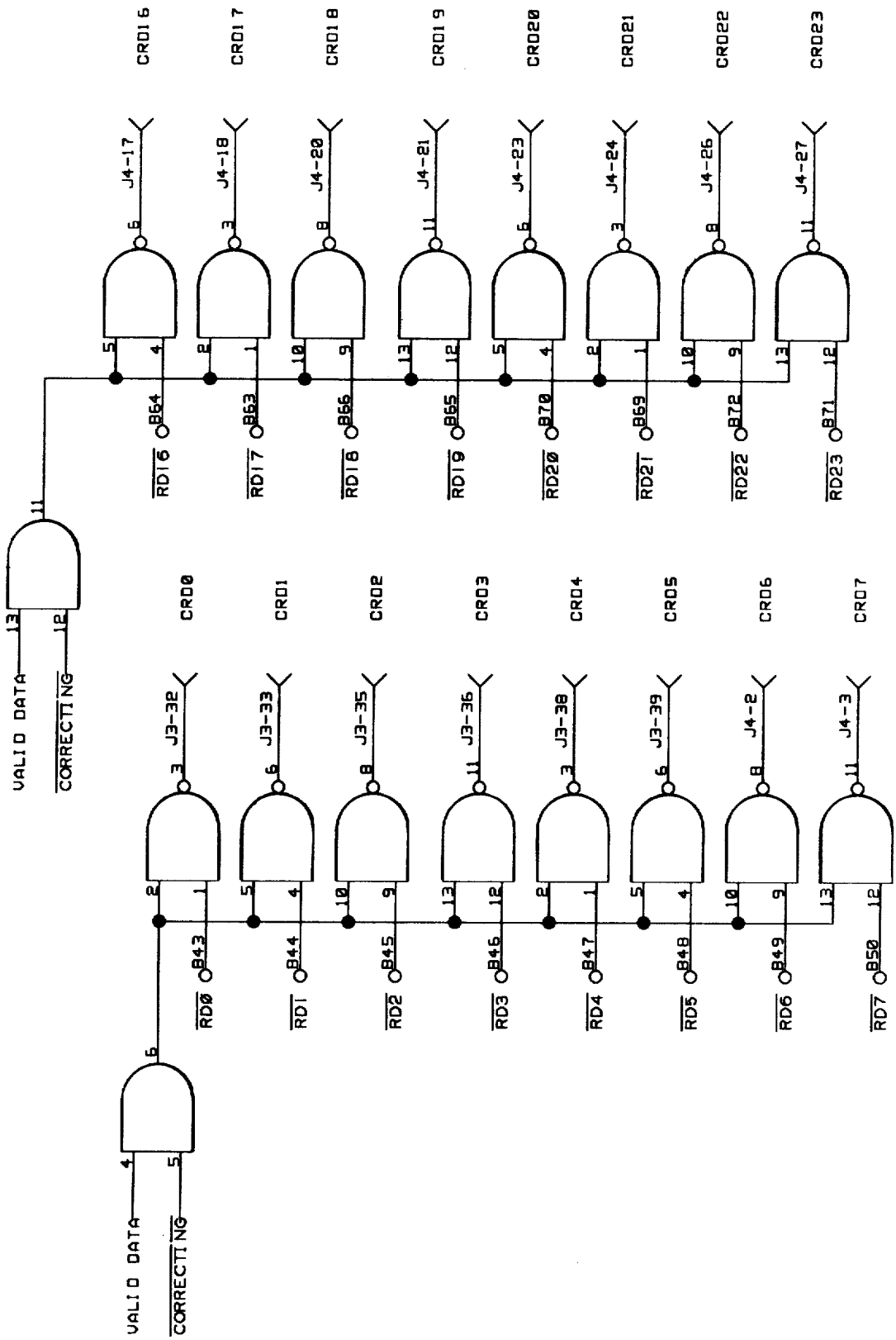
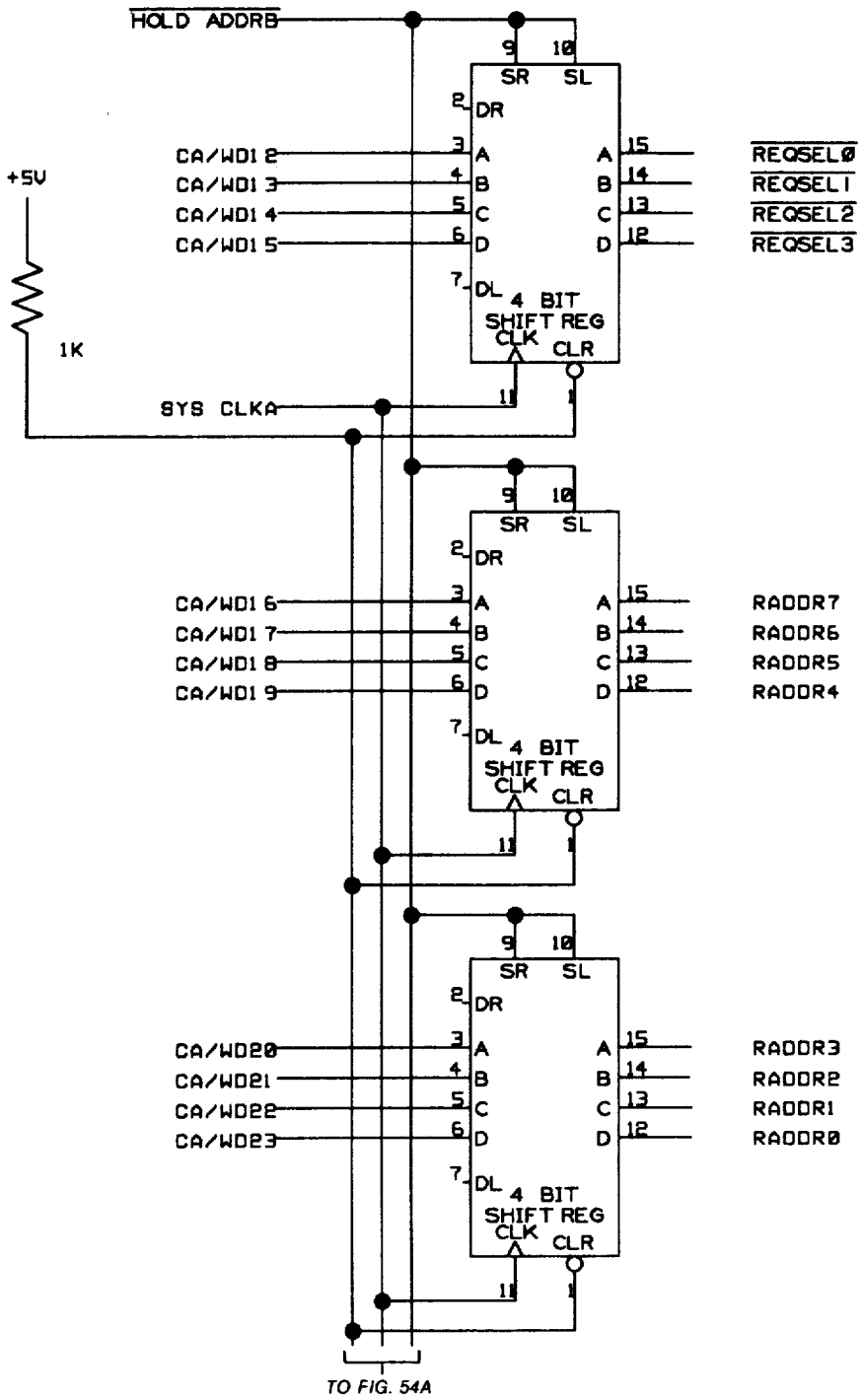

VALID DATA
CORRECTING
CRD0
CRD1
CRD2
CRD3
CRD4
CRD5
CRD6
CRD7
J3-32
J3-33
J3-35
J3-36
J3-38
J3-39
J4-2
J4-3
RD0
RD1
RD2
RD3
RD4
RD5
RD6
RD7
B43
B44
B45
B46
B47
B48
B49
B50
VALID DATA
CORRECTING
CRD16
CRD17
CRD18
CRD19
CRD20
CRD21
CRD22
CRD23
J4-17
J4-18
J4-20
J4-21
J4-23
J4-24
J4-26
J4-27
RD16
RD17
RD18
RD19
RD20
RD21
RD22
RD23
B64
B63
B66
B65
B70
B69
B72
B71

FIG. 53A



R/W MOD SELS UNIT,
RADDR UNIT AND CADDR UNIT

FIG. 54

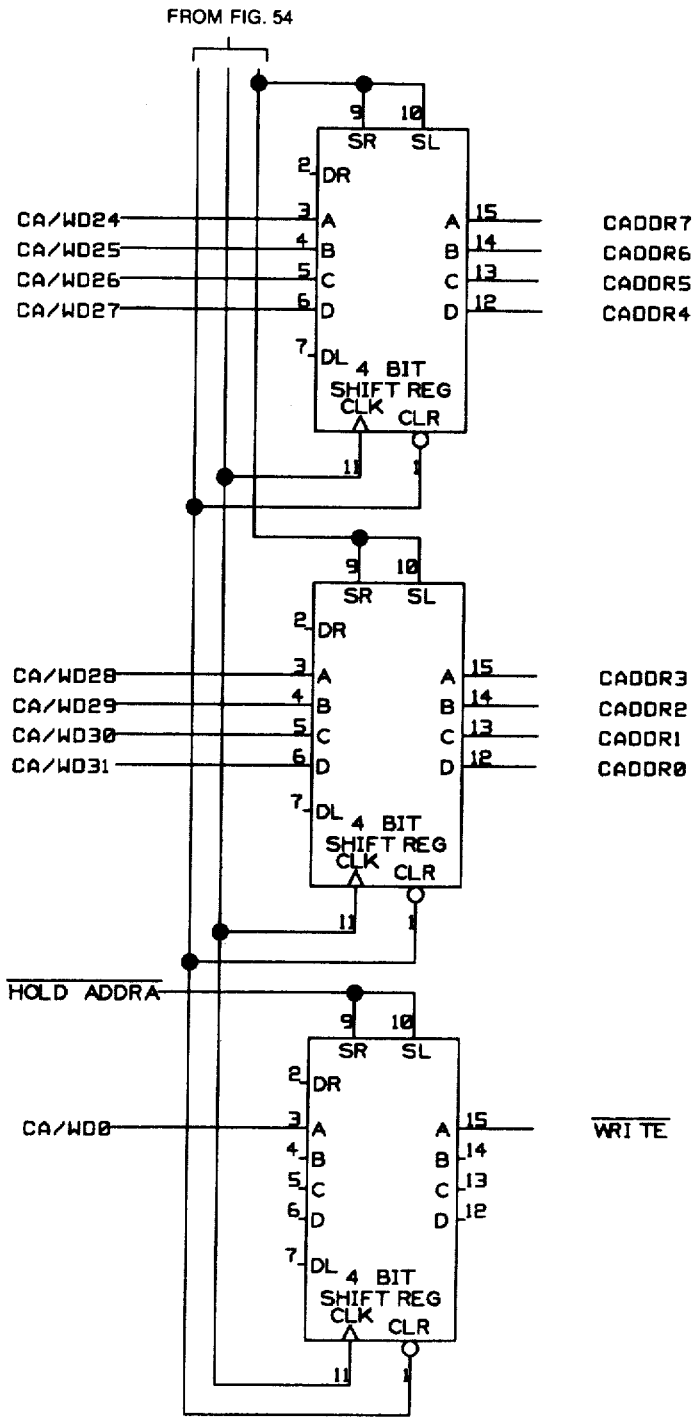
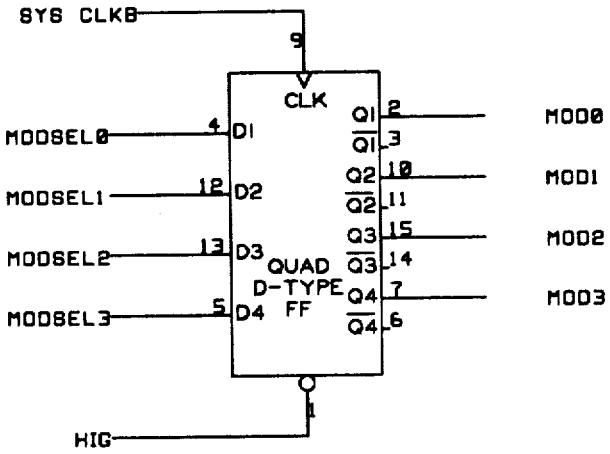
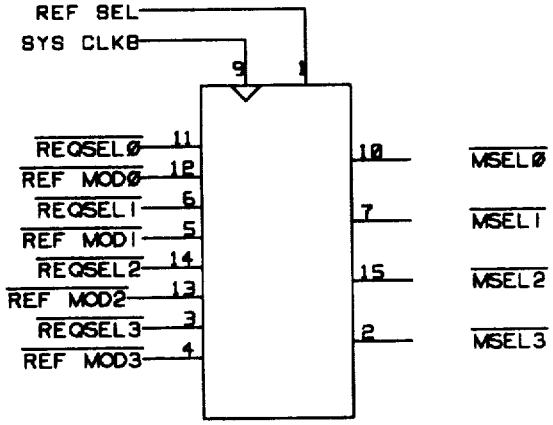
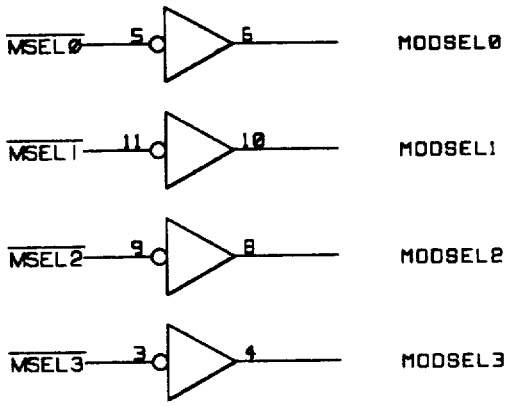

FROM FIG. 54
SR
SL
DR
CA/WD24
CA/WD25
CA/WD26
CA/WD27
A
B
C
D
DL
4 BIT
SHIFT REG
CLK
CLR
CADDR7
CADDR6
CADDR5
CADDR4
CA/WD28
CA/WD29
CA/WD30
CA/WD31
CADDR3
CADDR2
CADDR1
CADDR0
HOLD ADDRA
CA/WD0
WRITE

FIG. 54A

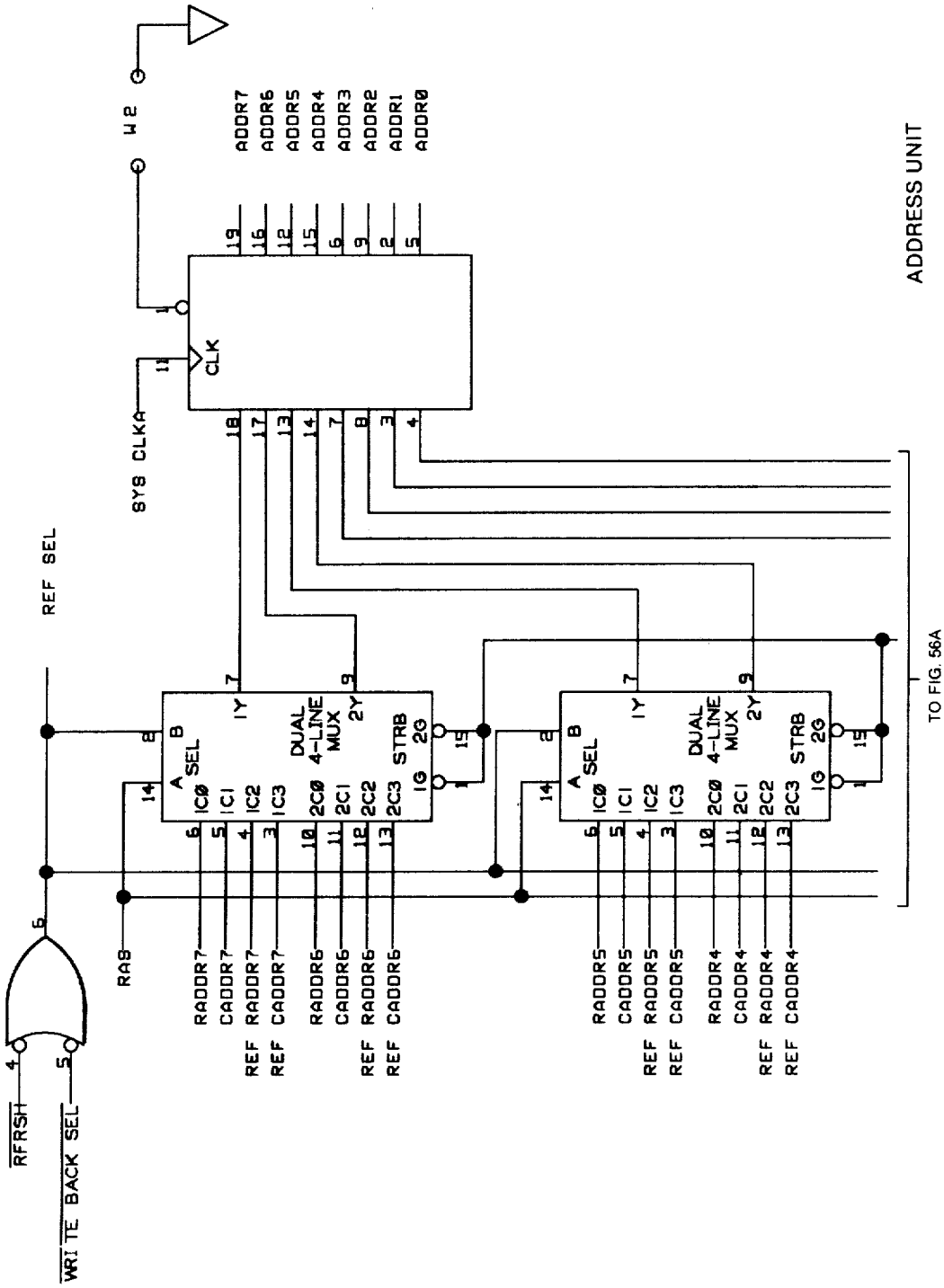






MODSEL LOGIC

FIG. 55



ADDRESS UNIT

FIG. 56

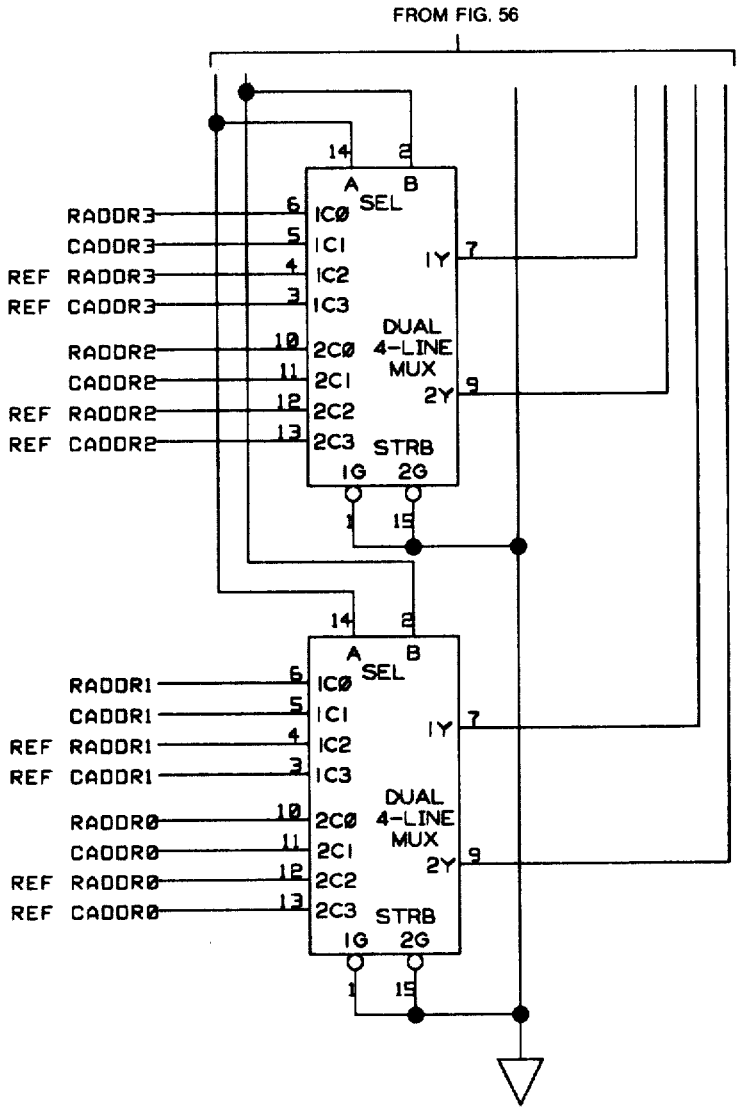
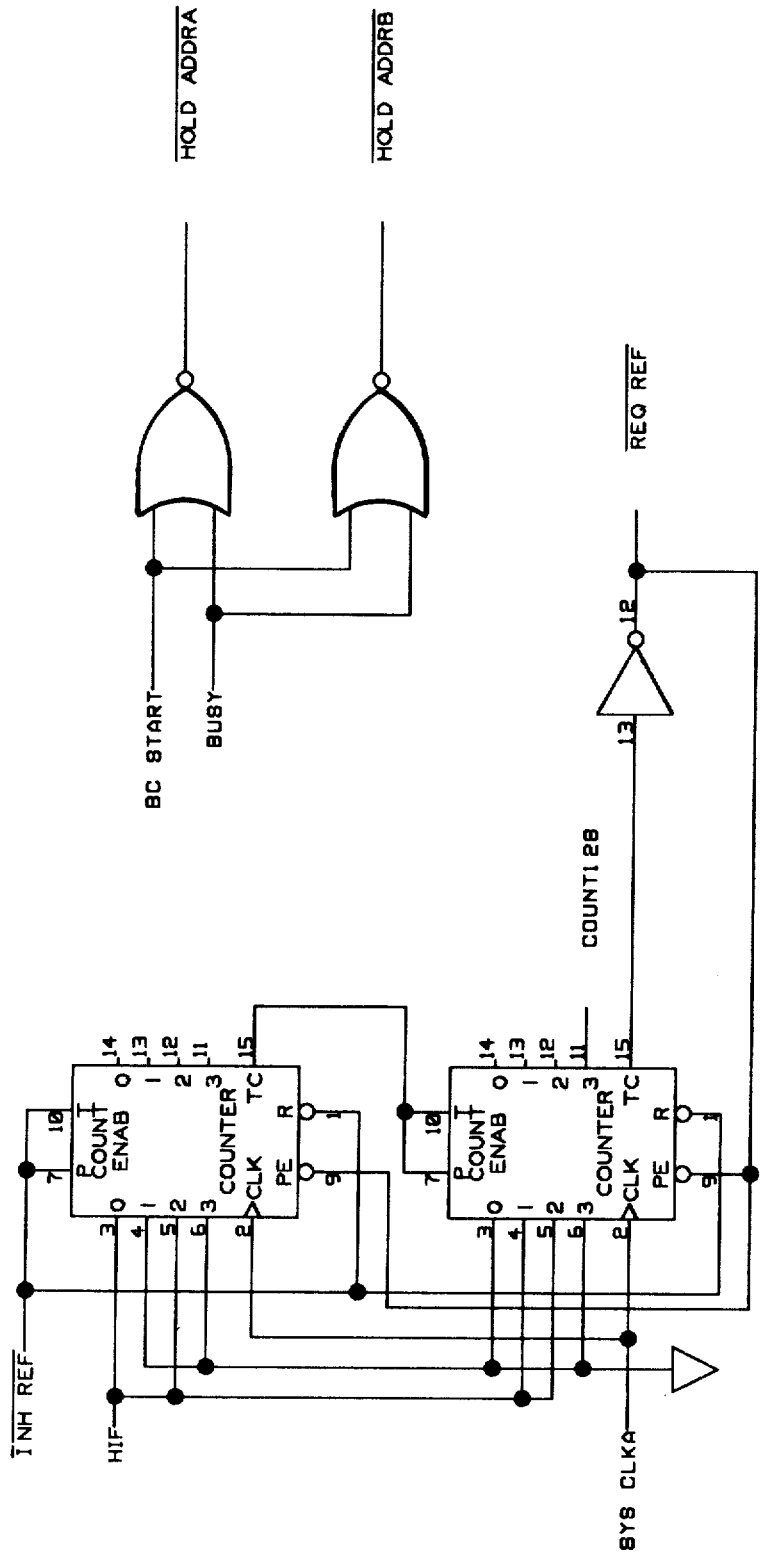

FROM FIG. 56
RADDR3
CADDR3
REF RADDR3
REF CADDR3
RADDR2
CADDR2
REF RADDR2
REF CADDR2
RADDR1
CADDR1
REF RADDR1
REF CADDR1
RADDR0
CADDR0
REF RADDR0
REF CADDR0
A
B
SEL
1C0
1C1
1C2
1C3
1Y
DUAL
4-LINE
MUX
2C0
2C1
2C2
2C3
2Y
STRB
1G
2G

FIG. 56A



BANK CONTROLLER TIMING AND CONTROL LOGIC

FIG. 57

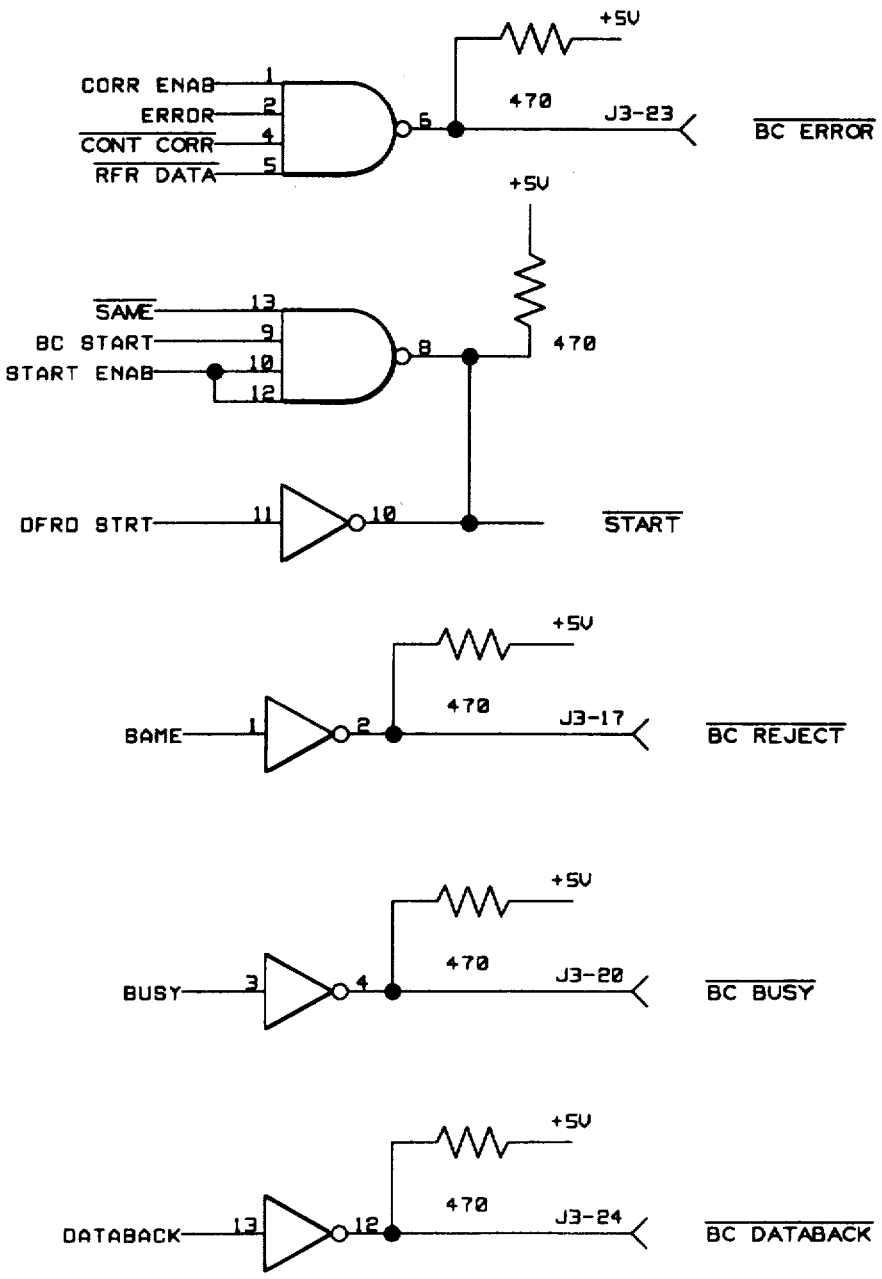

+5V
CORR ENAB
ERROR
CONT CORR
RFR DATA
1
2
4
5
6
470
J3-23
BC ERROR
+5V
SAME
BC START
START ENAB
13
9
10
12
8
470
DFRD STRT
11
10
START
+5V
BAME
1
2
470
J3-17
BC REJECT
+5V
BUSY
3
4
470
J3-20
BC BUSY
+5V
DATABACK
13
12
470
J3-24
BC DATABACK

FIG. 57A

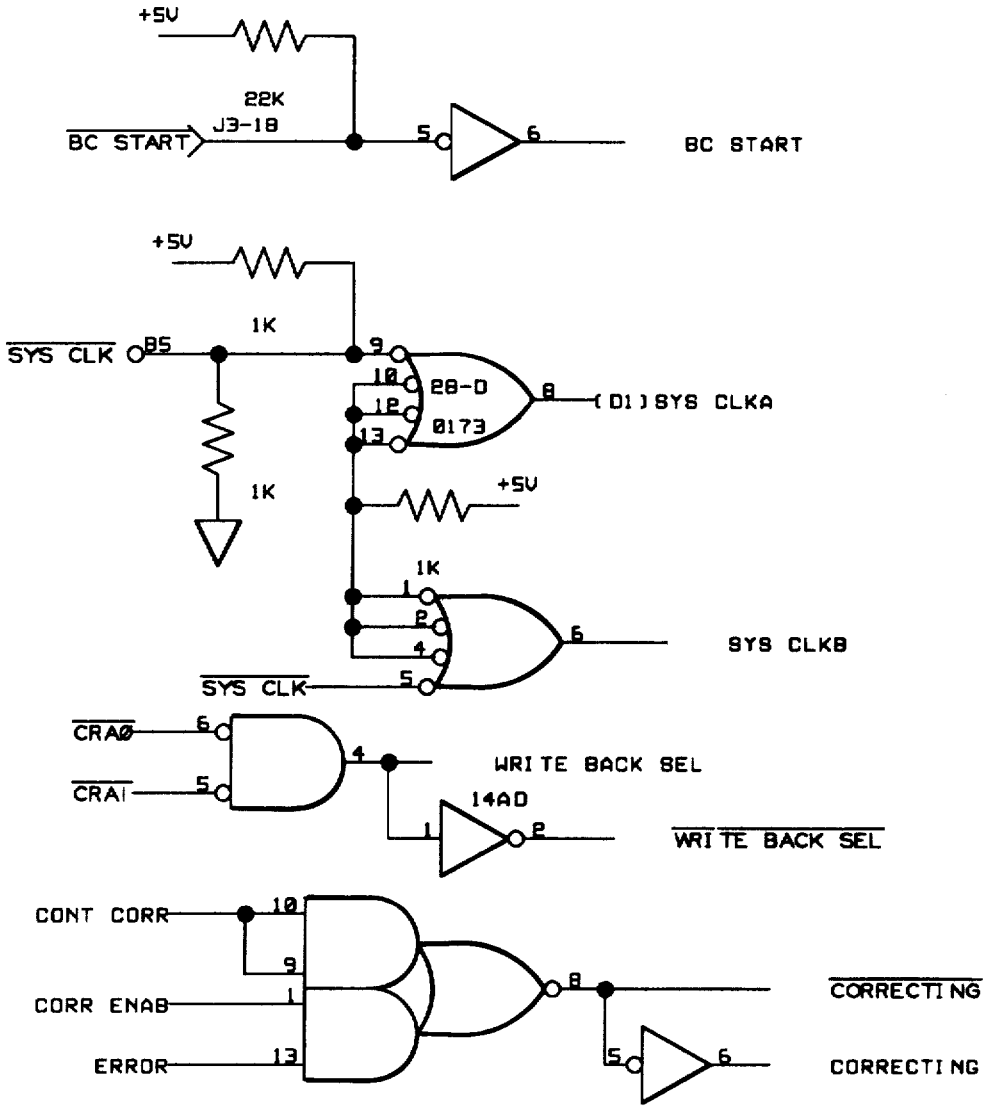
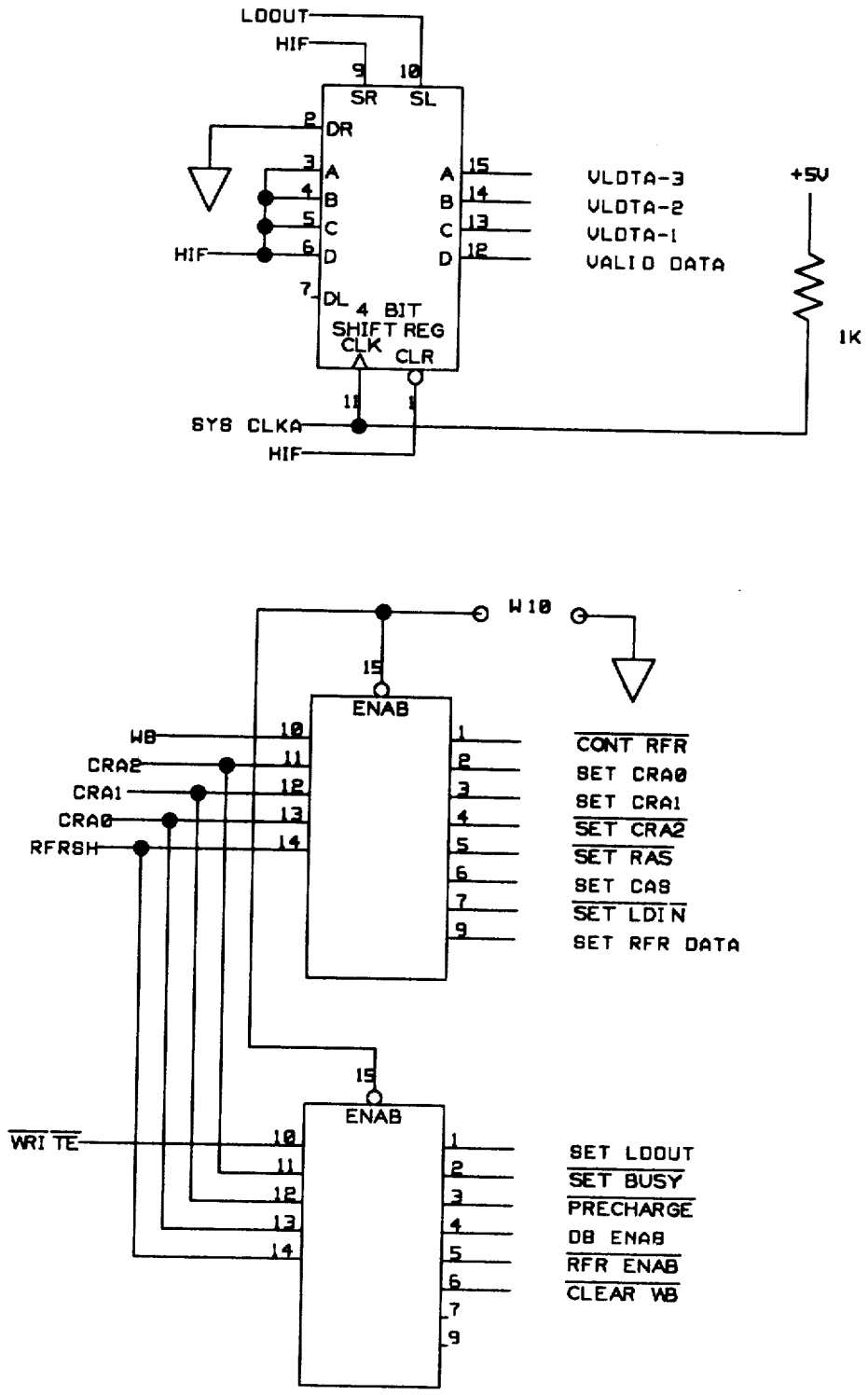

+5V
22K
J3-18
BC START
5
6
BC START
+5V
1K
SYS CLK
B5
9
10
12
13
28-D
0173
8
(D1) SYS CLKA
1K
+5V
1K
1
2
4
5
6
SYS CLKB
SYS CLK
CRA0
6
CRA1
5
4
WRITE BACK SEL
14AD
1
2
WRITE BACK SEL
CONT CORR
10
9
CORR ENAB
1
ERROR
13
8
CORRECTING
5
6
CORRECTING

FIG. 57B



BANK CONTROLLER TIMING,
REFRESH AND CONTROL LOGIC

FIG. 58

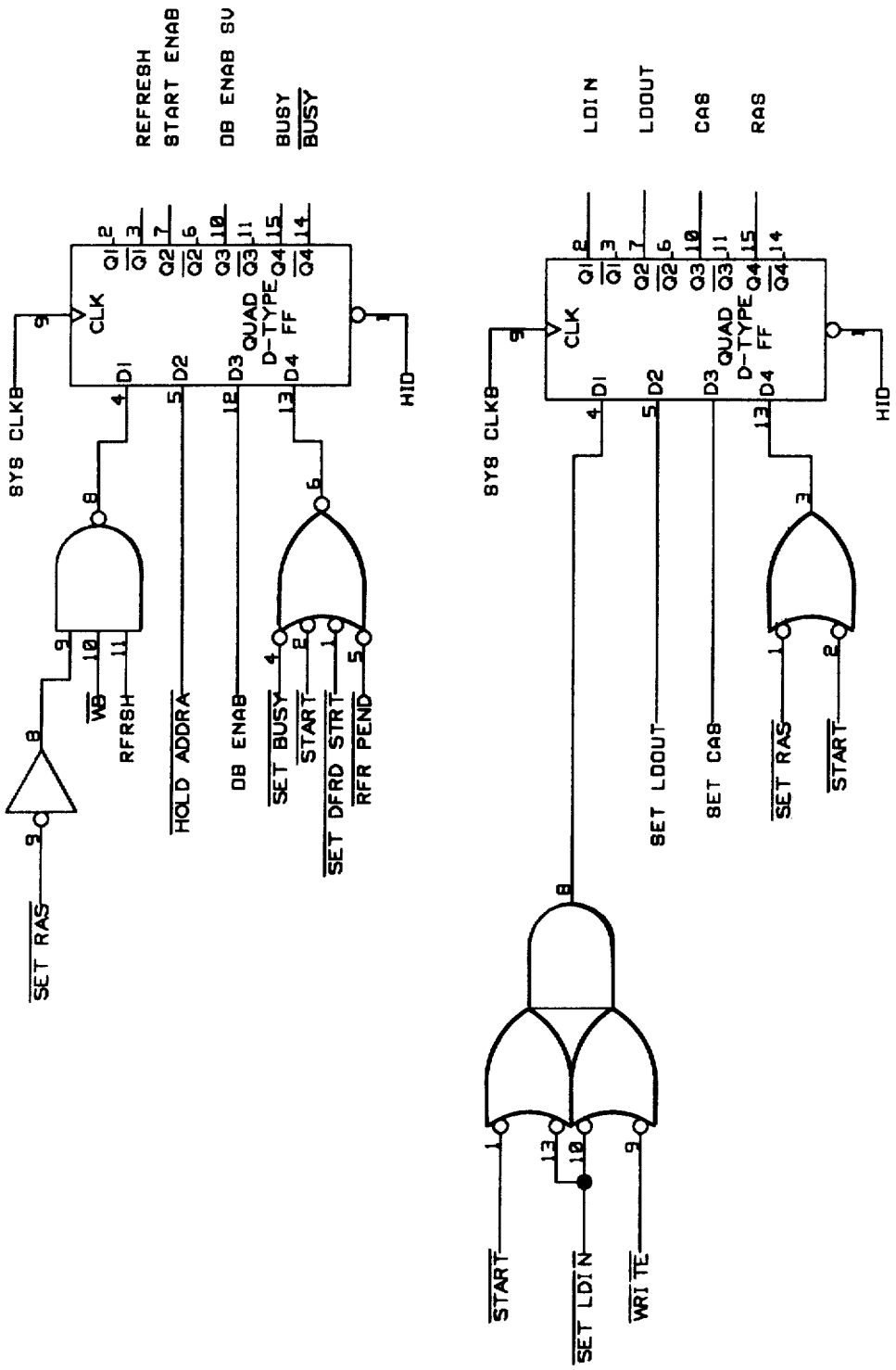

REFRESH
START ENAB
DB ENAB SV
BUSY
BUSY
LDIN
LDOUT
CAS
RAS
SYS CLKB
CLK
QUAD D-TYPE FF
HID
SET RAS
WB
RFRSH
HOLD ADDRA
DB ENAB
SET BUSY
START
SET DFRD STRT
RFR PEND
SET LDOUT
SET CAS
SET RAS
START
START
SET LDIN
WRITE

FIG. 58A

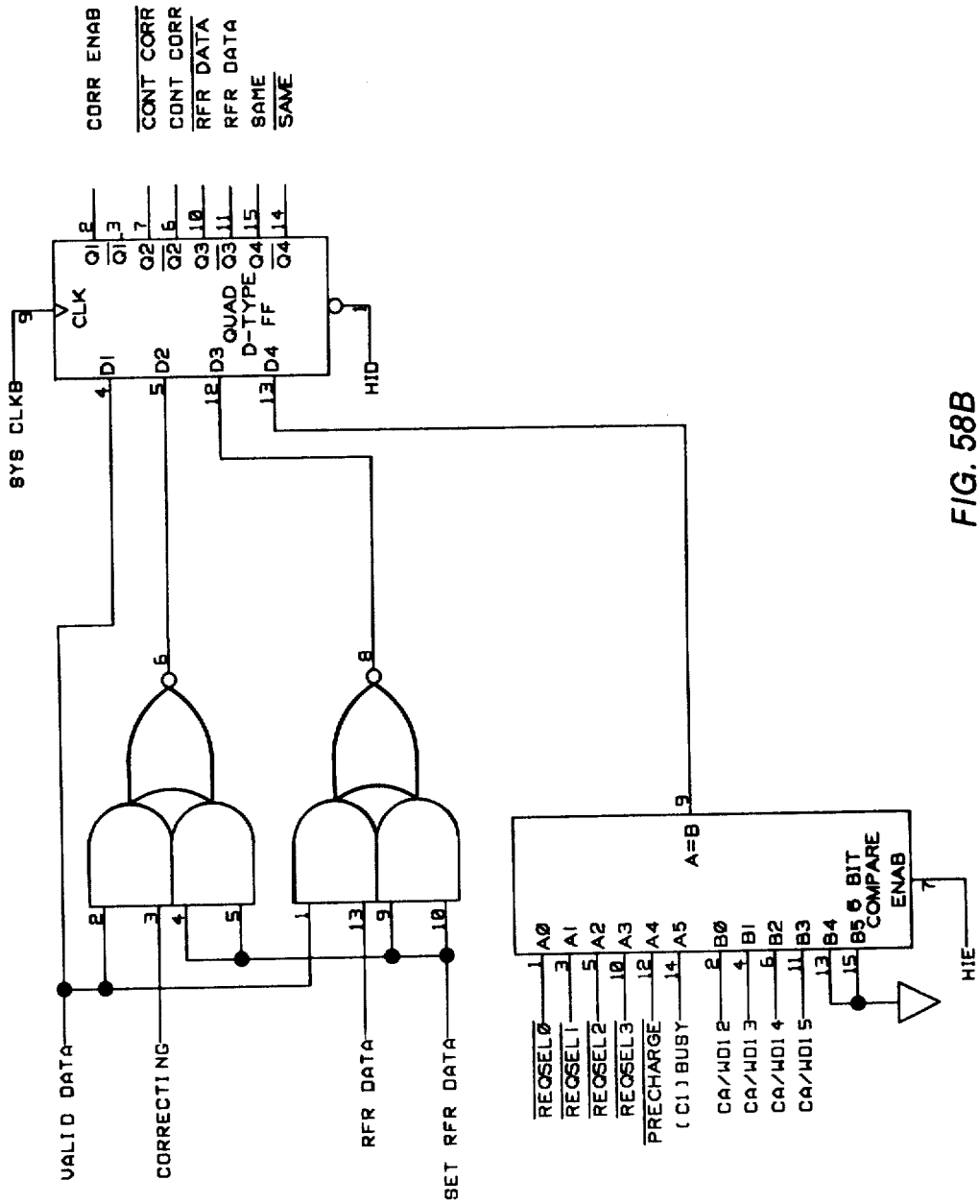

CORR ENAB
CONT CORR
CONT CORR
RFR DATA
RFR DATA
SAME
SAME
SYS CLKB
CLK
QUAD D-TYPE FF
HID
VALID DATA
CORRECTING
RFR DATA
SET RFR DATA
A=B
6 BIT COMPARE
ENAB
HIE
REQSEL0
REQSEL1
REQSEL2
REQSEL3
PRECHARGE
(C1) BUSY
CA/WD12
CA/WD13
CA/WD14
CA/WD15

FIG. 58B

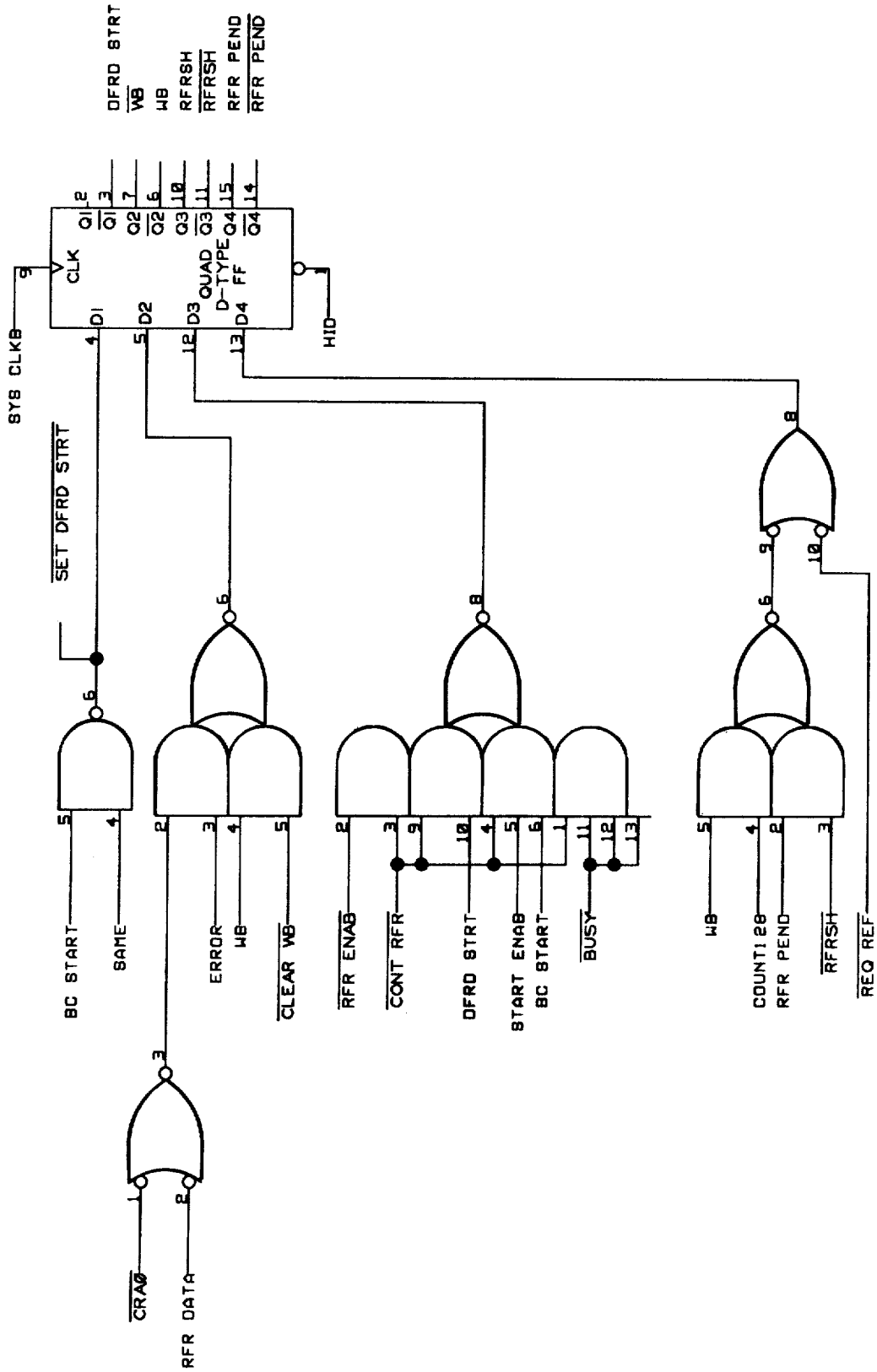

DFRD STRT
WB
WB
RFRSH
RFRSH
RFR PEND
RFR PEND
Q1
Q1
Q2
Q2
Q3
Q3
Q4
Q4
CLK
QUAD D-TYPE FF
D1
D2
D3
D4
HID
SYS CLKB
SET DFRD STRT
BC START
SAME
ERROR
WB
CLEAR WB
RFR ENAB
CONT RFR
DFRD STRT
START ENAB
BC START
BUSY
WB
COUNT128
RFR PEND
RFRSH
REQ REF
CRAØ
RFR DATA

FIG. 58C

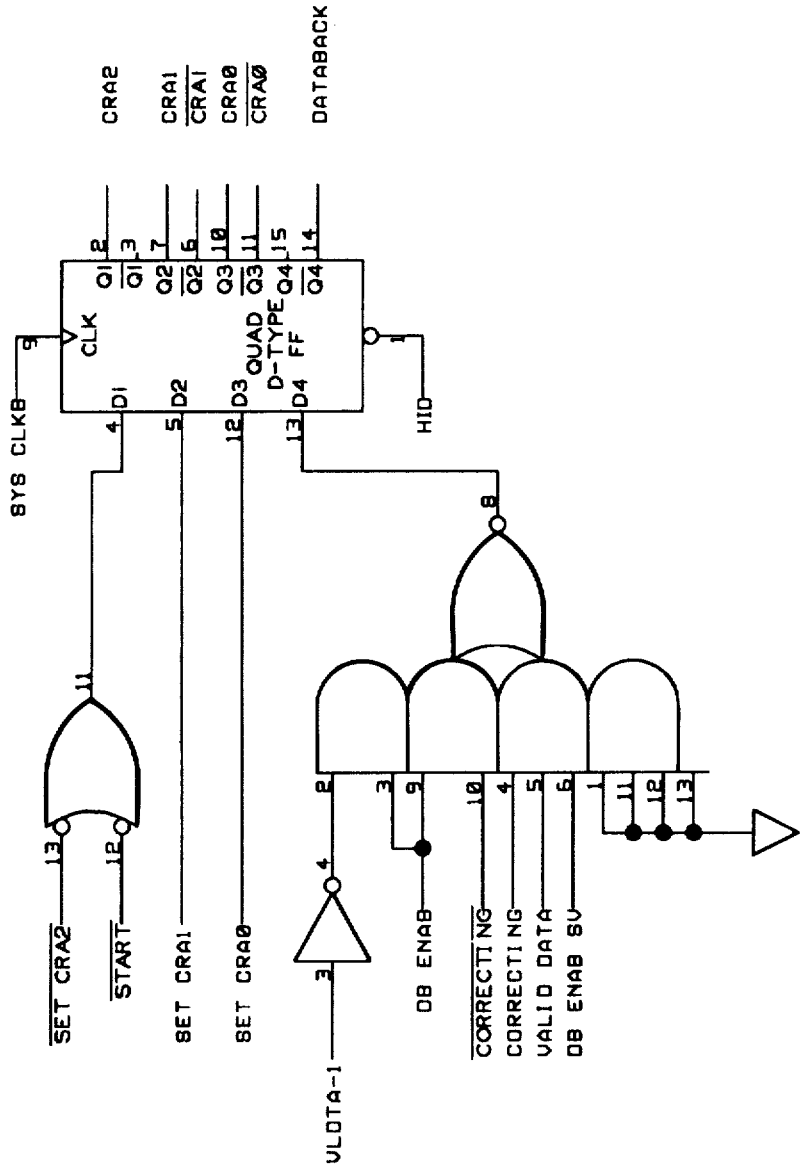

CRA2
CRA1
CRA1
CRA0
CRA0
DATABACK
Q1
Q1
Q2
Q2
Q3
Q3
Q4
Q4
CLK
QUAD
D-TYPE
FF
D1
D2
D3
D4
SYS CLKB
HID
SET CRA2
START
SET CRA1
SET CRA0
VLDTA-1
DB ENAB
CORRECTING
CORRECTING
VALID DATA
DB ENAB SV

FIG. 58D

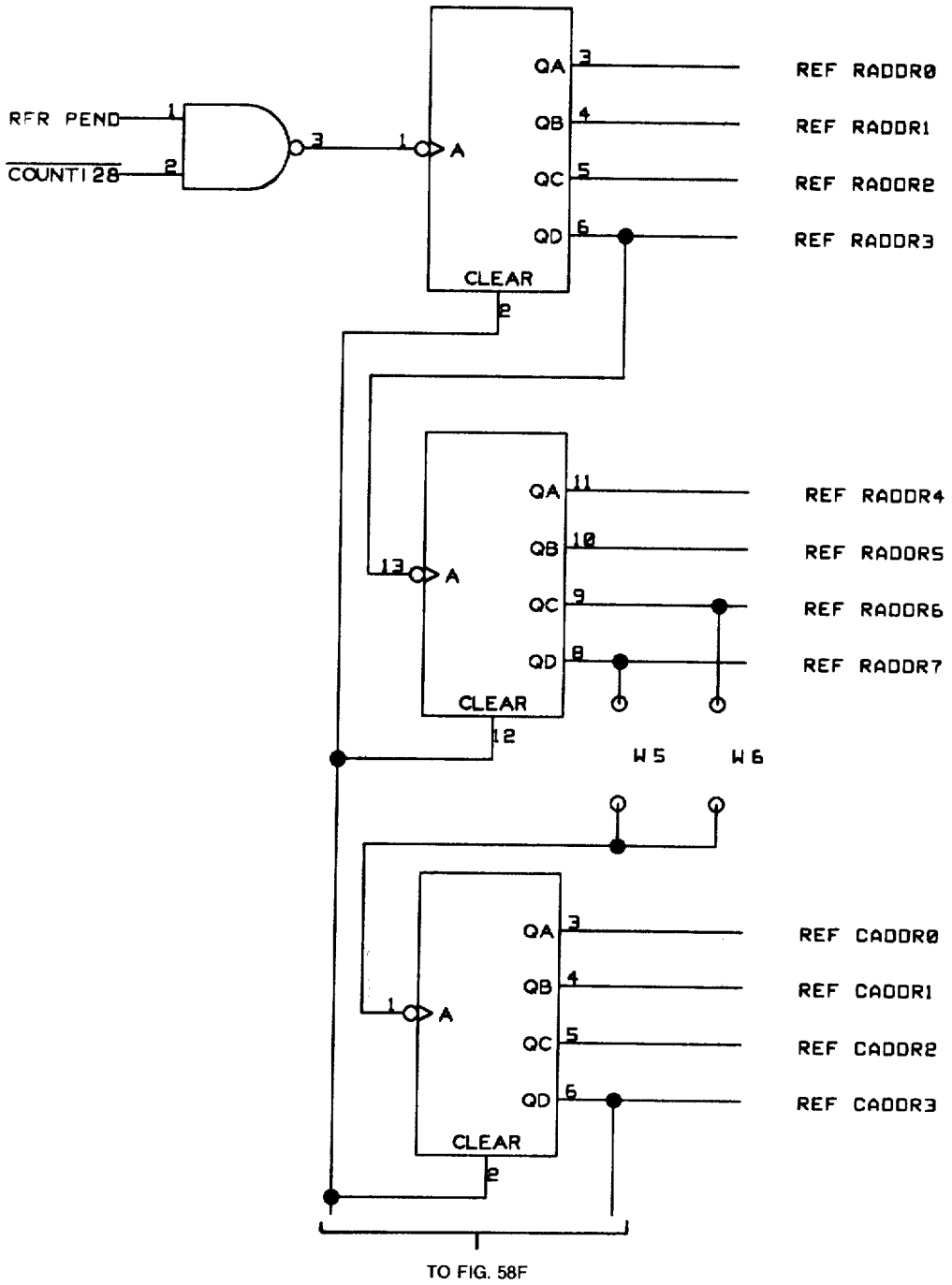

REF PEND
COUNT128
QA
QB
QC
QD
CLEAR
A
REF RADDR0
REF RADDR1
REF RADDR2
REF RADDR3
REF RADDR4
REF RADDR5
REF RADDR6
REF RADDR7
W5
W6
REF CADDR0
REF CADDR1
REF CADDR2
REF CADDR3
TO FIG. 58F

FIG. 58E

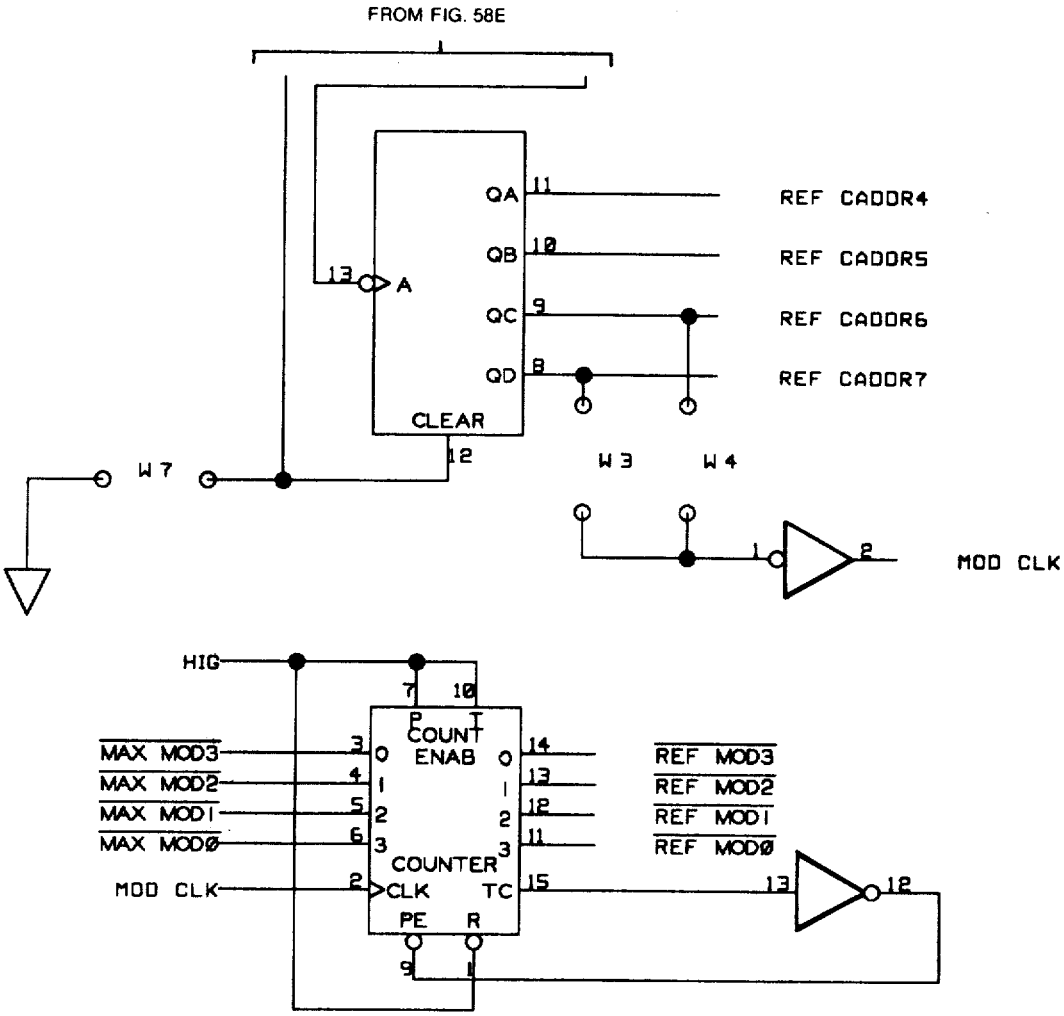
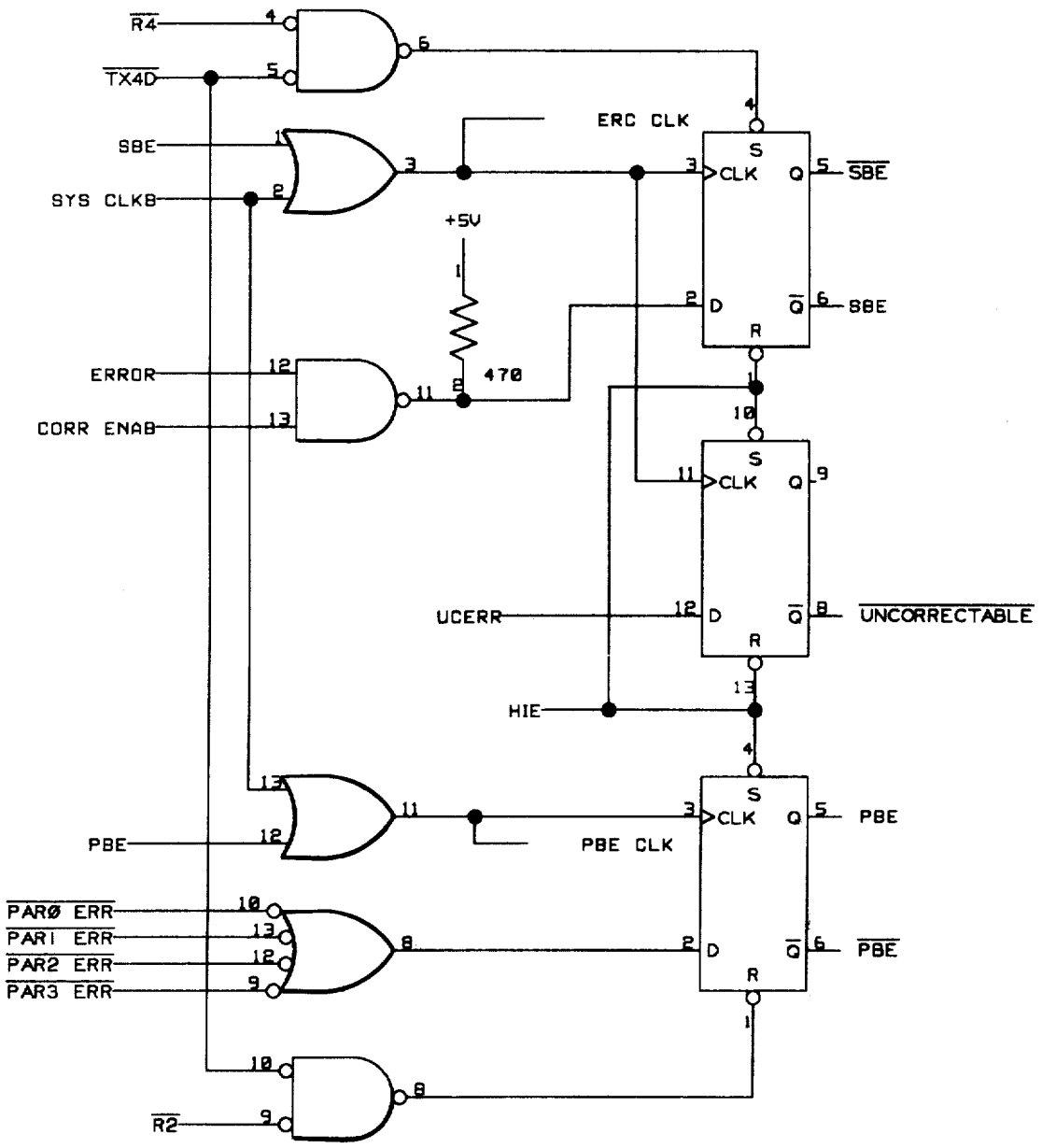

FROM FIG. 58E
QA
QB
QC
QD
A
CLEAR
REF CADDR4
REF CADDR5
REF CADDR6
REF CADDR7
W7
W3
W4
MOD CLK
HIG
COUNT ENAB
COUNTER
CLK
TC
PE
R
MAX MOD3
MAX MOD2
MAX MOD1
MAX MOD0
MOD CLK
REF MOD3
REF MOD2
REF MOD1
REF MOD0

FIG. 58F



PARITY LOGIC CONTROL LOGIC

FIG. 59

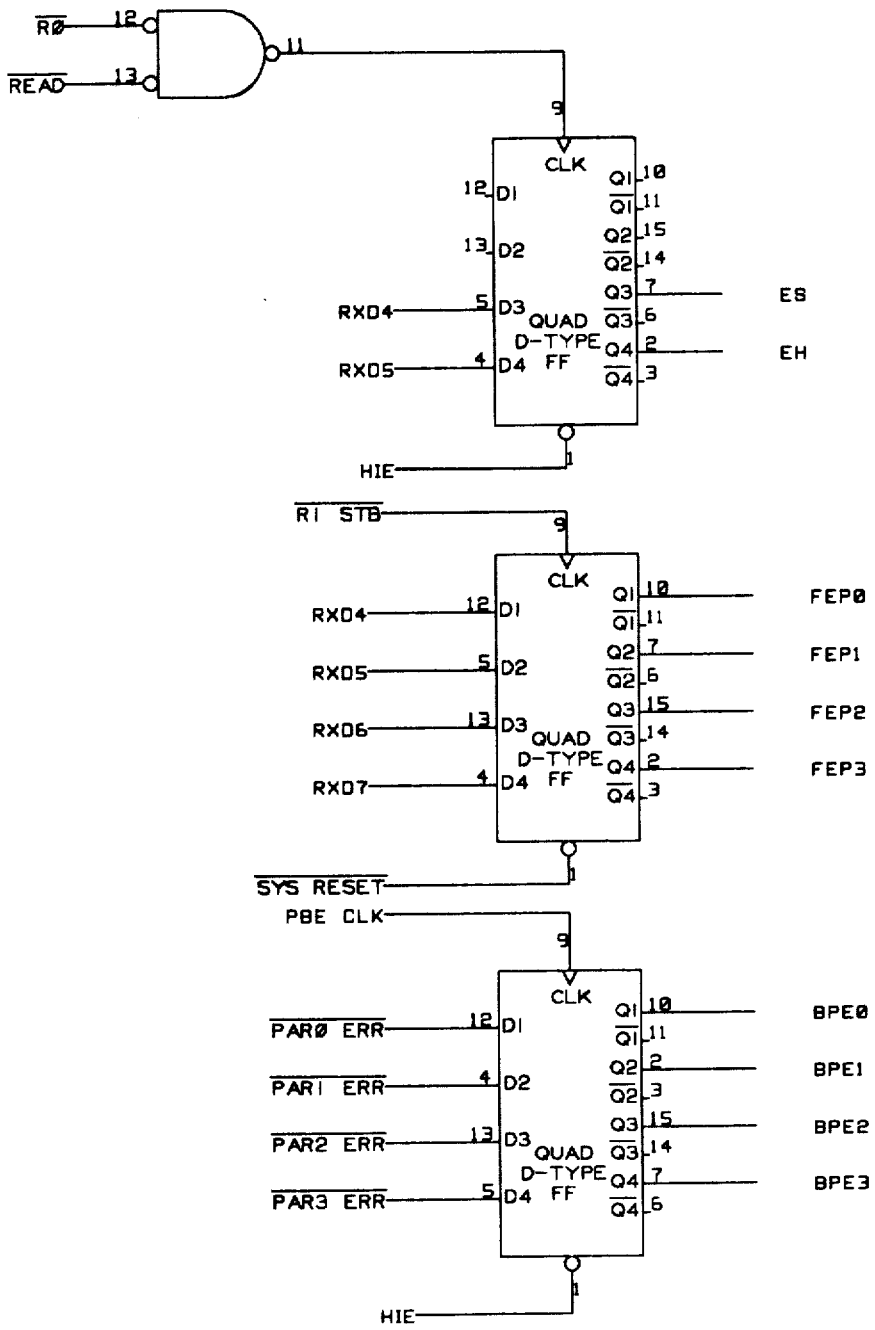
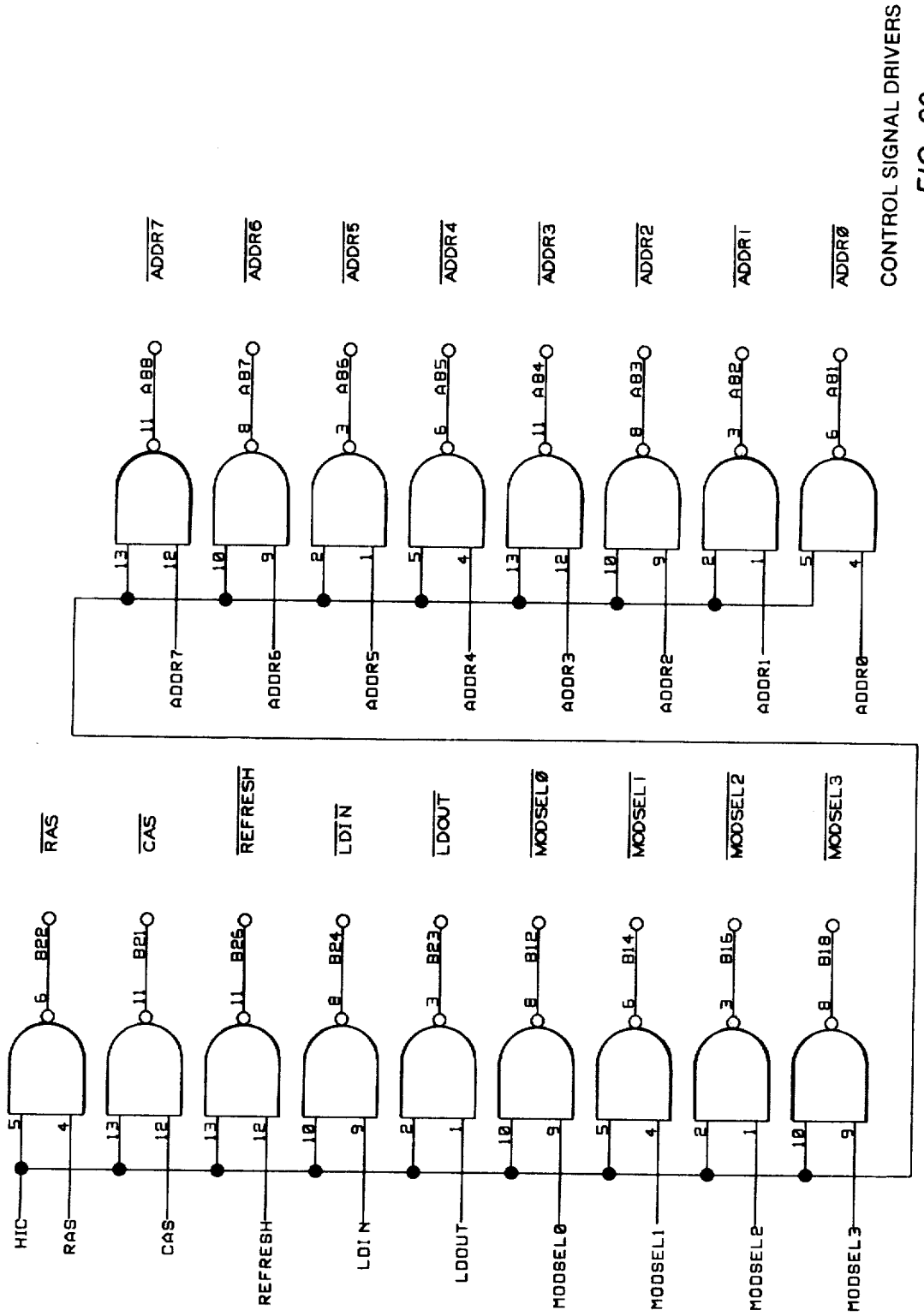
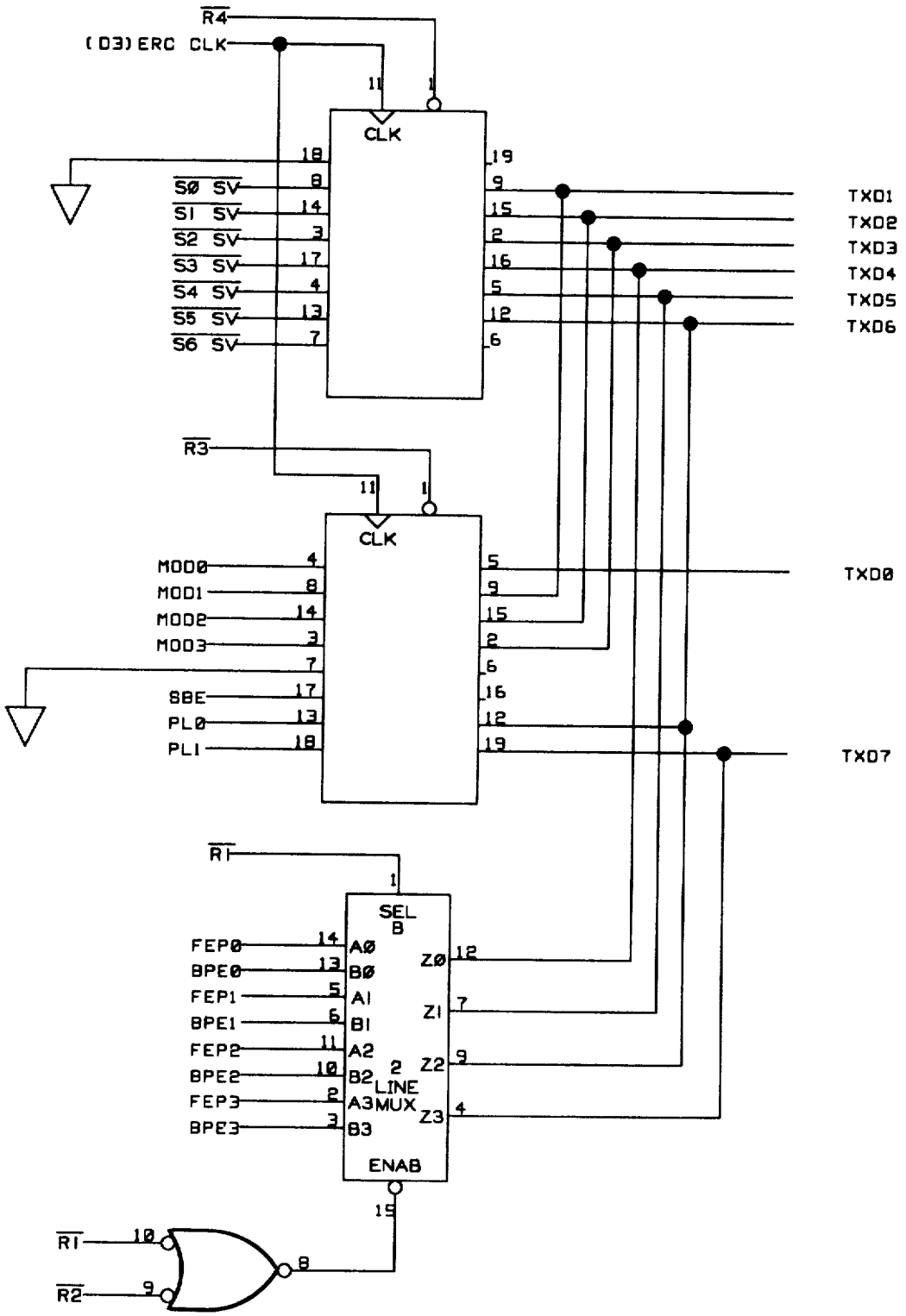

RØ
READ
CLK
D1
D2
D3
D4
QUAD
D-TYPE
FF
Q1
Q2
Q3
Q4
RXD4
RXD5
ES
EH
HIE
R1 STB
RXD6
RXD7
FEPØ
FEP1
FEP2
FEP3
SYS RESET
PBE CLK
PARØ ERR
PAR1 ERR
PAR2 ERR
PAR3 ERR
BPEØ
BPE1
BPE2
BPE3

FIG. 59A



CONTROL SIGNAL DRIVERS

FIG. 60



CBUS INTERFACE LOGIC

FIG. 61

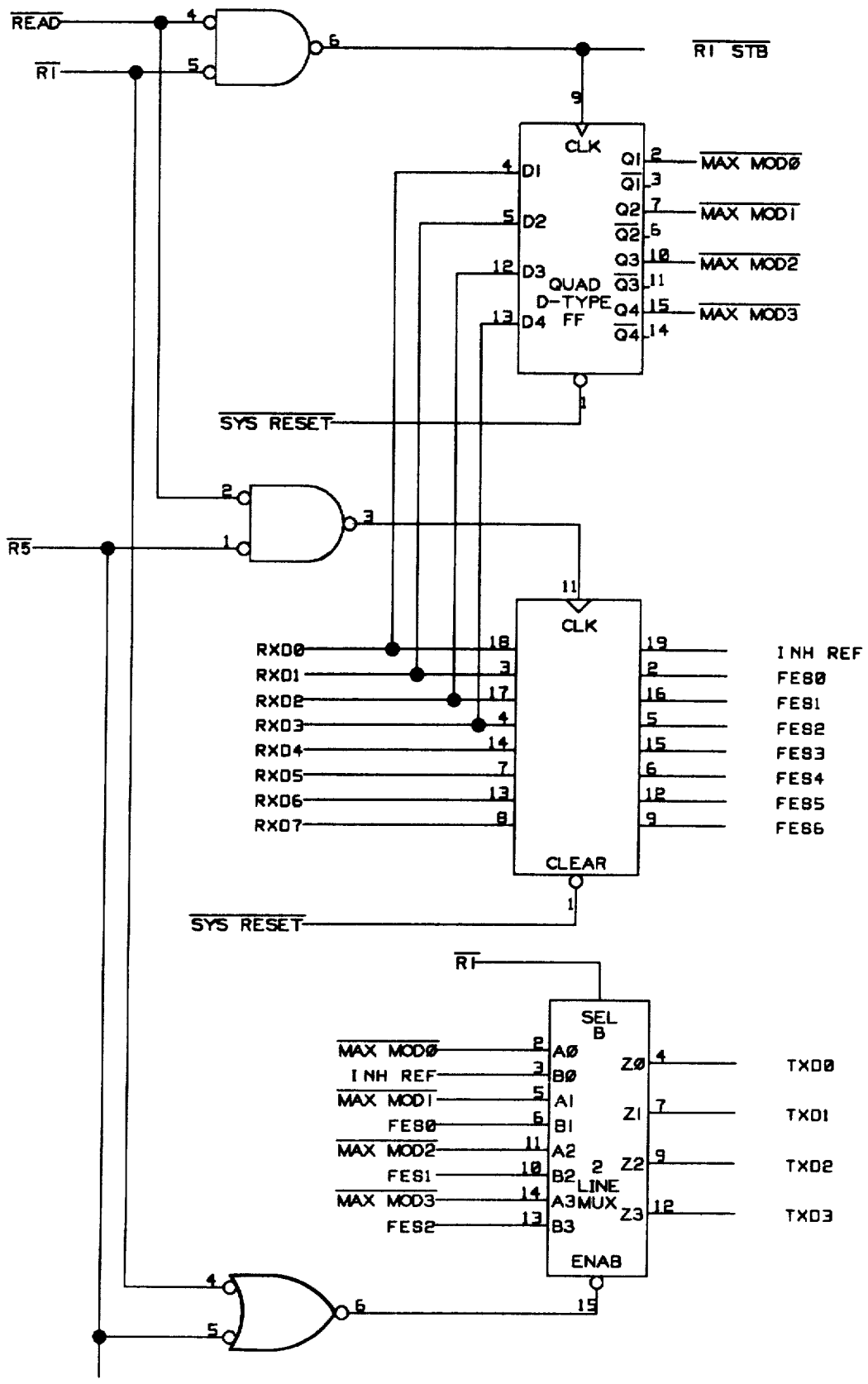

READ
RI
RI STB
CLK
D1
D2
D3
D4
QUAD D-TYPE FF
Q1
Q2
Q3
Q4
MAX MOD0
MAX MOD1
MAX MOD2
MAX MOD3
SYS RESET
R5
RXD0
RXD1
RXD2
RXD3
RXD4
RXD5
RXD6
RXD7
CLEAR
INH REF
FES0
FES1
FES2
FES3
FES4
FES5
FES6
SEL B
A0
B0
A1
B1
A2
B2
A3
B3
2 LINE MUX
Z0
Z1
Z2
Z3
ENAB
TXD0
TXD1
TXD2
TXD3
TO FIG. 61B

FIG. 61A

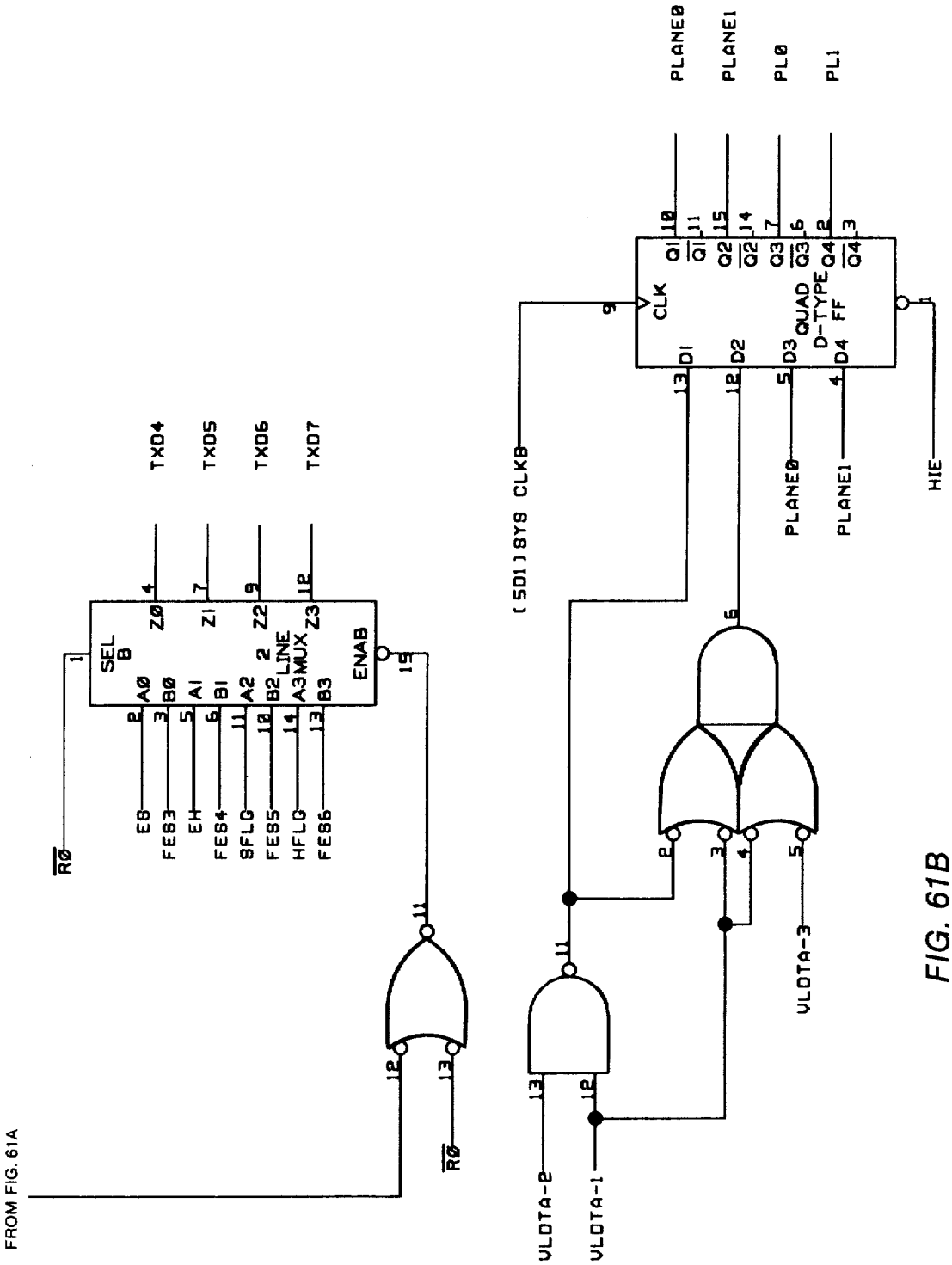
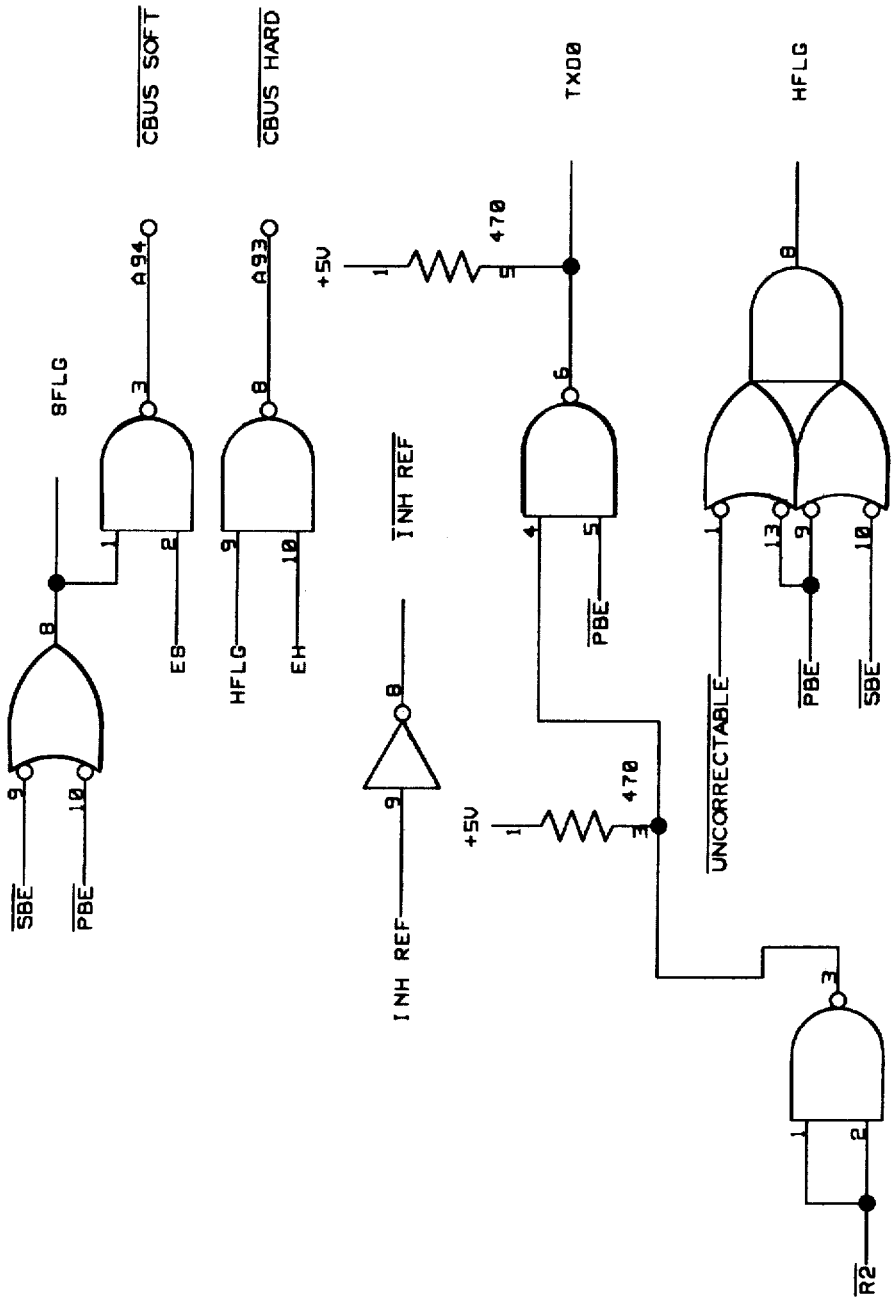
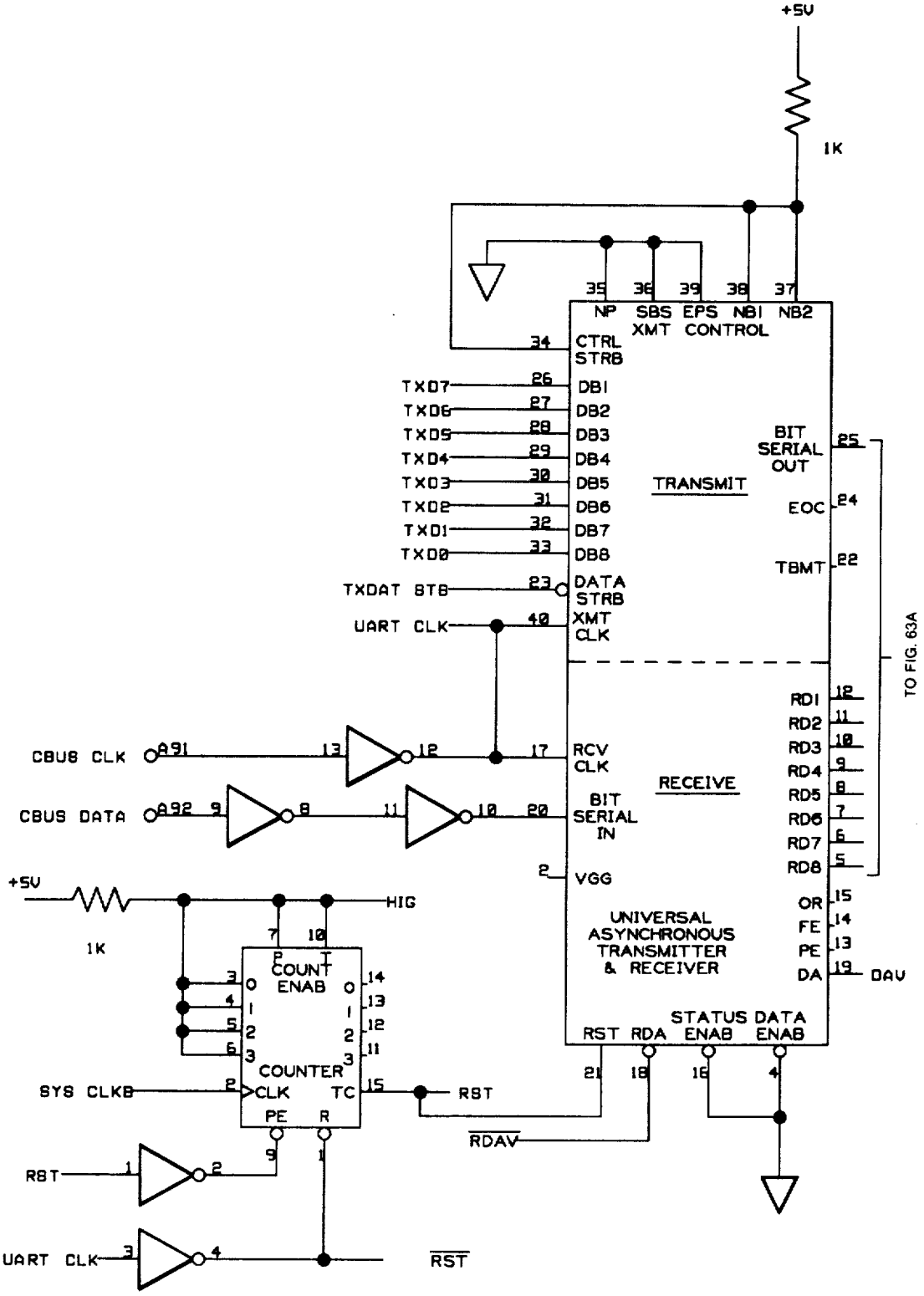

FROM FIG. 61A
SEL B
ENAB
2 LINE MUX
TXD4
TXD5
TXD6
TXD7
ES
FES3
EH
FES4
SFLG
FES5
HFLG
FES6
RØ
(SDI) SYS CLKB
CLK
QUAD D-TYPE FF
PLANEØ
PLANE1
PLØ
PL1
HIE
VLDTA-2
VLDTA-1
VLDTA-3

FIG. 61B



CBUS INTERFACE LOGIC

FIG. 62



CBUS INTERFACE LOGIC

FIG. 63

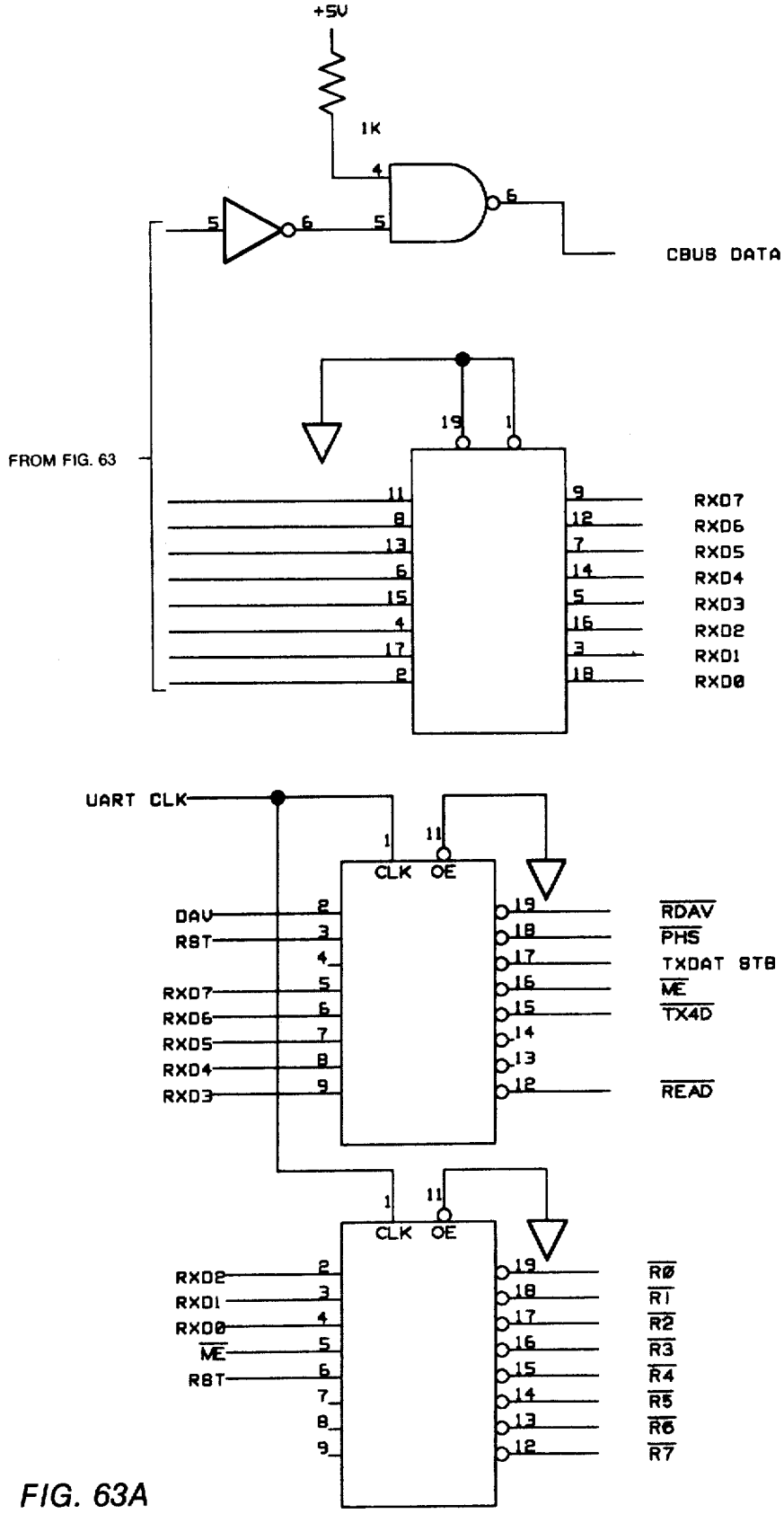
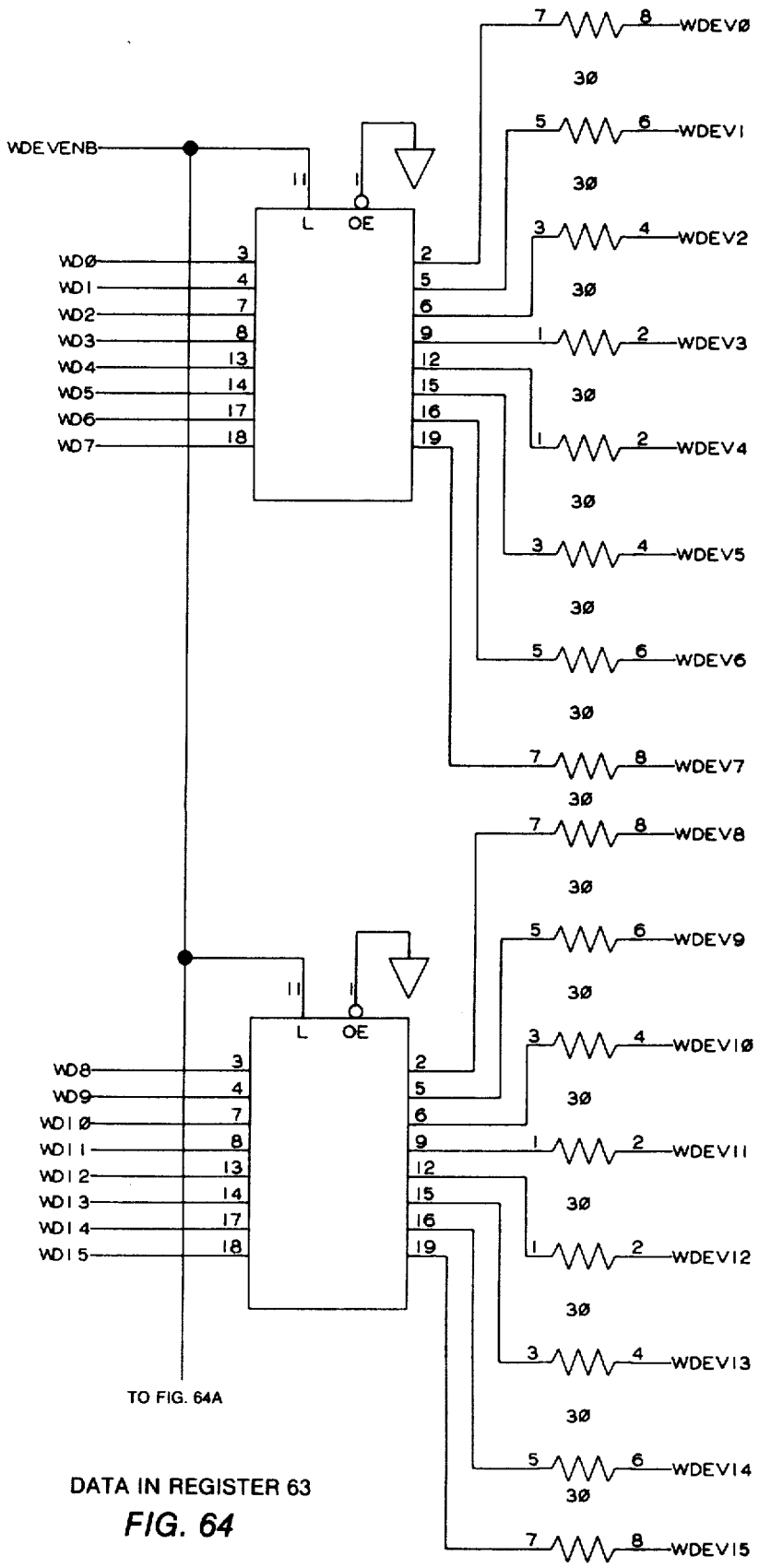

+5V
1K
CBUS DATA
FROM FIG. 63
RXD7
RXD6
RXD5
RXD4
RXD3
RXD2
RXD1
RXD0
UART CLK
CLK
OE
DAV
RST
RDAV
PHS
TXDAT STB
ME
TX4D
READ
R0
R1
R2
R3
R4
R5
R6
R7

FIG. 63A



DATA IN REGISTER 63

FIG. 64

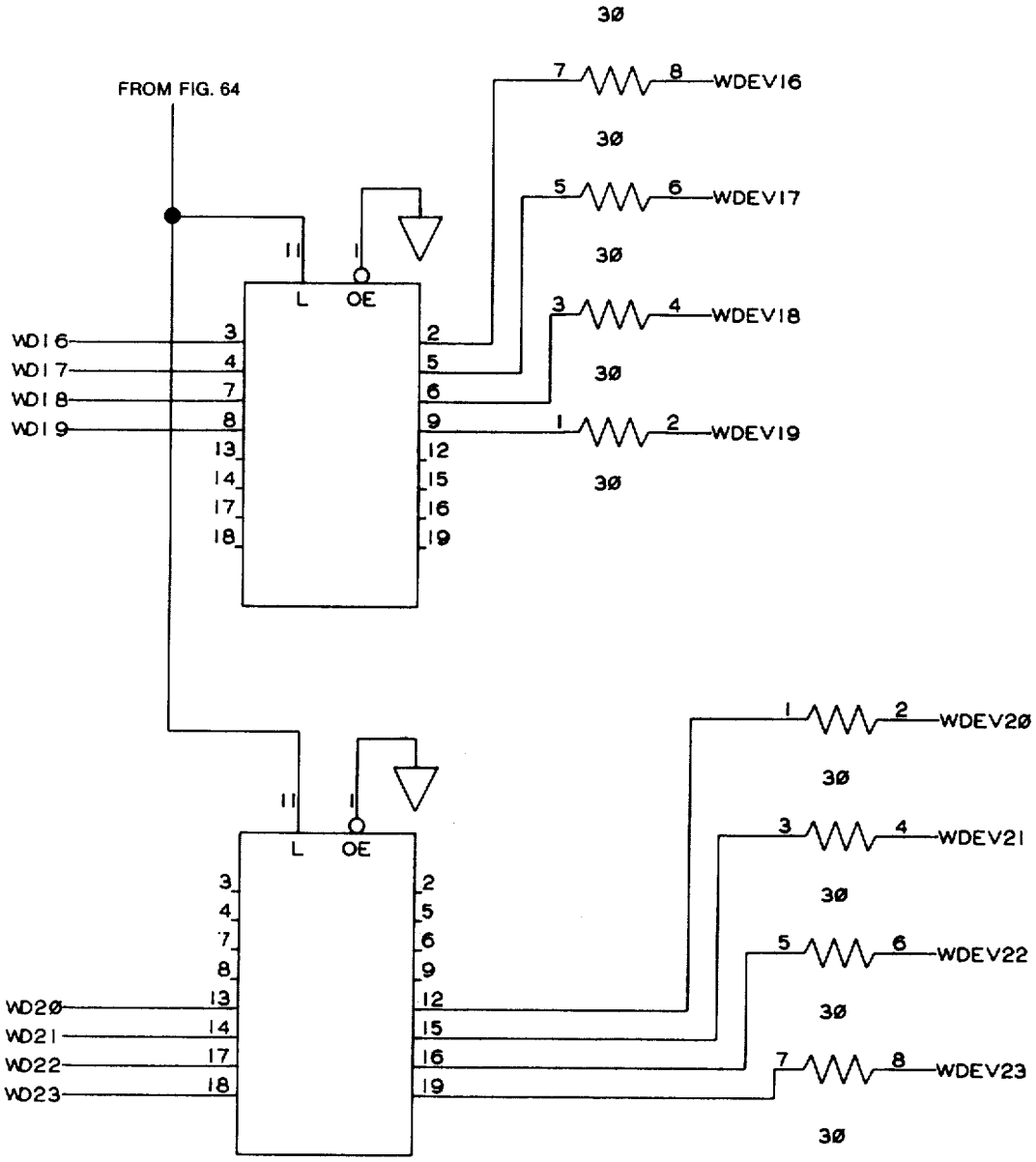

FROM FIG. 64
WD16
WD17
WD18
WD19
L
OE
WDEV16
WDEV17
WDEV18
WDEV19
WD20
WD21
WD22
WD23
WDEV20
WDEV21
WDEV22
WDEV23
30

FIG. 64A

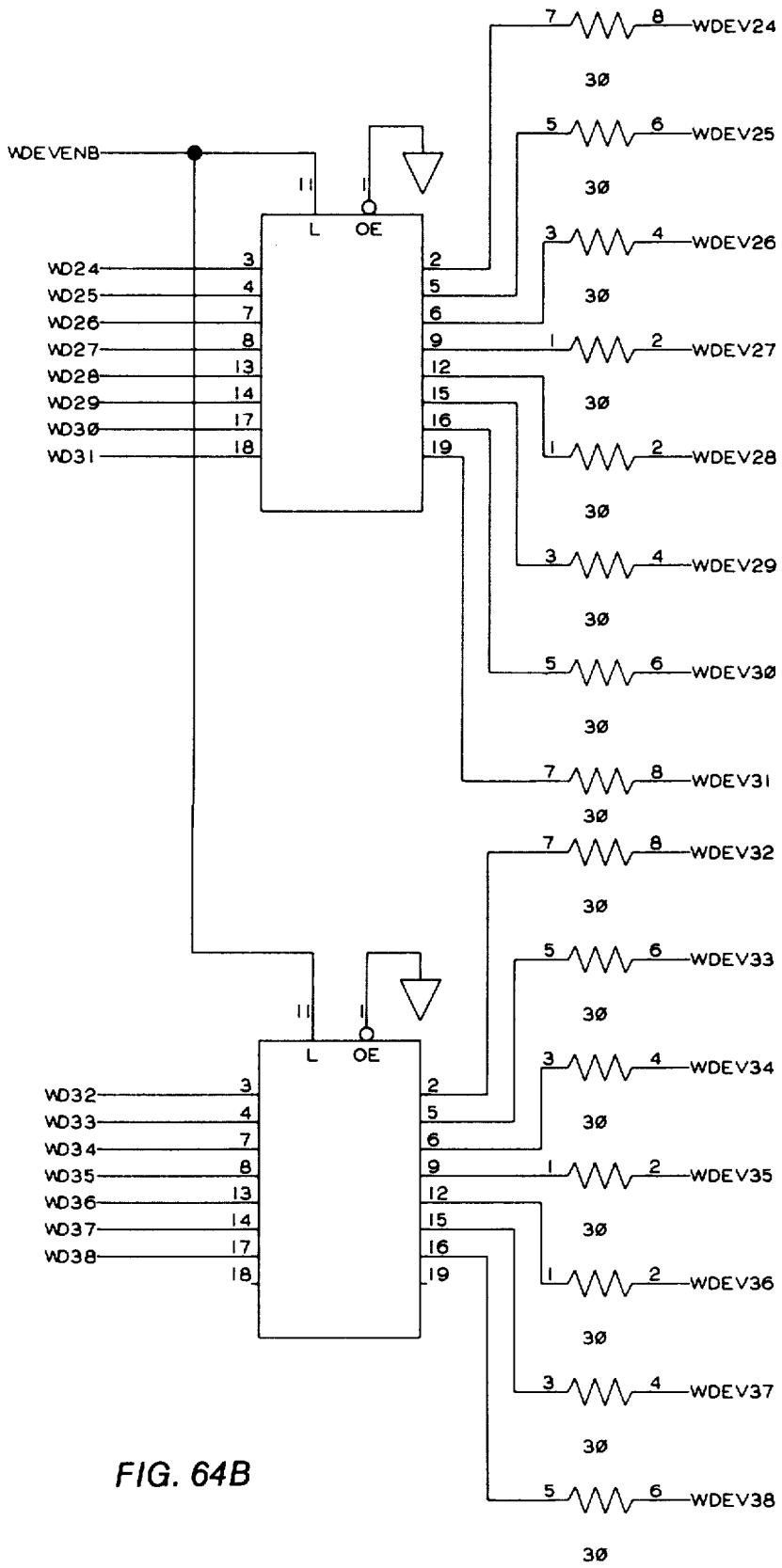
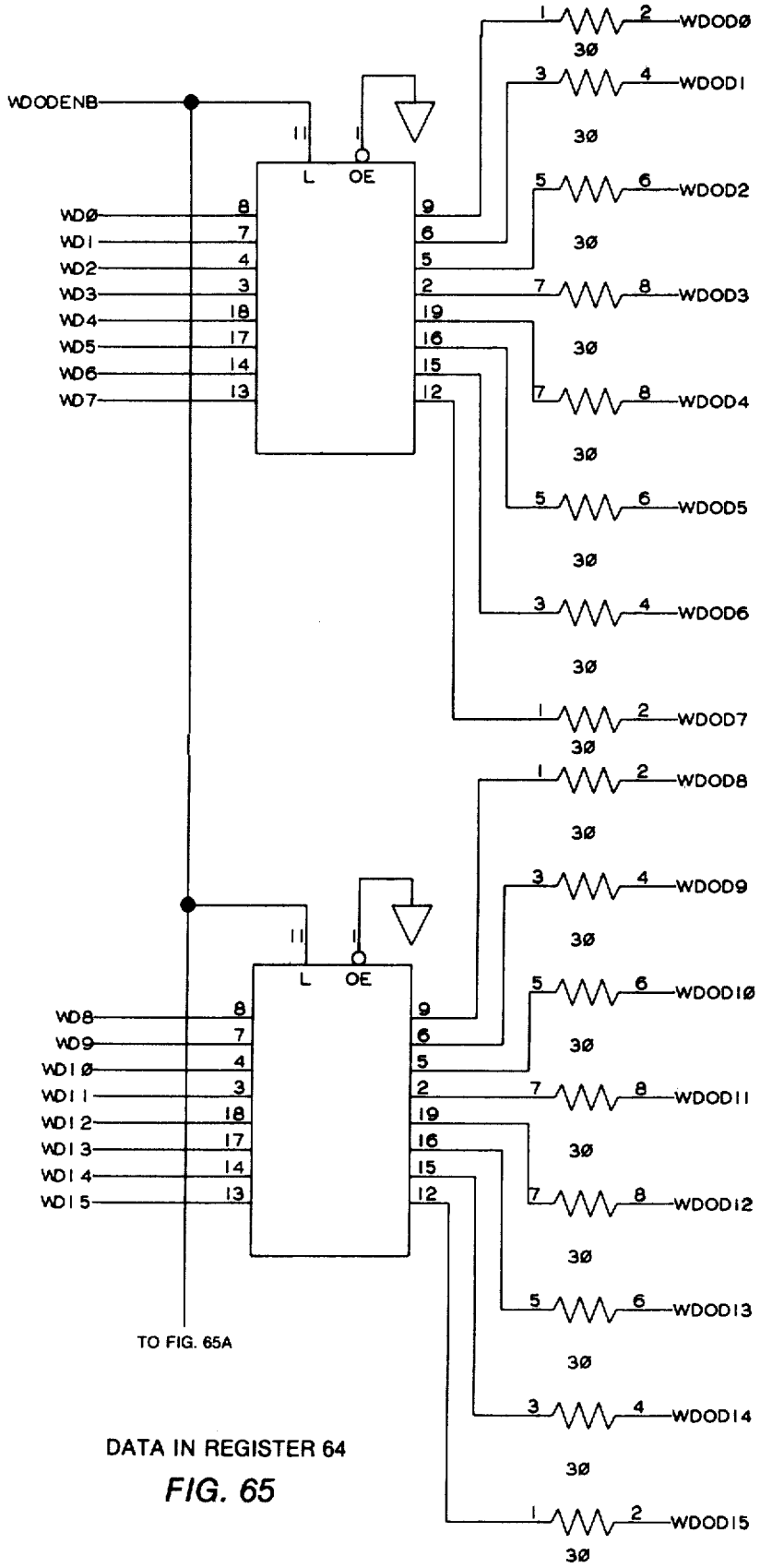

WDEVENB
L
OE
WD24
WD25
WD26
WD27
WD28
WD29
WD30
WD31
WDEV24
WDEV25
WDEV26
WDEV27
WDEV28
WDEV29
WDEV30
WDEV31
WD32
WD33
WD34
WD35
WD36
WD37
WD38
WDEV32
WDEV33
WDEV34
WDEV35
WDEV36
WDEV37
WDEV38
30

FIG. 64B



DATA IN REGISTER 64

FIG. 65

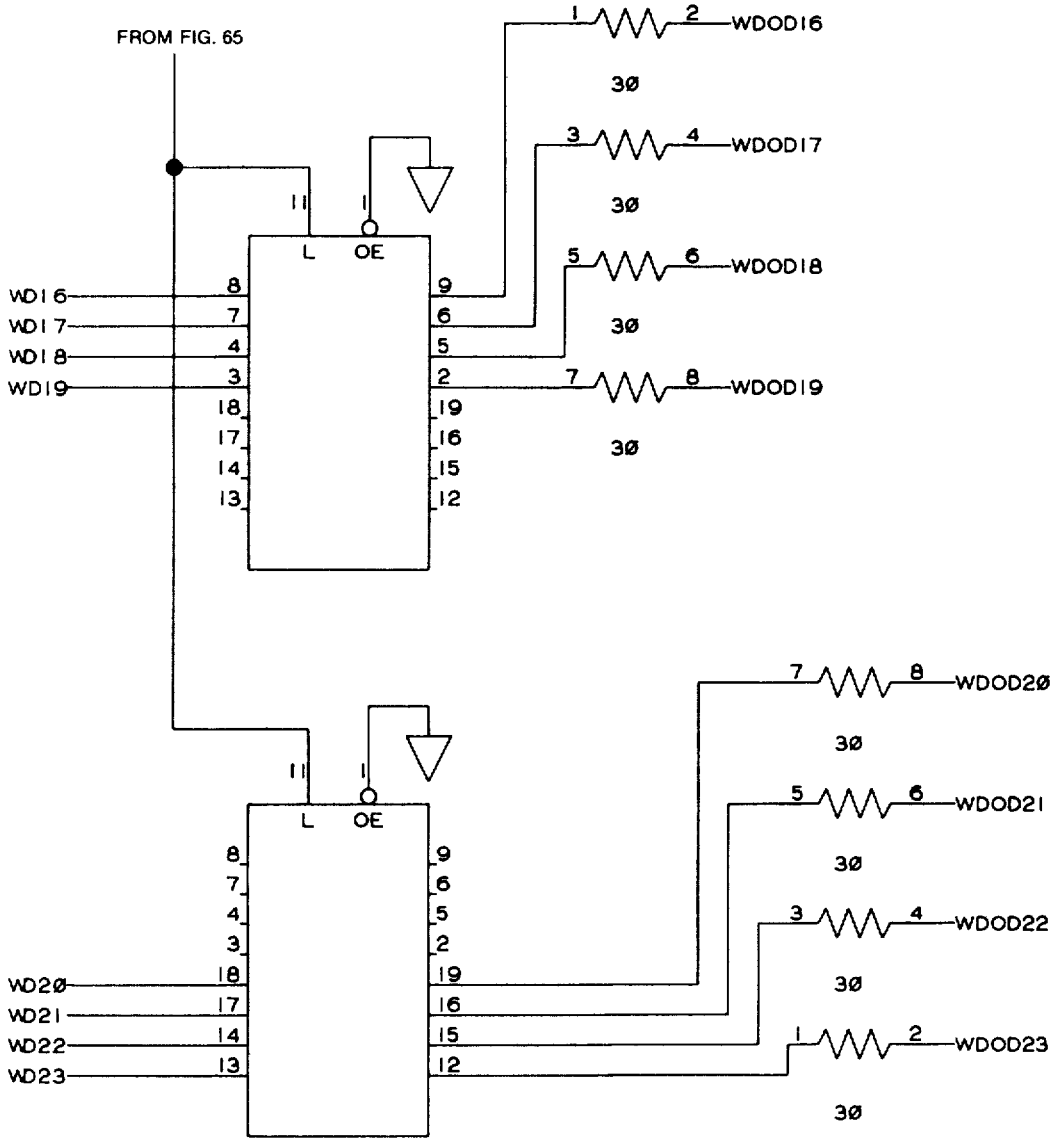

FROM FIG. 65
WD16
WD17
WD18
WD19
WD20
WD21
WD22
WD23
L
OE
WDOD16
WDOD17
WDOD18
WDOD19
WDOD20
WDOD21
WDOD22
WDOD23
30

FIG. 65A

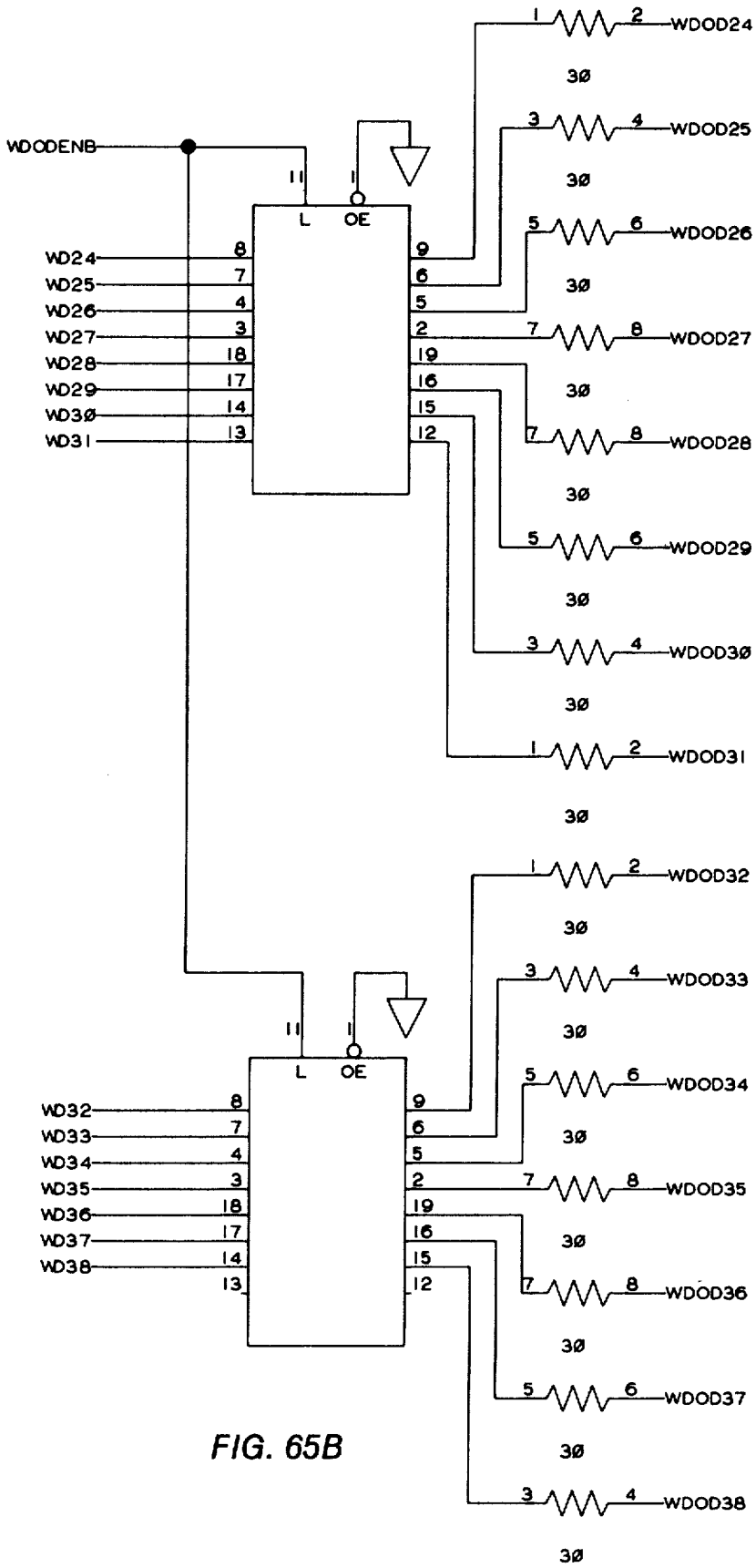
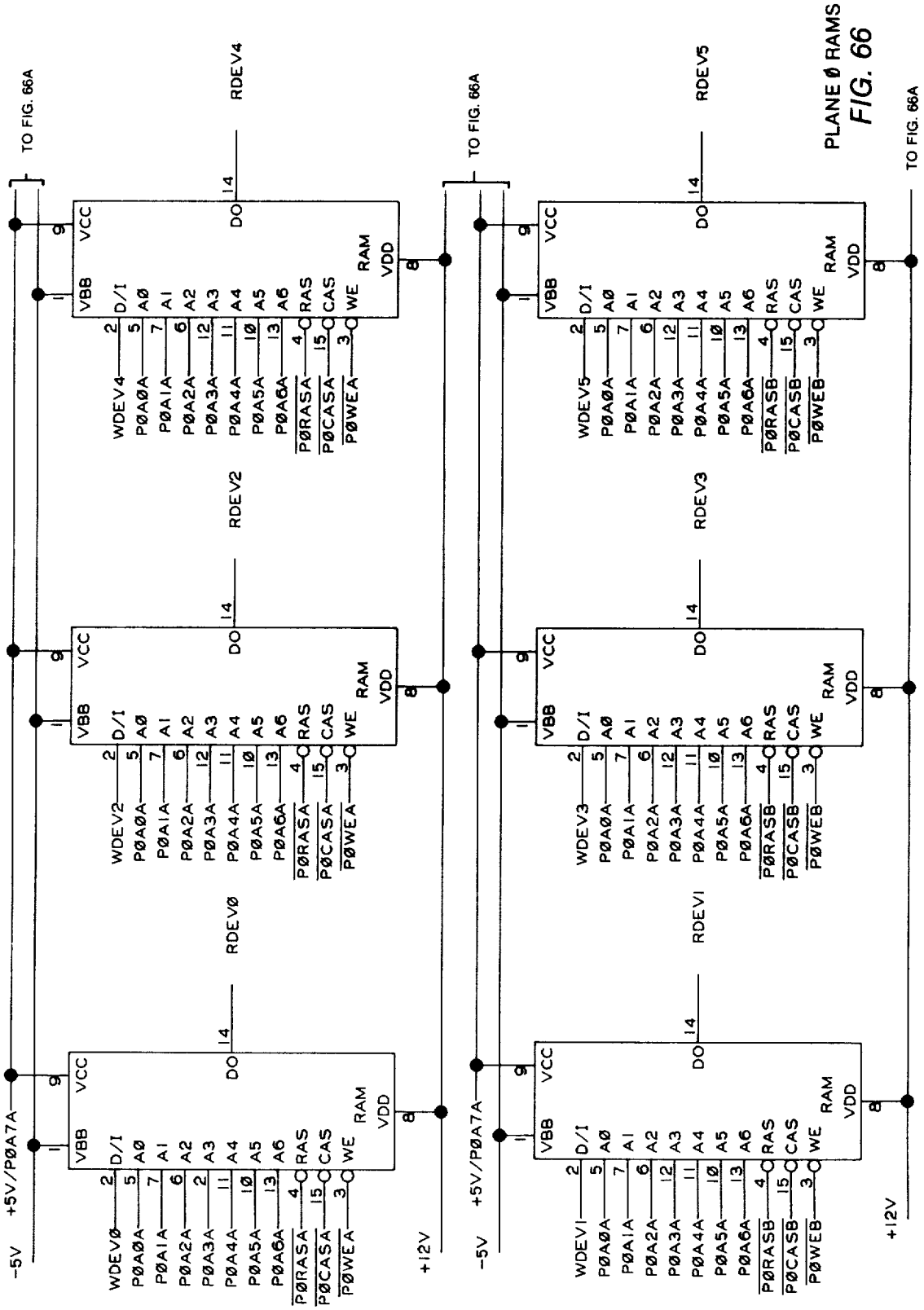

WDODENB
L
OE
WD24
WD25
WD26
WD27
WD28
WD29
WD30
WD31
WDOD24
WDOD25
WDOD26
WDOD27
WDOD28
WDOD29
WDOD30
WDOD31
WD32
WD33
WD34
WD35
WD36
WD37
WD38
WDOD32
WDOD33
WDOD34
WDOD35
WDOD36
WDOD37
WDOD38
30

FIG. 65B



PLANE Ø RAMS
FIG. 66

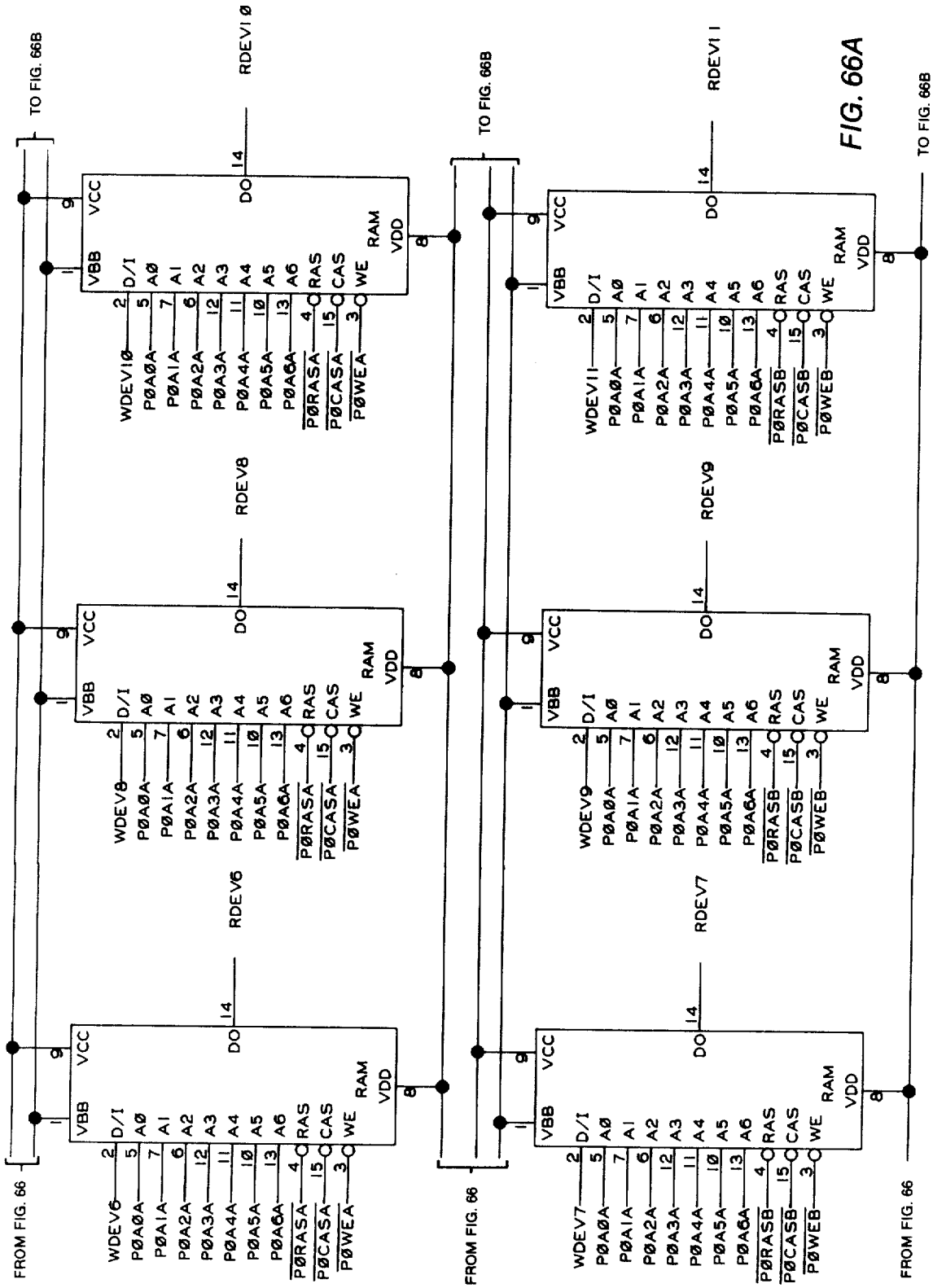

TO FIG. 66B
RDEV10
RDEV8
RDEV6
FROM FIG. 66
TO FIG. 66B
RDEV11
RDEV9
RDEV7
FROM FIG. 66
TO FIG. 66B
FROM FIG. 66
VCC
VBB
DO
RAM
VDD
D/I
A0
A1
A2
A3
A4
A5
A6
RAS
CAS
WE
WDEV10
WDEV8
WDEV6
WDEV11
WDEV9
WDEV7
P0A0A
P0A1A
P0A2A
P0A3A
P0A4A
P0A5A
P0A6A
P0RASA
P0CASA
P0WEA
P0RASB
P0CASB
P0WEB

FIG. 66A

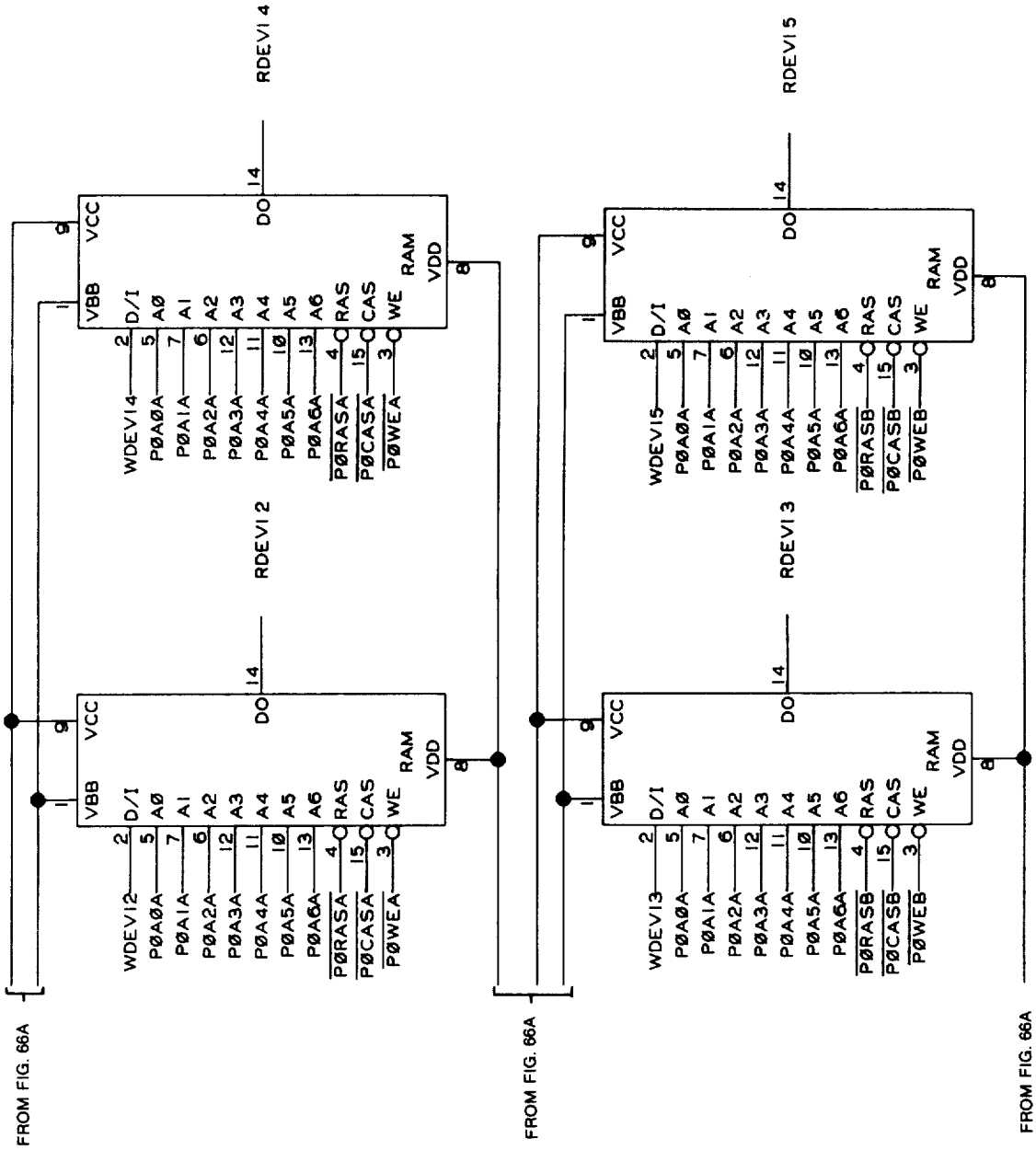

FROM FIG. 66A
WDEVI2
PØA0A
PØA1A
PØA2A
PØA3A
PØA4A
PØA5A
PØA6A
PØRASA
PØCASA
PØWEA
VBB
VCC
D/I
A0
A1
A2
A3
A4
A5
A6
RAS
CAS
WE
DO
RAM
VDD
RDEVI2
WDEVI4
RDEVI4
FROM FIG. 66A
WDEVI3
PØRASB
PØCASB
PØWEB
RDEVI3
WDEVI5
RDEVI5
FROM FIG. 66A

FIG. 66B

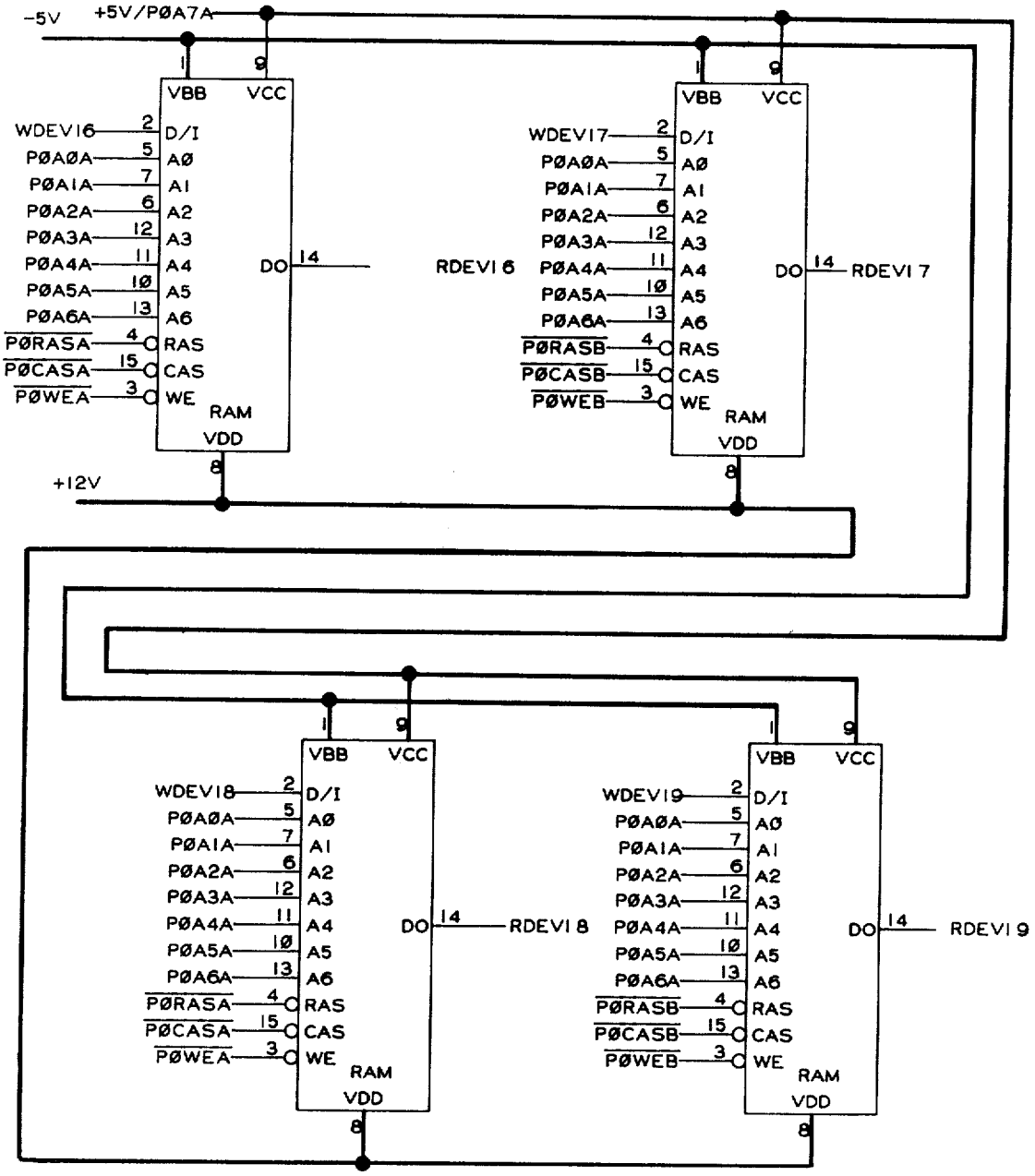

-5V
+5V/PØA7A
VBB
VCC
WDEVI6
D/I
PØAØA
AØ
PØA1A
A1
PØA2A
A2
PØA3A
A3
PØA4A
A4
PØA5A
A5
PØA6A
A6
PØRASA
RAS
PØCASA
CAS
PØWEA
WE
DO
RDEVI 6
RAM
VDD
+12V
WDEVI7
PØRASB
PØCASB
PØWEB
RDEVI 7
WDEVI8
RDEVI 8
WDEVI9
RDEVI 9

FIG. 66C

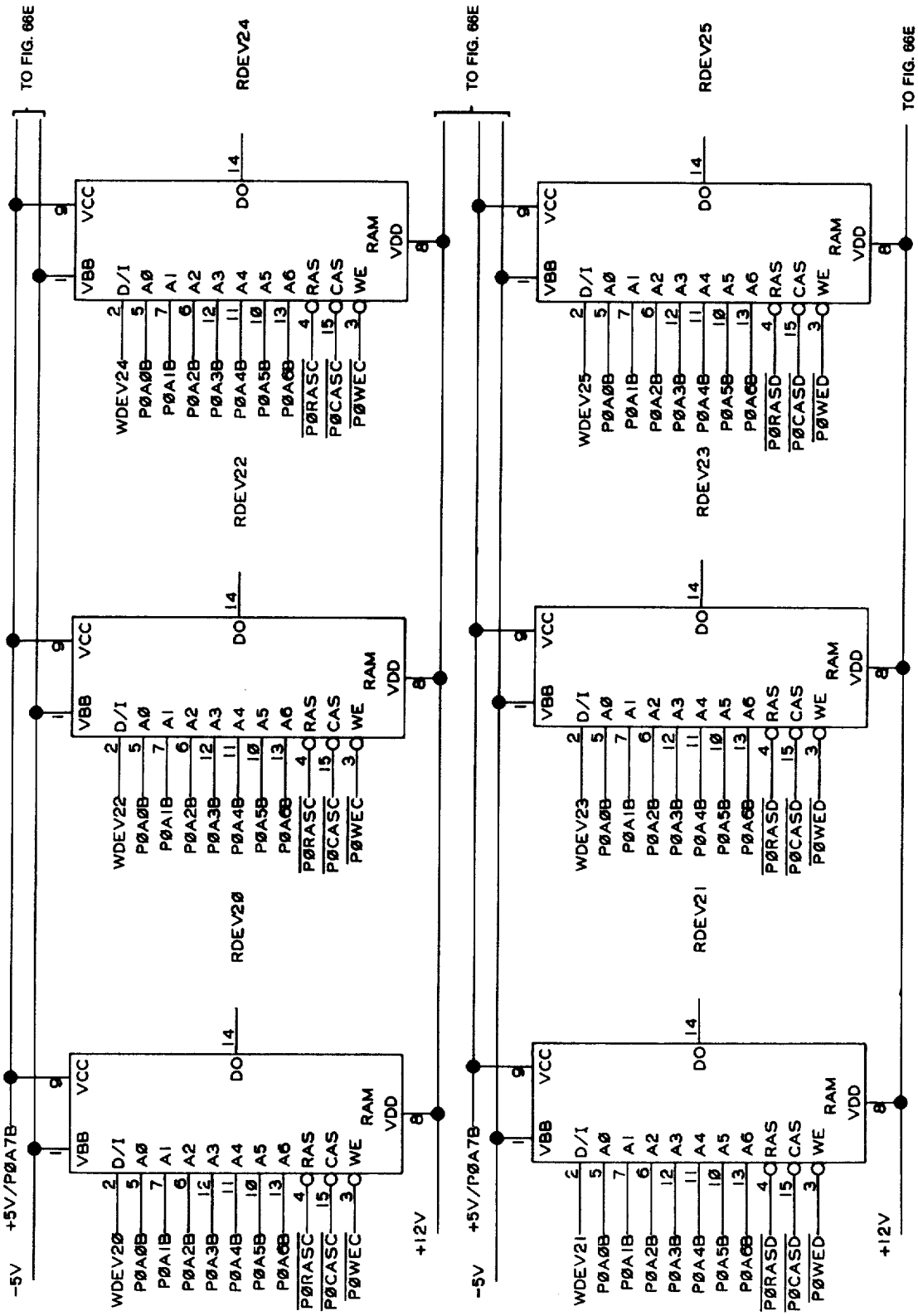

TO FIG. 66E
RDEV24
RDEV22
RDEV20
RDEV25
RDEV23
RDEV21
VCC
VBB
DO
RAM
VDD
D/I
A0
A1
A2
A3
A4
A5
A6
RAS
CAS
WE
WDEV24
WDEV22
WDEV20
WDEV25
WDEV23
WDEV21
P0A0B
P0A1B
P0A2B
P0A3B
P0A4B
P0A5B
P0A6B
P0RASC
P0CASC
P0WEC
P0RASD
P0CASD
P0WED
-5V
+5V/P0A7B
+12V

FIG. 66D

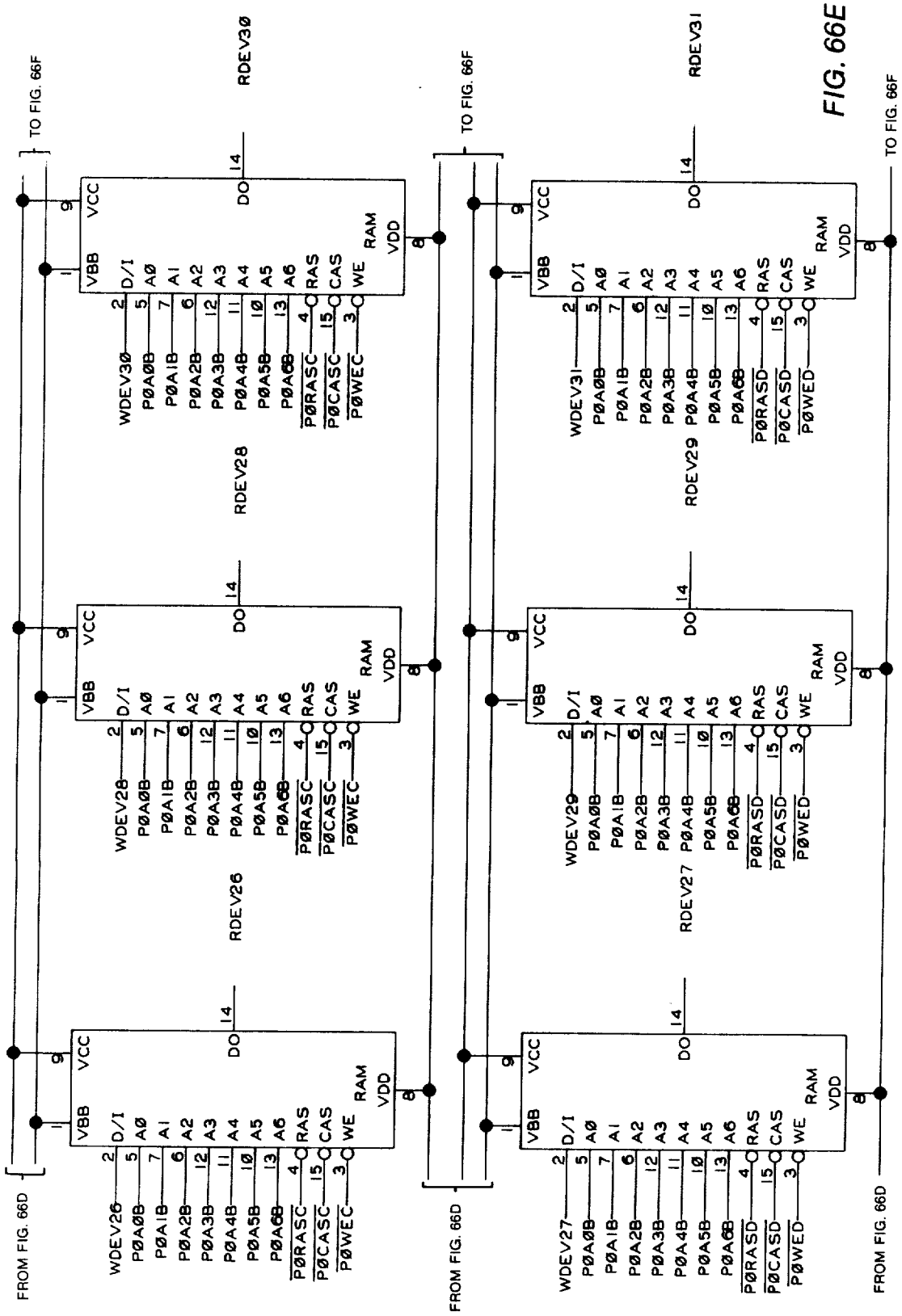

FROM FIG. 66D
TO FIG. 66F
RDEV26
RDEV28
RDEV30
RDEV27
RDEV29
RDEV31
VCC
VBB
DO
RAM
VDD
D/I
A0
A1
A2
A3
A4
A5
A6
RAS
CAS
WE
WDEV26
WDEV28
WDEV30
WDEV27
WDEV29
WDEV31
PØA0B
PØA1B
PØA2B
PØA3B
PØA4B
PØA5B
PØA6B
PØRASC
PØCASC
PØWEC
PØRASD
PØCASD
PØWED

FIG. 66E

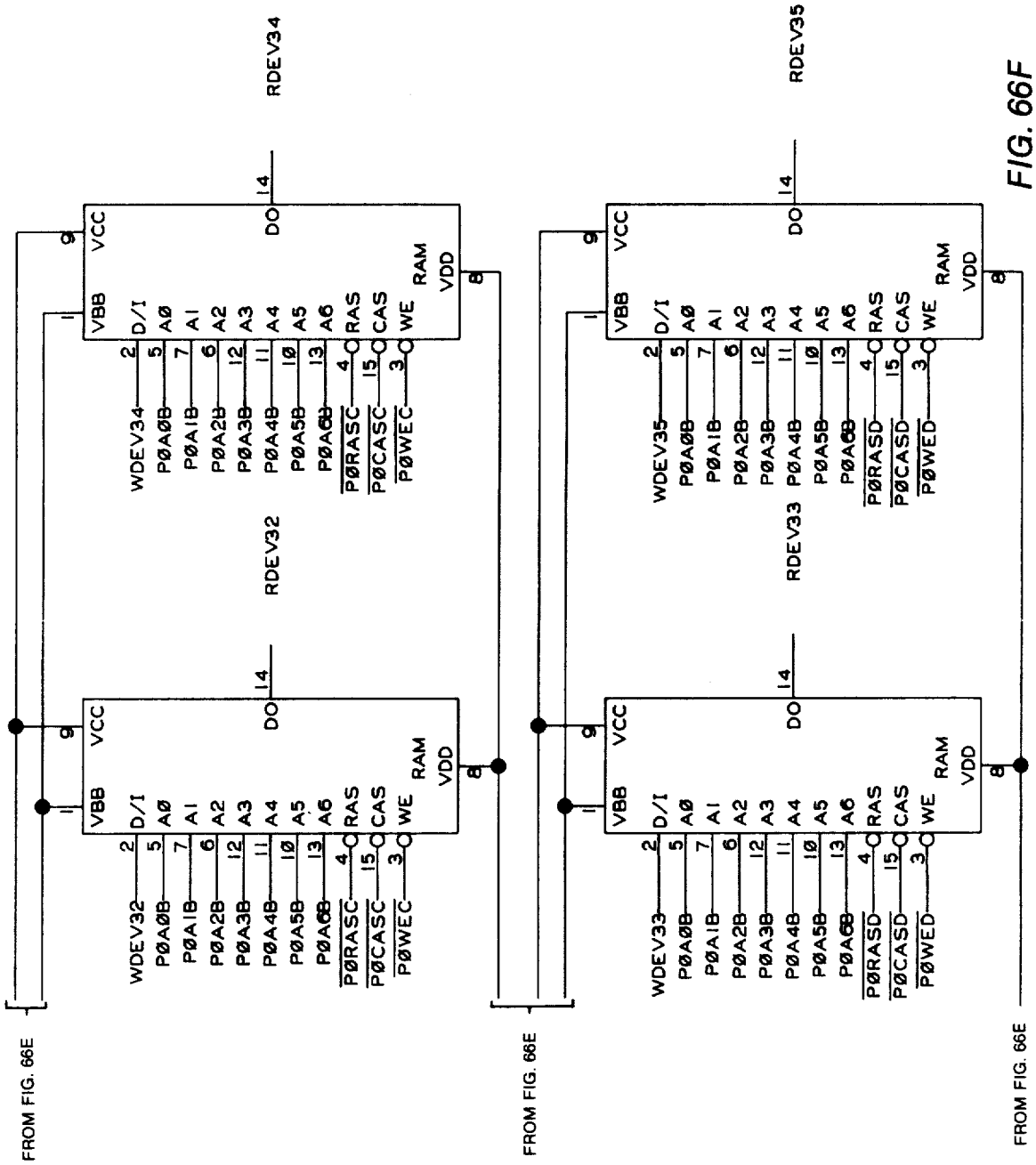

FROM FIG. 66E
RDEV34
RDEV32
RAM
VCC
VBB
VDD
DO
D/I
A0
A1
A2
A3
A4
A5
A6
RAS
CAS
WE
WDEV34
PØA0B
PØA1B
PØA2B
PØA3B
PØA4B
PØA5B
PØA6B
PØRASC
PØCASC
PØWEC
WDEV32
FROM FIG. 66E
RDEV35
RDEV33
PØRASD
PØCASD
PØWED
WDEV35
WDEV33
FROM FIG. 66E

FIG. 66F

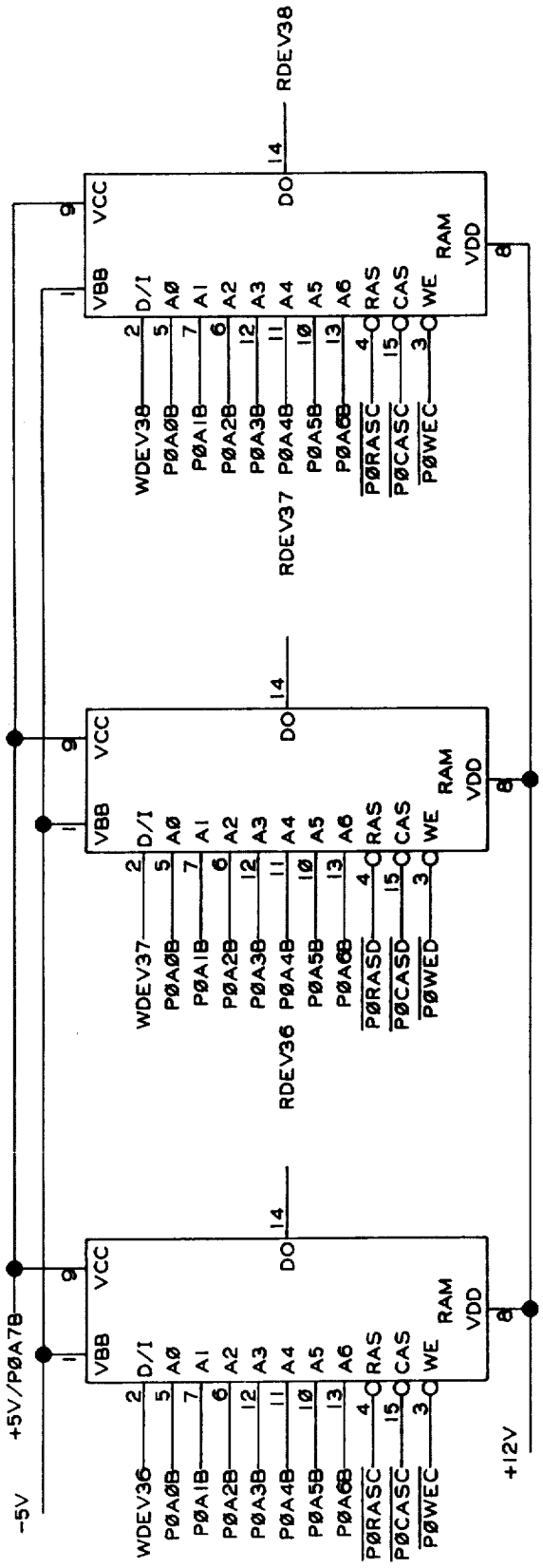
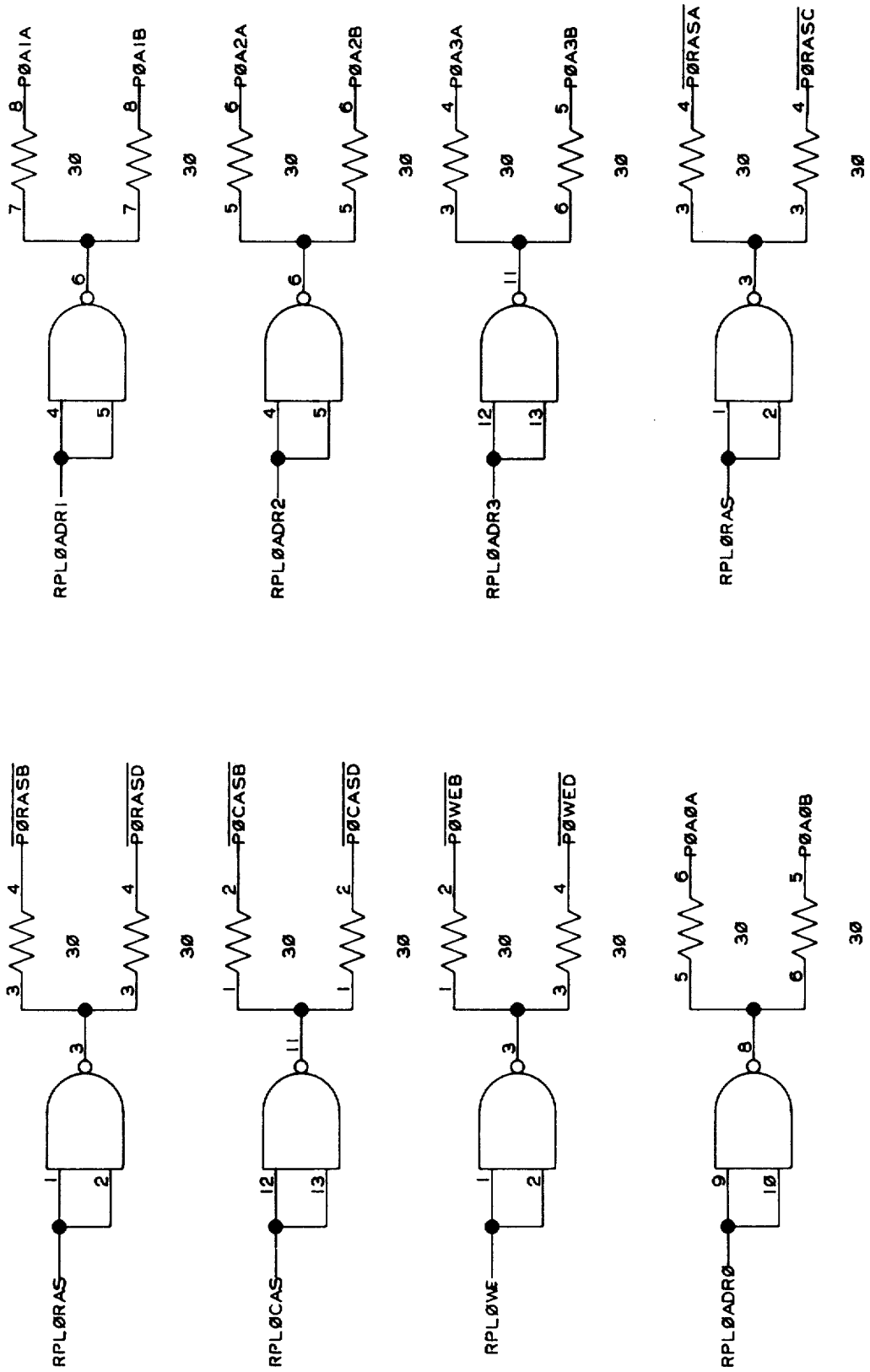

RDEV38
14
DO
VCC
9
VBB
1
D/I
A0
A1
A2
A3
A4
A5
A6
RAS
CAS
WE
RAM
VDD
8
WDEV38
PØA0B
PØA1B
PØA2B
PØA3B
PØA4B
PØA5B
PØA6B
PØRASC
PØCASC
PØWEC
RDEV37
WDEV37
PØRASD
PØCASD
PØWED
RDEV36
WDEV36
-5V
+5V/PØA7B
+12V

FIG. 66G



PLANE Ø CONTROL INPUTS

FIG. 67

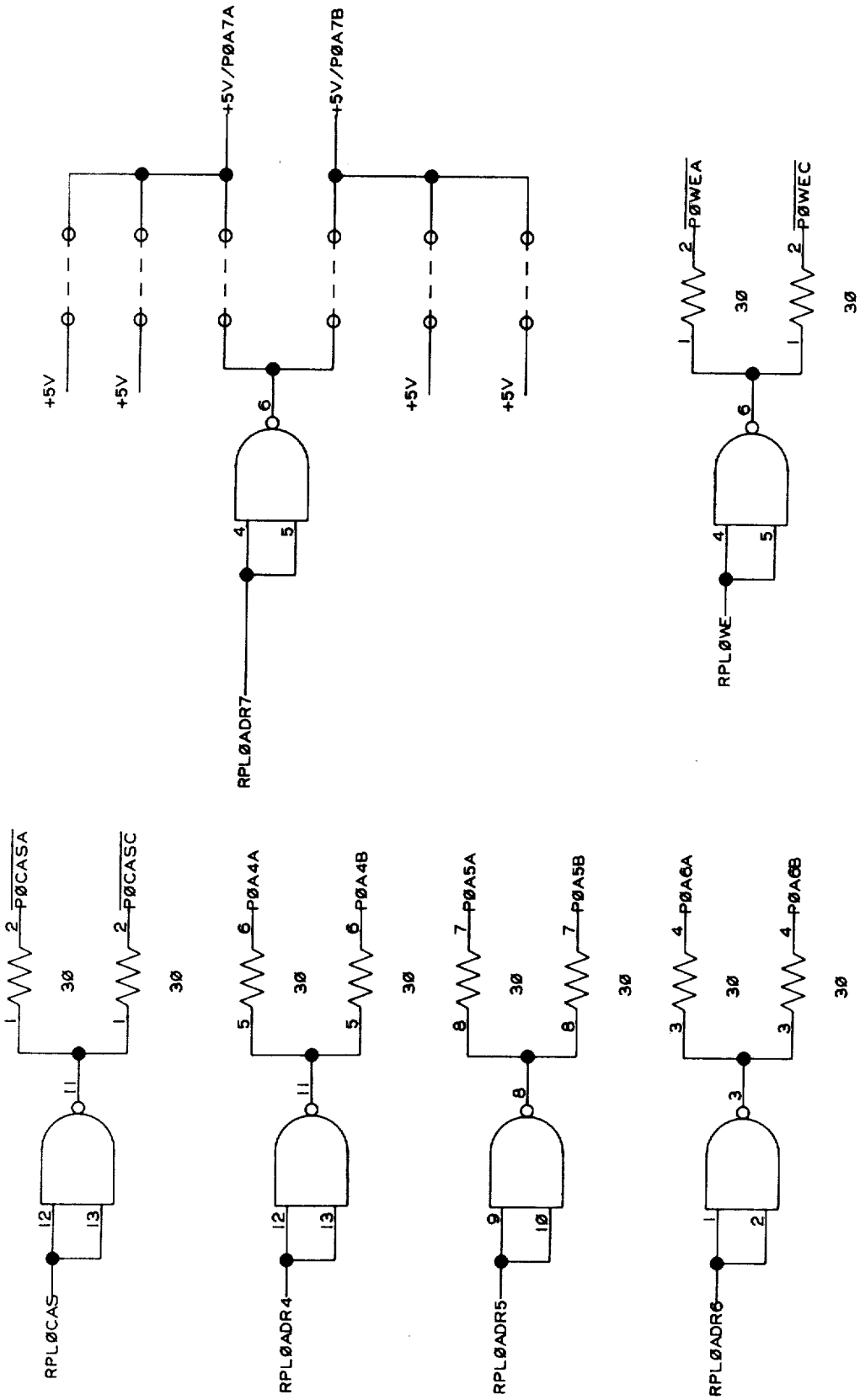
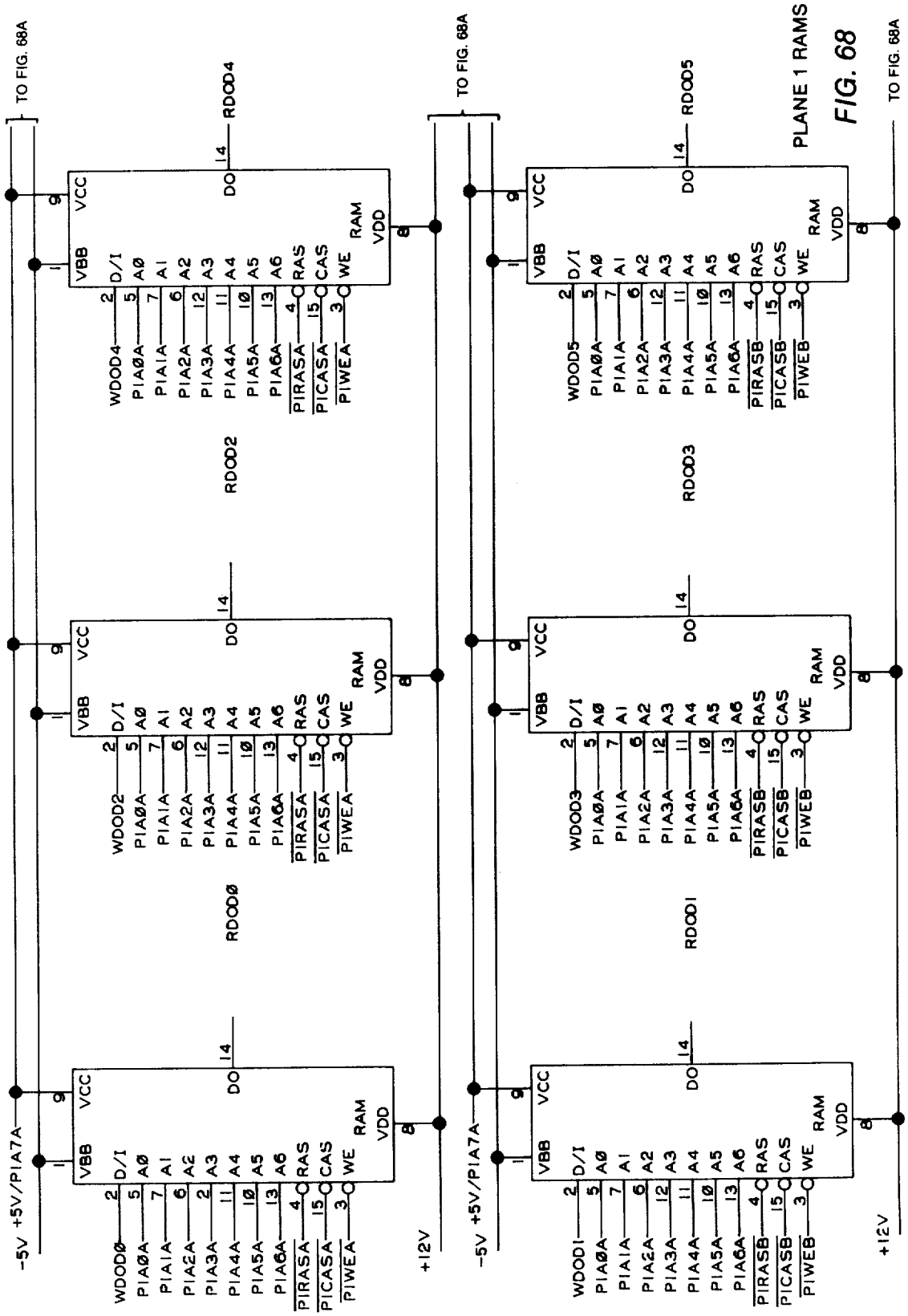

+5V
+5V
+5V/PØA7A
+5V/PØA7B
+5V
+5V
RPLØADR7
RPLØWE
PØWEA
PØWEC
3Ø
RPLØCAS
PØCASA
PØCASC
RPLØADR4
PØA4A
PØA4B
RPLØADR5
PØA5A
PØA5B
RPLØADR6
PØA6A
PØA6B

FIG. 67A


TO FIG. 68A
-5V +5V/PIA7A
+12V
RAM
VCC
VBB
VDD
DO
D/I
RAS
CAS
WE
WDOD0
WDOD2
WDOD4
WDOD1
WDOD3
WDOD5
PIA0A
PIA1A
PIA2A
PIA3A
PIA4A
PIA5A
PIA6A
PIRASA
PICASA
PIWEA
PIRASB
PICASB
PIWEB
RDOD0
RDOD2
RDOD4
RDOD1
RDOD3
RDOD5
PLANE 1 RAMS
FIG. 68

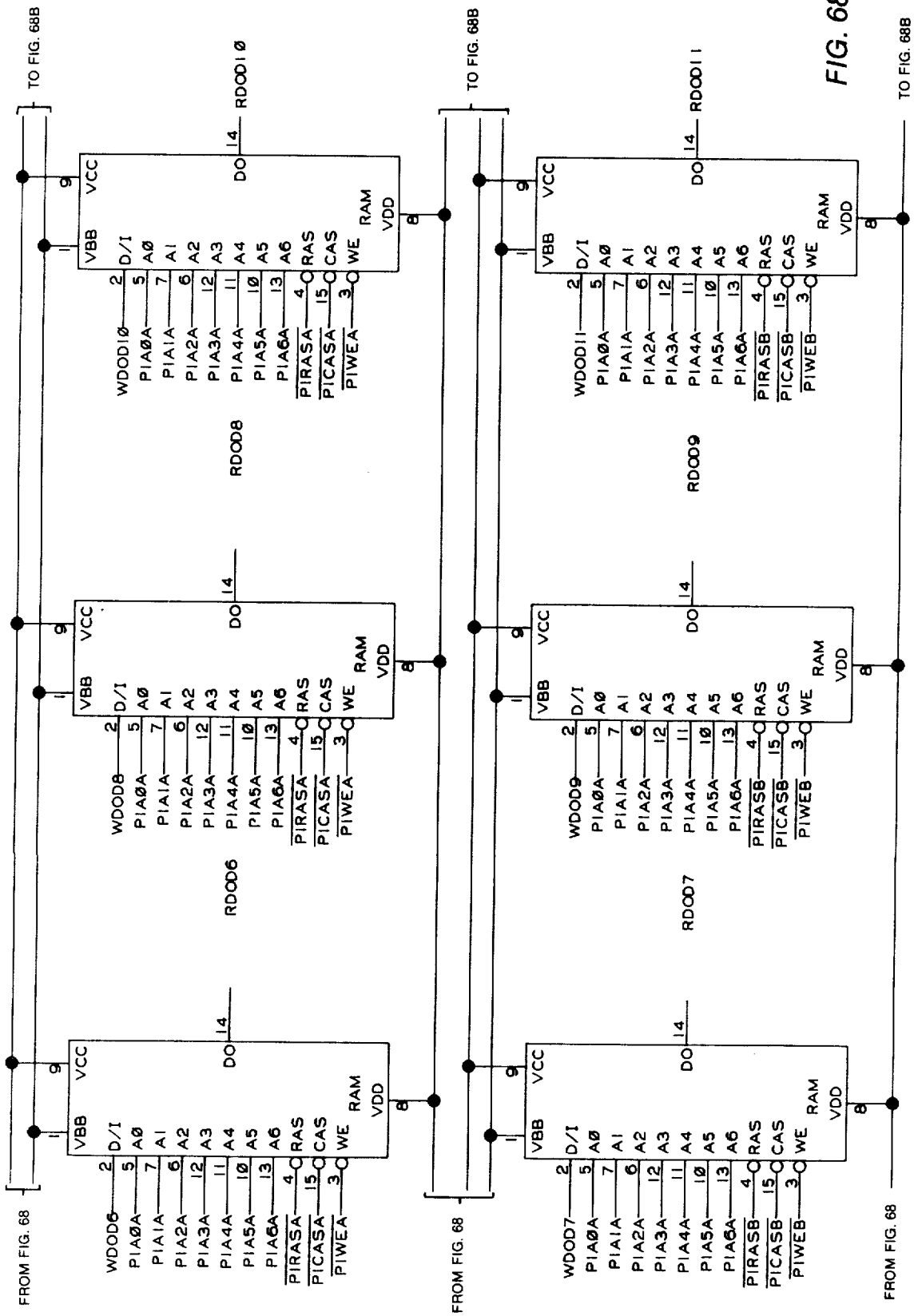

TO FIG. 68B
FROM FIG. 68
RDOD10
RDOD8
RDOD6
RDOD11
RDOD9
RDOD7
VCC
VBB
DO
RAM
VDD
D/I
A0
A1
A2
A3
A4
A5
A6
RAS
CAS
WE
WDOD10
PIA0A
PIA1A
PIA2A
PIA3A
PIA4A
PIA5A
PIA6A
PIRASA
PICASA
PIWEA
WDOD8
WDOD6
PIRASB
PICASB
PIWEB
WDOD11
WDOD9
WDOD7

FIG. 68A

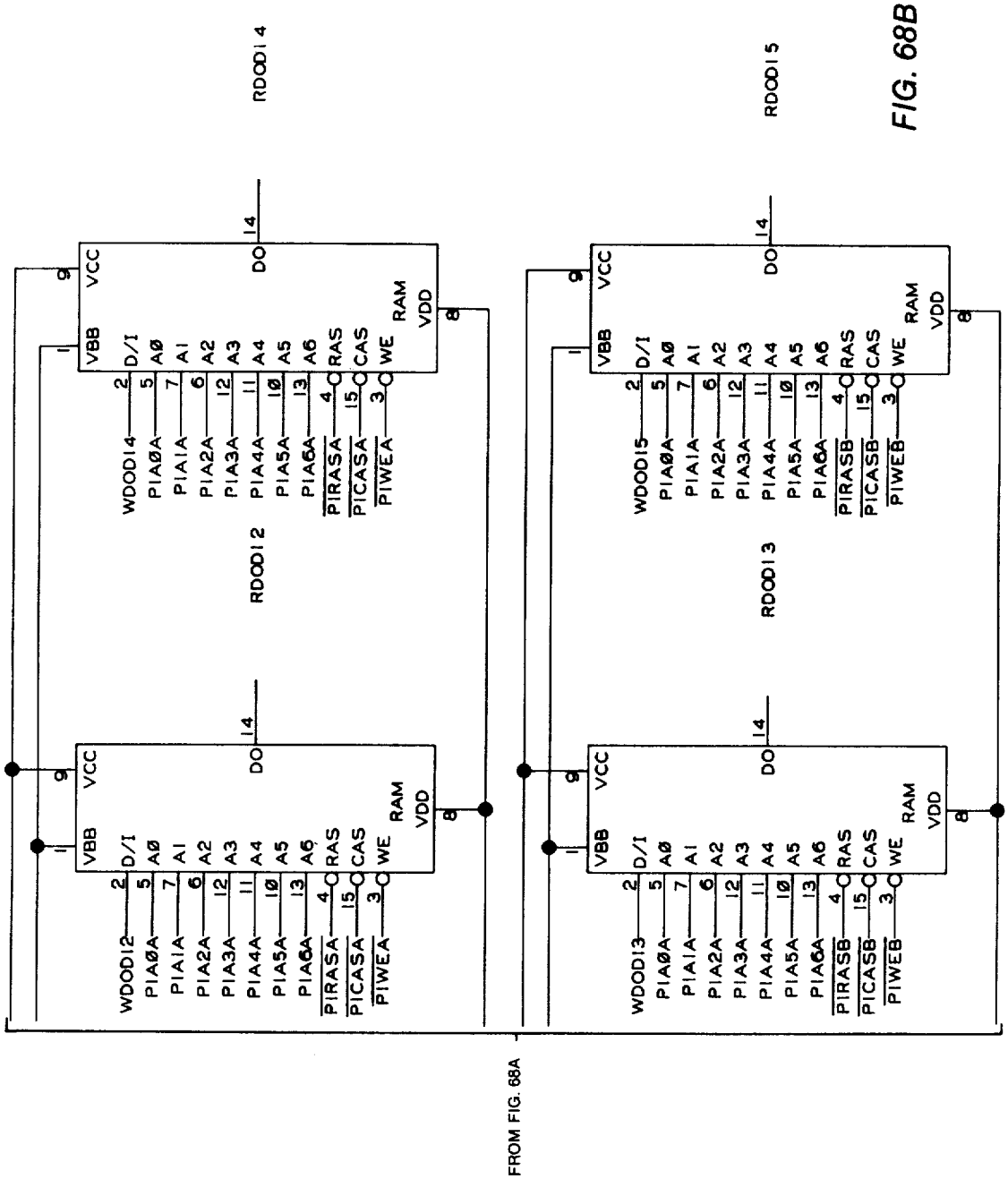

RDOD14
RDOD12
RDOD15
RDOD13
VCC
VBB
DO
RAM
VDD
D/I
A0
A1
A2
A3
A4
A5
A6
RAS
CAS
WE
WDOD14
PIA0A
PIA1A
PIA2A
PIA3A
PIA4A
PIA5A
PIA6A
PIRASA
PICASA
PIWEA
WDOD12
WDOD15
PIRASB
PICASB
PIWEB
WDOD13
FROM FIG. 68A

FIG. 68B

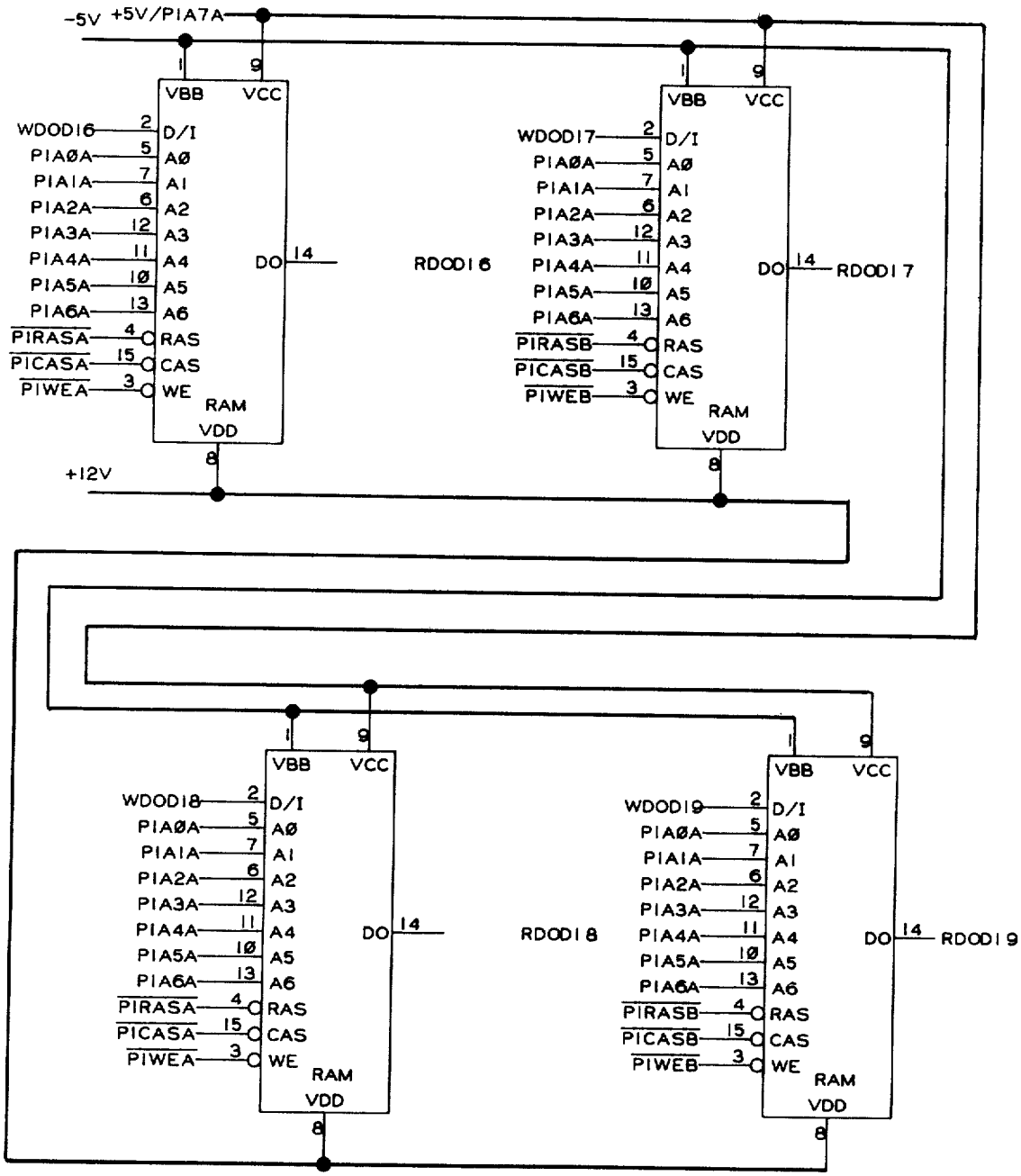

-5V
+5V/PIA7A
VBB
VCC
WDOD16
D/I
PIAØA
AØ
PIA1A
A1
PIA2A
A2
PIA3A
A3
PIA4A
A4
PIA5A
A5
PIA6A
A6
PIRASA
RAS
PICASA
CAS
PIWEA
WE
RAM
VDD
DO
RDOD16
WDOD17
PIRASB
PICASB
PIWEB
RDOD17
+12V
WDOD18
RDOD18
WDOD19
RDOD19

FIG. 68C

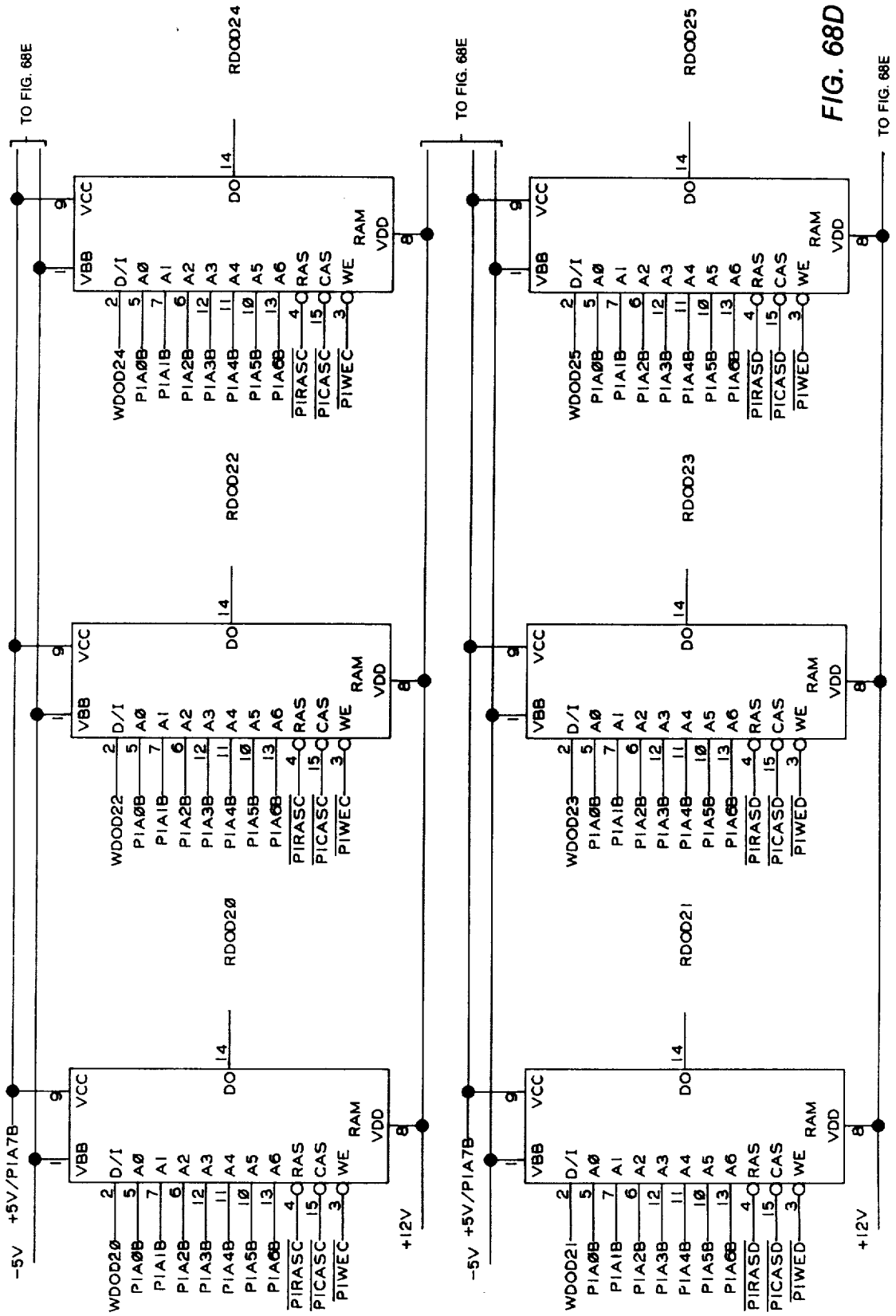

TO FIG. 68E
RDOD24
RDOD22
RDOD20
RDOD25
RDOD23
RDOD21
VCC
VBB
DO
RAM
VDD
D/I
A0
A1
A2
A3
A4
A5
A6
RAS
CAS
WE
WDOD24
WDOD22
WDOD20
WDOD25
WDOD23
WDOD21
PIA0B
PIA1B
PIA2B
PIA3B
PIA4B
PIA5B
PIA6B
PIRASC
PICASC
PIWEC
PIRASD
PICASD
PIWED
-5V
+5V/PIA7B
+12V

FIG. 68D

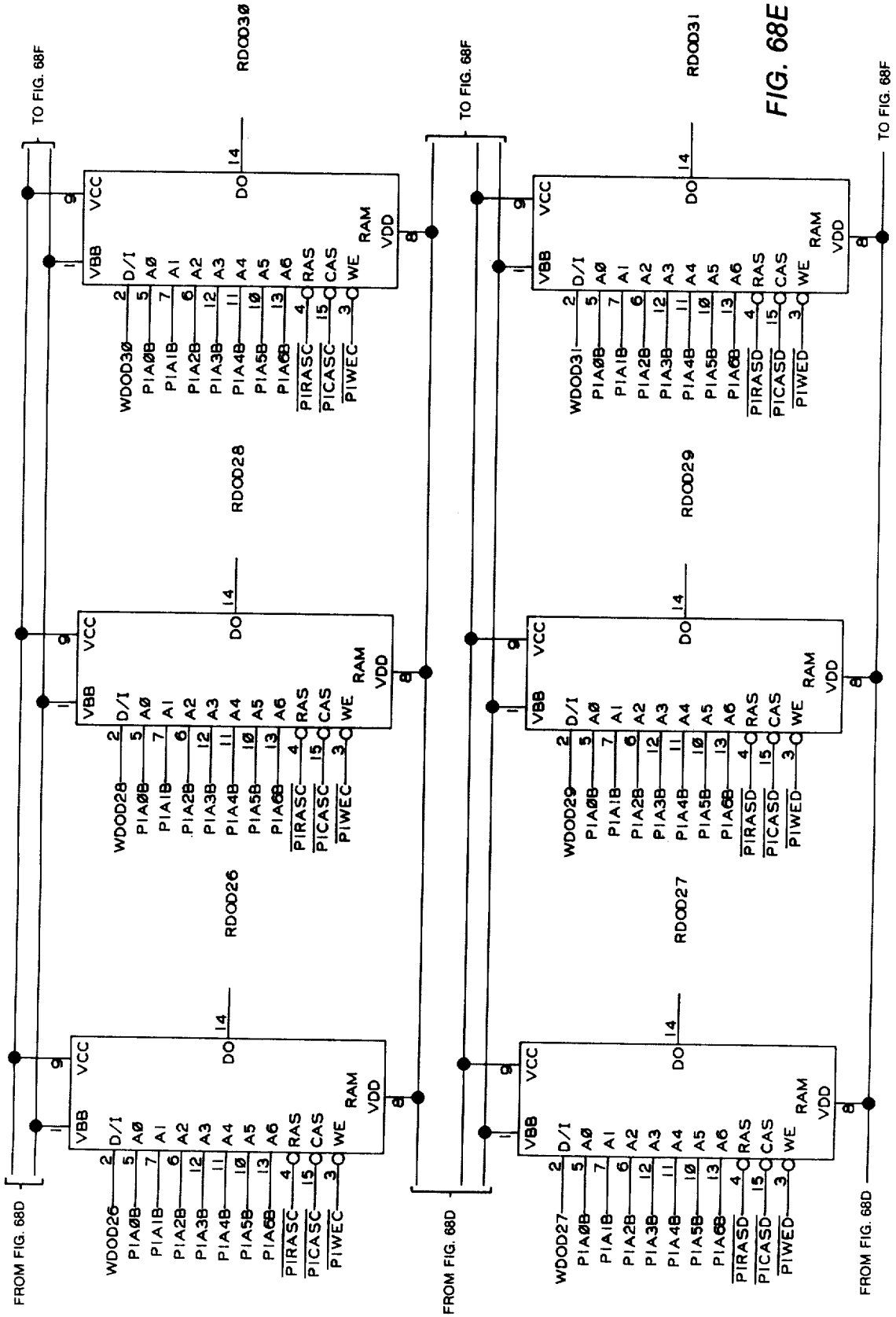


FIG. 68E
TO FIG. 68F
FROM FIG. 68D
RDOD26
RDOD27
RDOD28
RDOD29
RDOD30
RDOD31
WDOD26
WDOD27
WDOD28
WDOD29
WDOD30
WDOD31
VCC
VBB
DO
RAM
VDD
D/I
A0
A1
A2
A3
A4
A5
A6
RAS
CAS
WE
PIA0B
PIA1B
PIA2B
PIA3B
PIA4B
PIA5B
PIA6B
PIRASC
PICASC
PIWEC
PIRASD
PICASD
PIWED

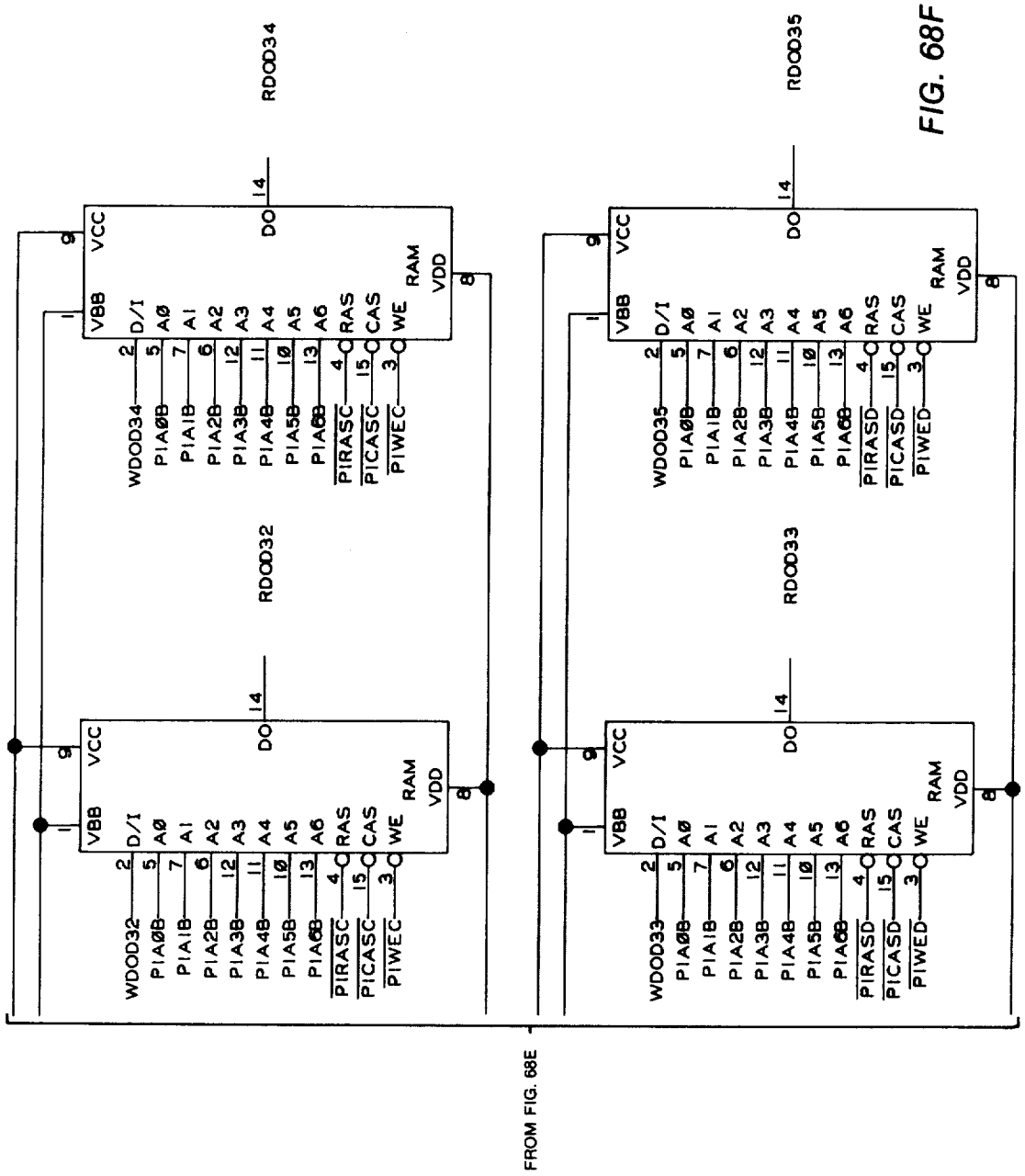

RDOD34
RDOD32
RDOD35
RDOD33
VCC
VBB
DO
RAM
VDD
D/I
A0
A1
A2
A3
A4
A5
A6
RAS
CAS
WE
WDOD34
WDOD32
WDOD35
WDOD33
PIA0B
PIA1B
PIA2B
PIA3B
PIA4B
PIA5B
PIA6B
PIRASC
PICASC
PIWEC
PIRASD
PICASD
PIWED
FROM FIG. 68E

FIG. 68F

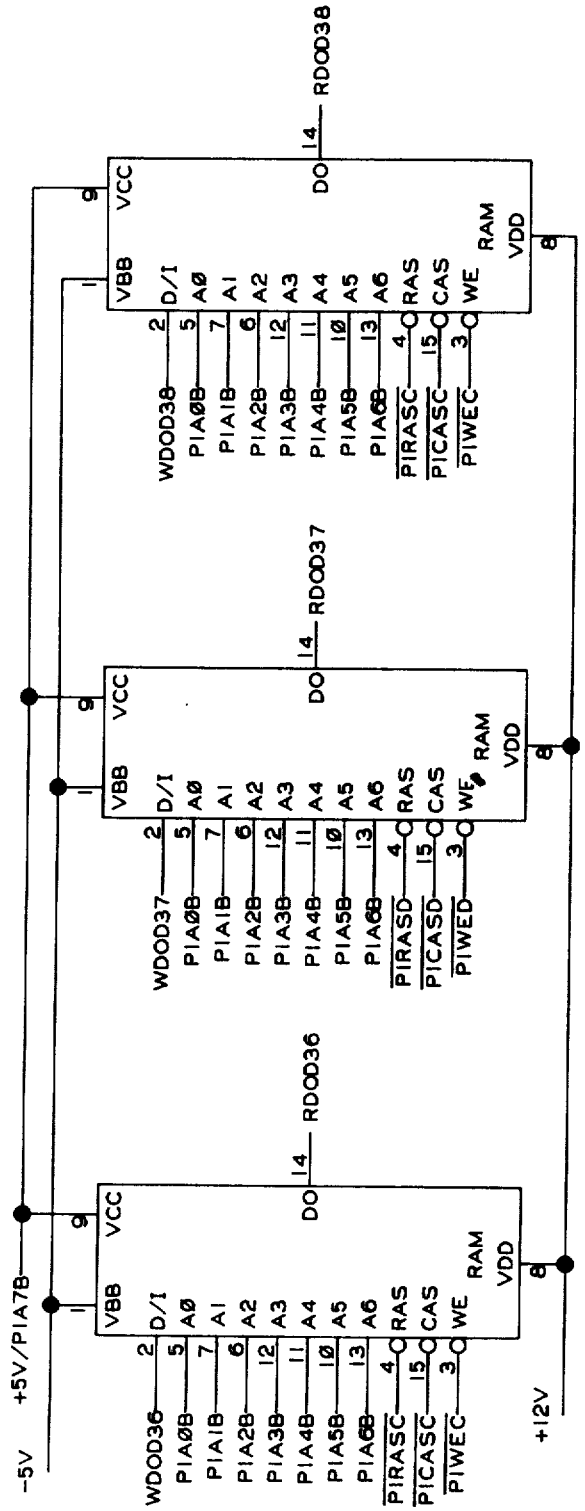
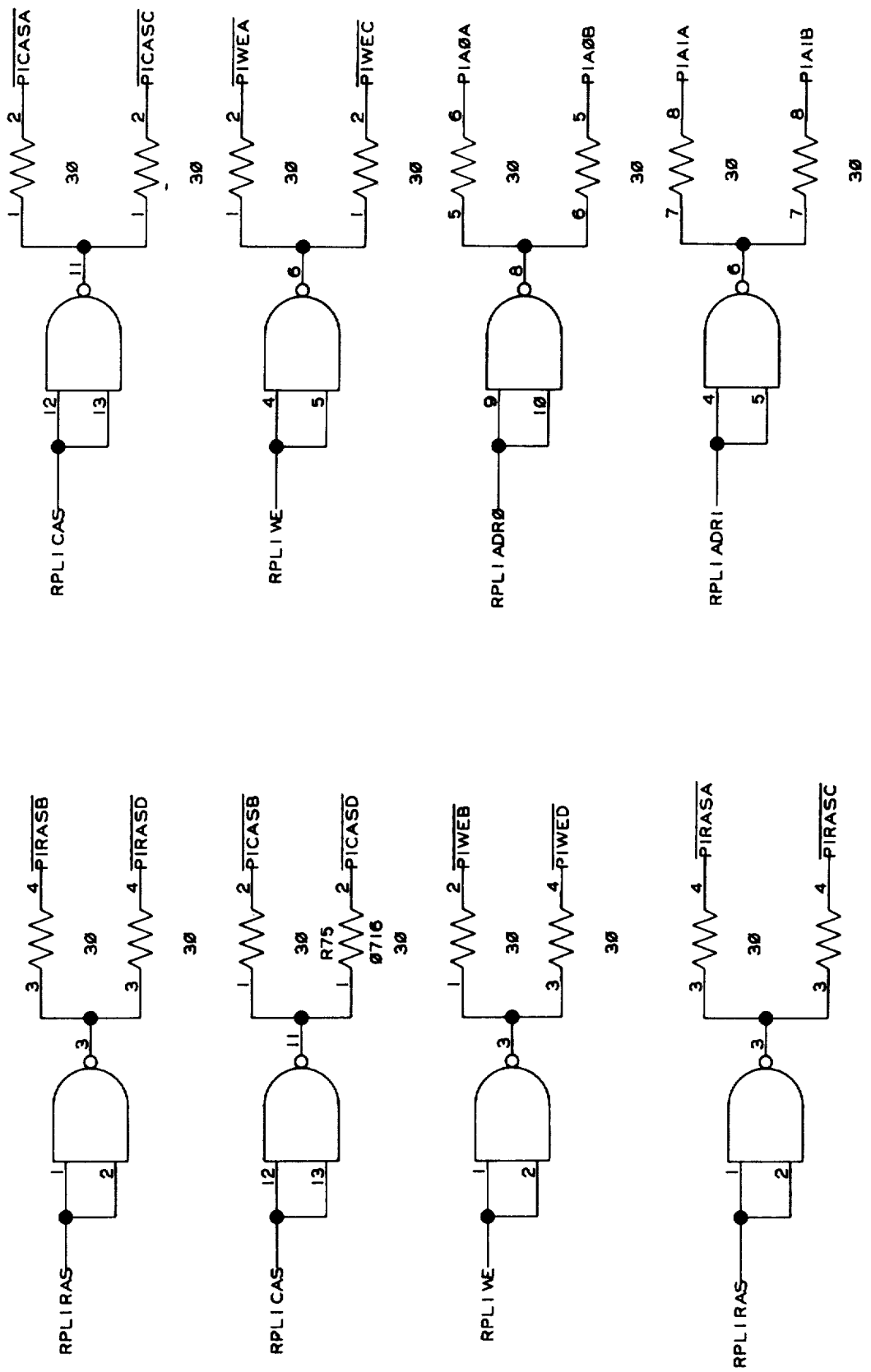

RDOD38
RDOD37
RDOD36
VCC
VBB
DO
RAM
VDD
D/I
A0
A1
A2
A3
A4
A5
A6
RAS
CAS
WE
WDOD38
PIA0B
PIA1B
PIA2B
PIA3B
PIA4B
PIA5B
PIA6B
PIRASC
PICASC
PIWEC
WDOD37
PIRASD
PICASD
PIWED
WDOD36
-5V
+5V/PIA7B
+12V

FIG. 68G



PLANE 1 CONTROL INPUTS

FIG. 69

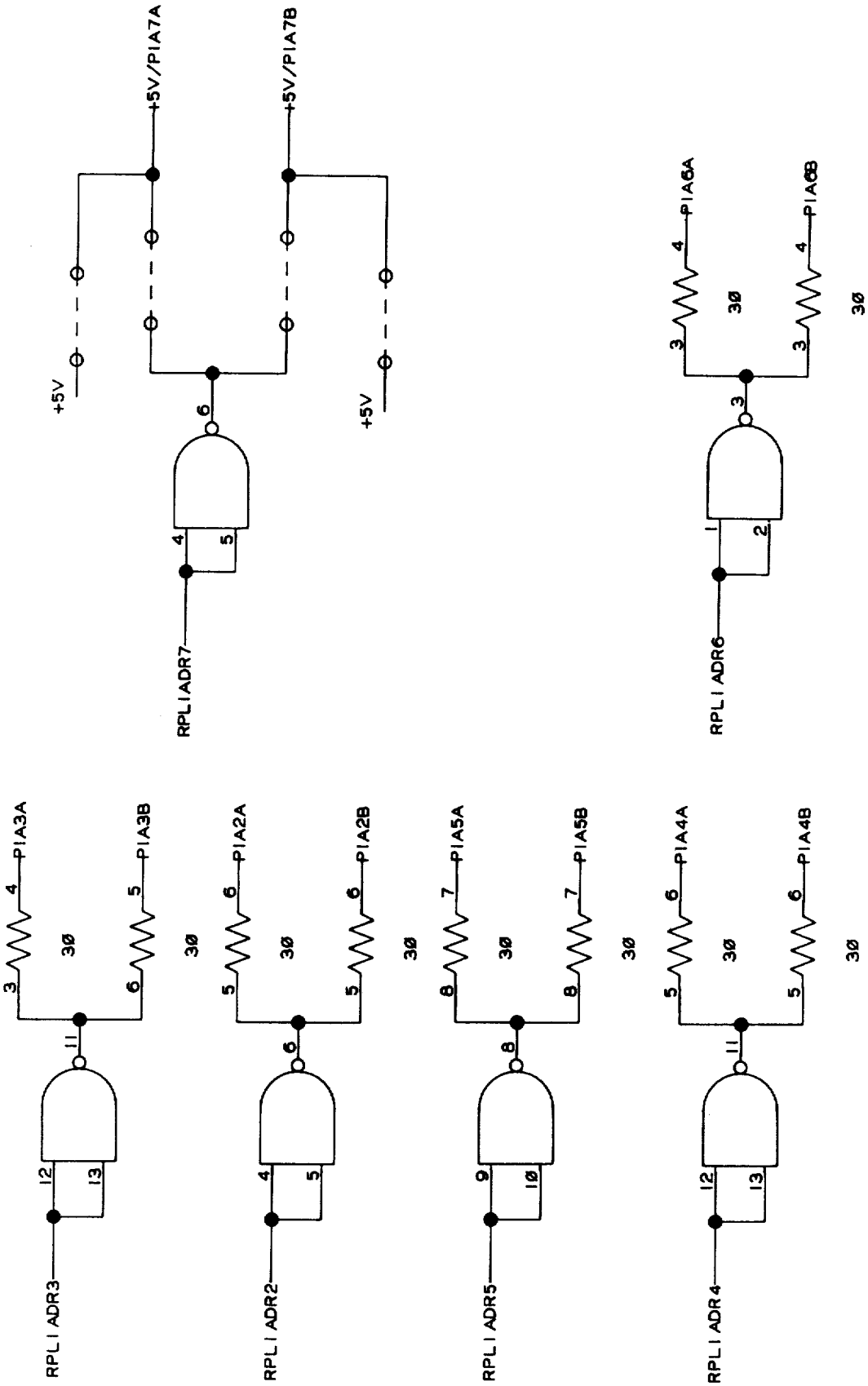

+5V
+5V/PIA7A
+5V/PIA7B
+5V
RPLI ADR7
RPLI ADR6
PIA6A
PIA6B
RPLI ADR3
PIA3A
PIA3B
RPLI ADR2
PIA2A
PIA2B
RPLI ADR5
PIA5A
PIA5B
RPLI ADR4
PIA4A
PIA4B
3Ø

FIG. 69A

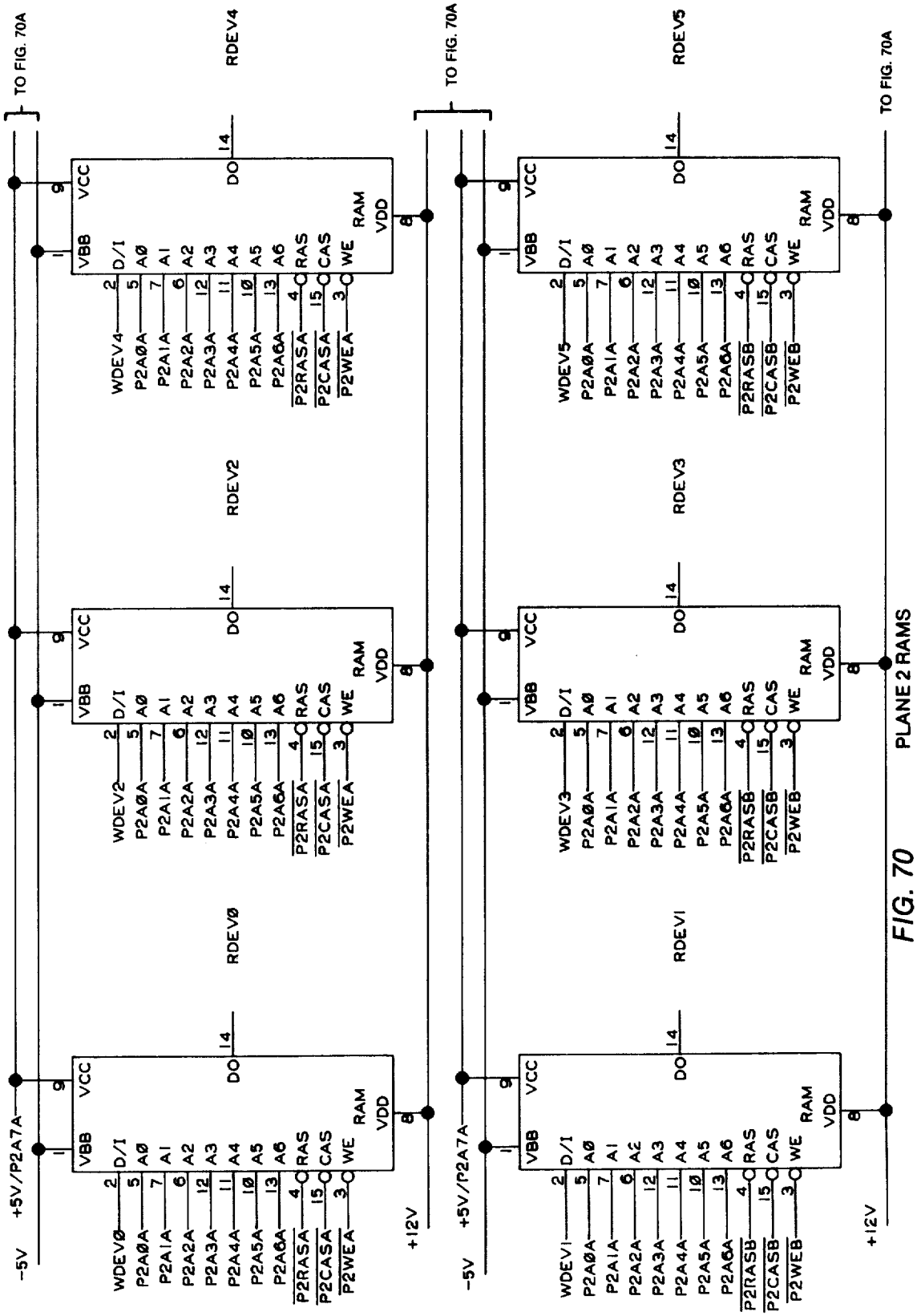

TO FIG. 70A
RDEV4
RDEV2
RDEV0
RDEV5
RDEV3
RDEV1
VBB
VCC
D/I
A0
A1
A2
A3
A4
A5
A6
RAS
CAS
WE
DO
RAM
VDD
WDEV4
P2A0A
P2A1A
P2A2A
P2A3A
P2A4A
P2A5A
P2A6A
P2RASA
P2CASA
P2WEA
WDEV2
WDEV0
WDEV5
WDEV3
WDEV1
P2RASB
P2CASB
P2WEB
-5V
+5V/P2A7A
+12V
PLANE 2 RAMS

FIG. 70

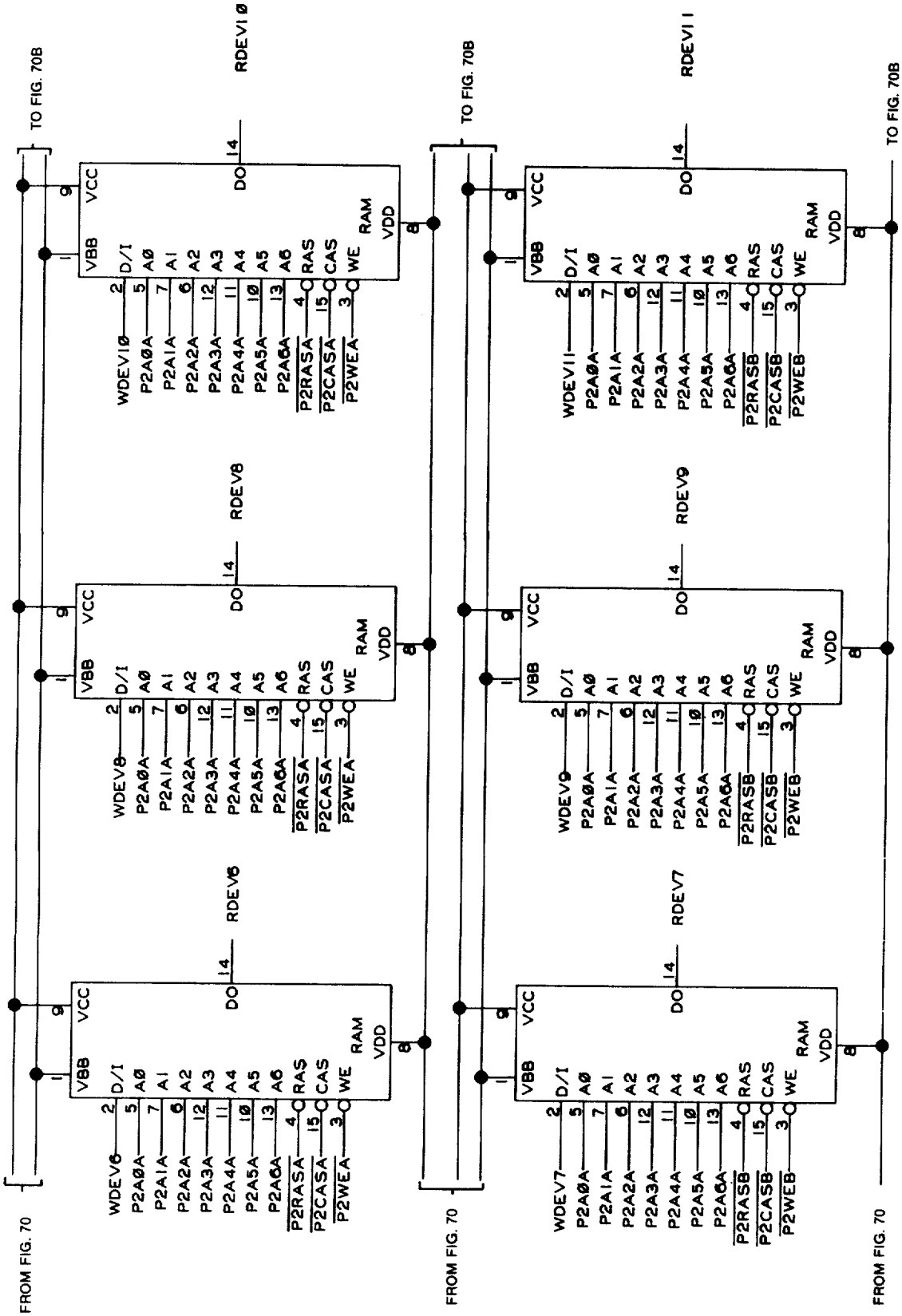

TO FIG. 70B
FROM FIG. 70
RDEV10
RDEV8
RDEV6
RDEV11
RDEV9
RDEV7
VCC
VBB
DO
RAM
VDD
D/I
A0
A1
A2
A3
A4
A5
A6
RAS
CAS
WE
WDEV10
WDEV8
WDEV6
WDEV11
WDEV9
WDEV7
P2A0A
P2A1A
P2A2A
P2A3A
P2A4A
P2A5A
P2A6A
P2RASA
P2CASA
P2WEA
P2RASB
P2CASB
P2WEB

FIG. 70A

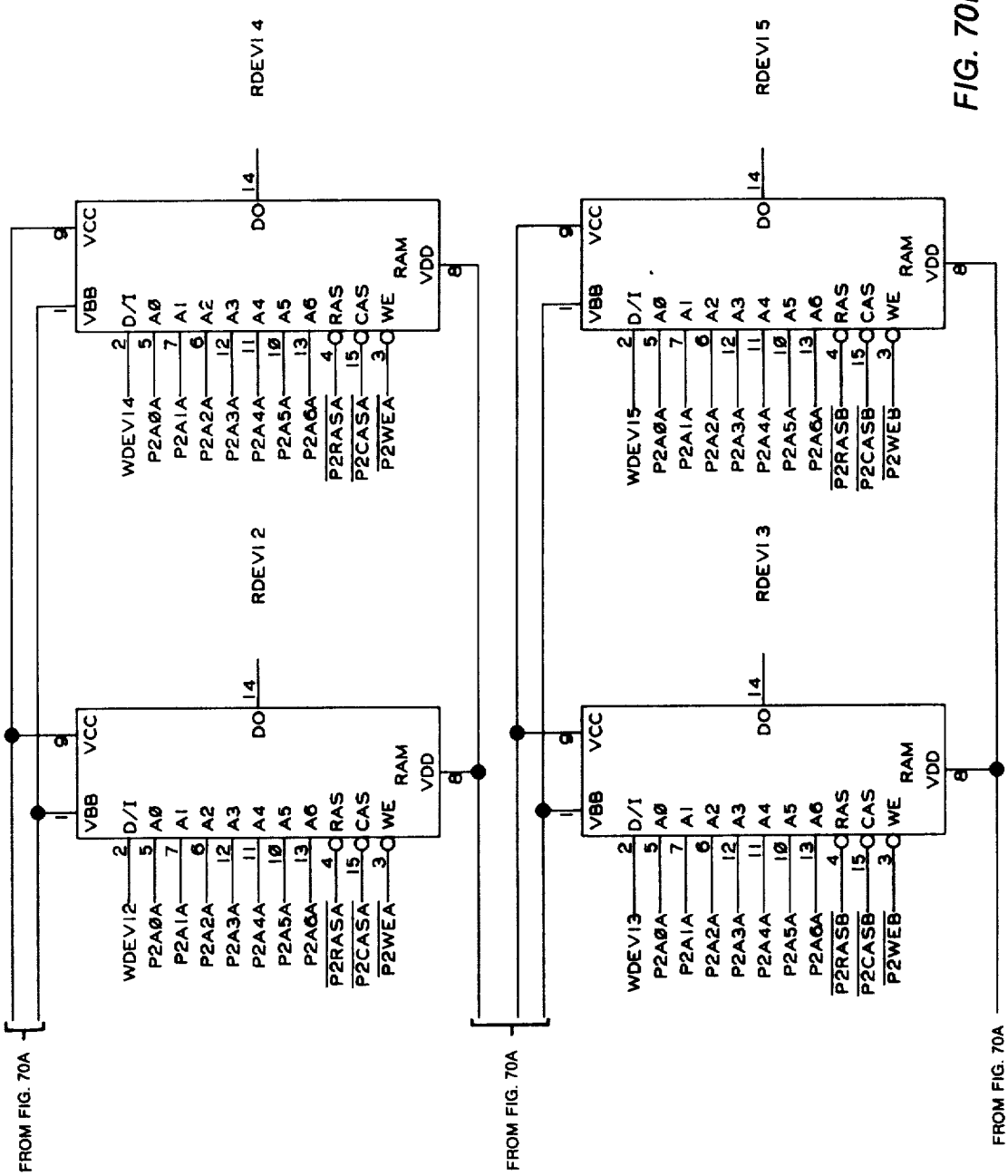

FROM FIG. 70A
RDEVI2
RDEVI4
RDEVI3
RDEVI5
VCC
VBB
DO
RAM
VDD
D/I
A0
A1
A2
A3
A4
A5
A6
RAS
CAS
WE
WDEVI2
WDEVI4
WDEVI3
WDEVI5
P2A0A
P2A1A
P2A2A
P2A3A
P2A4A
P2A5A
P2A6A
P2RASA
P2CASA
P2WEA
P2RASB
P2CASB
P2WEB
FROM FIG. 70A
FROM FIG. 70A

FIG. 70B

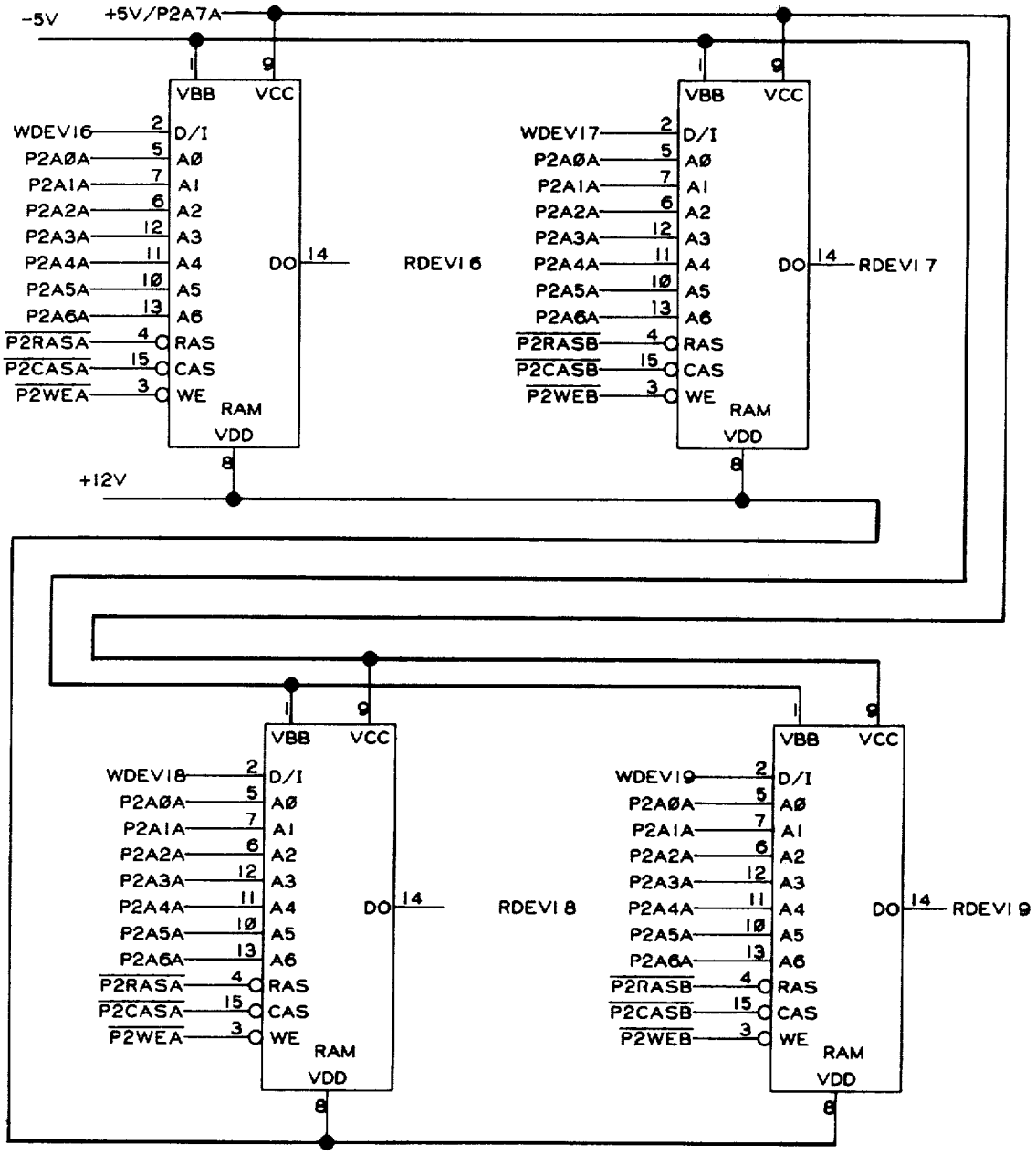

-5V
+5V/P2A7A
VBB
VCC
WDEVI6
D/I
P2AØA
AØ
P2A1A
A1
P2A2A
A2
P2A3A
A3
P2A4A
A4
P2A5A
A5
P2A6A
A6
P2RASA
RAS
P2CASA
CAS
P2WEA
WE
DO
RDEVI 6
RAM
VDD
+12V
WDEVI7
P2RASB
P2CASB
P2WEB
RDEVI 7
WDEVI8
RDEVI 8
WDEVI9
RDEVI 9

FIG. 70C

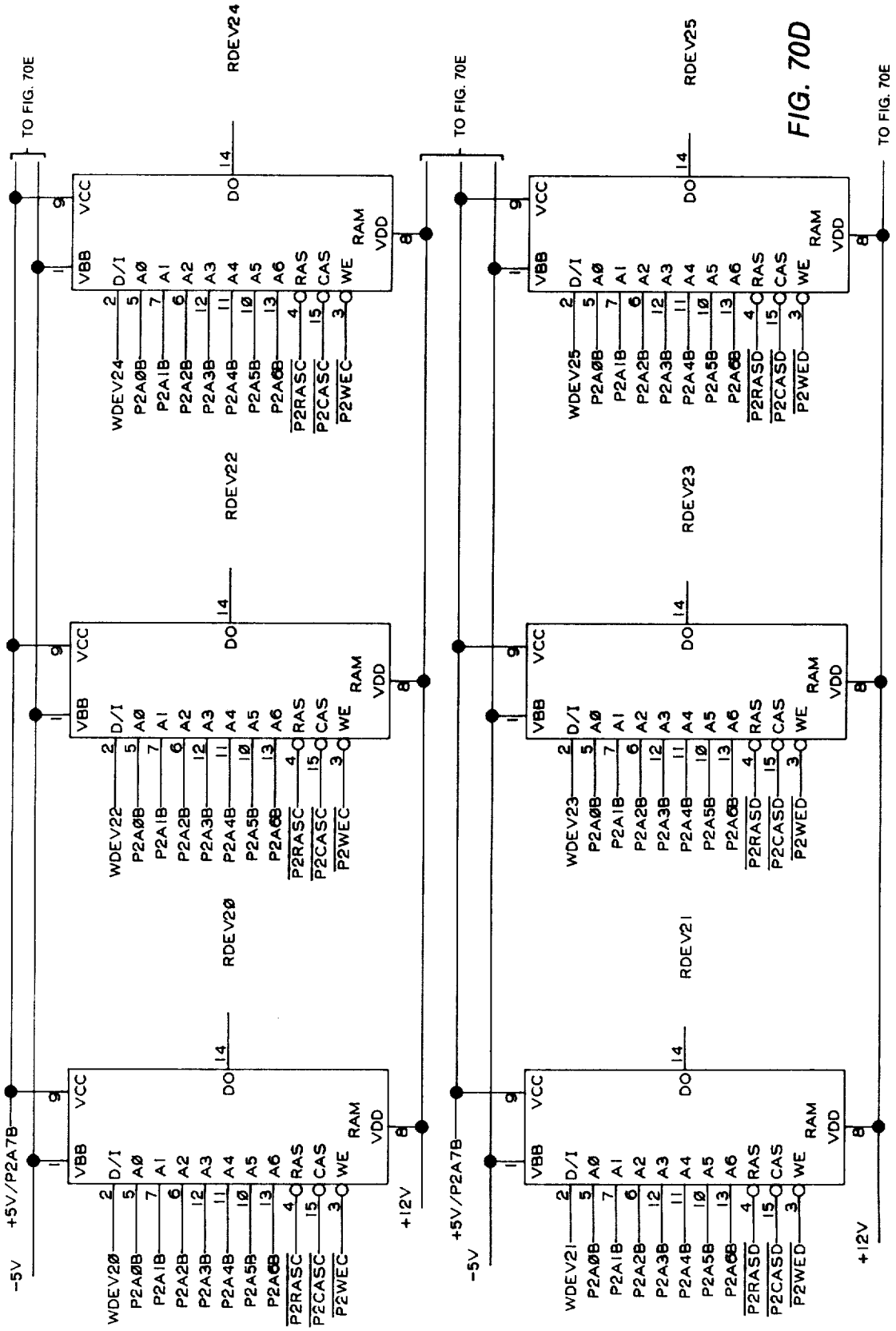

TO FIG. 70E
RDEV24
RDEV22
RDEV20
RDEV25
RDEV23
RDEV21
VCC
VBB
DO
RAM
VDD
D/I
A0
A1
A2
A3
A4
A5
A6
RAS
CAS
WE
WDEV24
WDEV22
WDEV20
WDEV25
WDEV23
WDEV21
P2A0B
P2A1B
P2A2B
P2A3B
P2A4B
P2A5B
P2A6B
P2RASC
P2CASC
P2WEC
P2RASD
P2CASD
P2WED
+5V/P2A7B
-5V
+12V

FIG. 70D

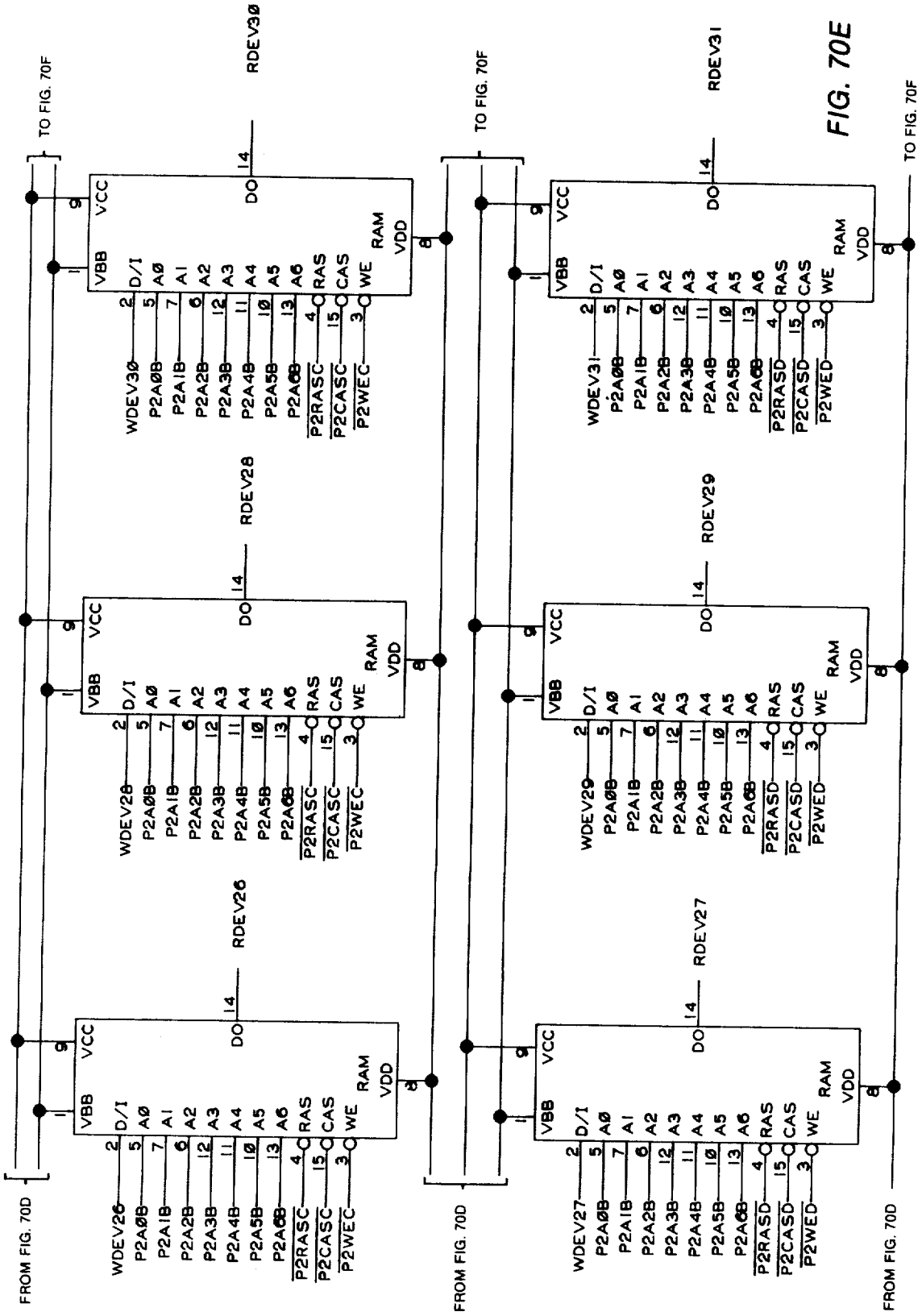

FROM FIG. 70D
TO FIG. 70F
RDEV26
RDEV27
RDEV28
RDEV29
RDEV30
RDEV31
VCC
VBB
DO
RAM
VDD
D/I
A0
A1
A2
A3
A4
A5
A6
RAS
CAS
WE
WDEV26
WDEV27
WDEV28
WDEV29
WDEV30
WDEV31
P2A0B
P2A1B
P2A2B
P2A3B
P2A4B
P2A5B
P2A6B
P2RASC
P2CASC
P2WEC
P2RASD
P2CASD
P2WED

FIG. 70E

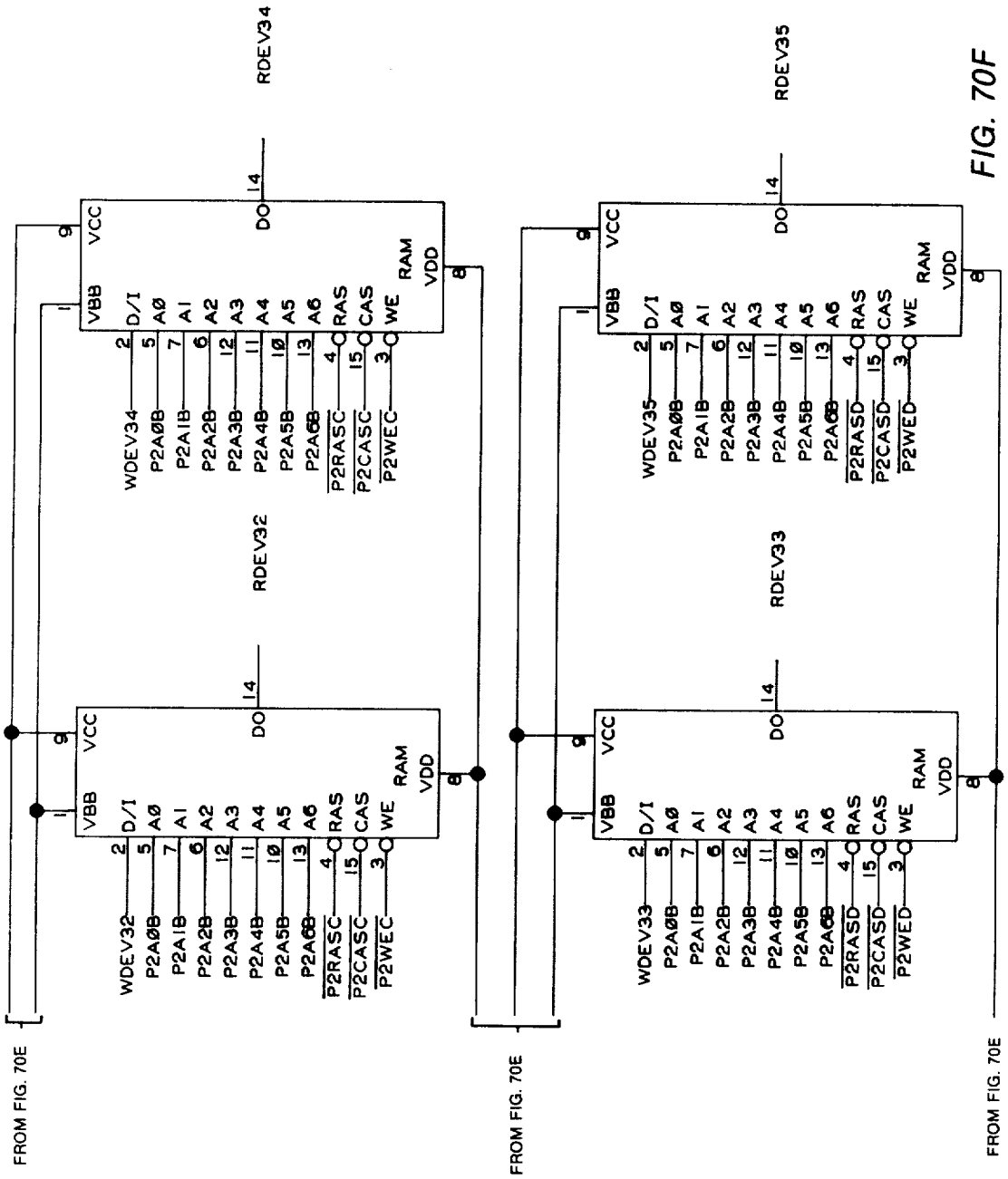

FROM FIG. 70E
RDEV32
RDEV34
RDEV33
RDEV35
VCC
VBB
DO
RAM
VDD
D/I
A0
A1
A2
A3
A4
A5
A6
RAS
CAS
WE
WDEV32
WDEV34
WDEV33
WDEV35
P2A0B
P2A1B
P2A2B
P2A3B
P2A4B
P2A5B
P2A6B
P2RASC
P2CASC
P2WEC
P2RASD
P2CASD
P2WED

FIG. 70F

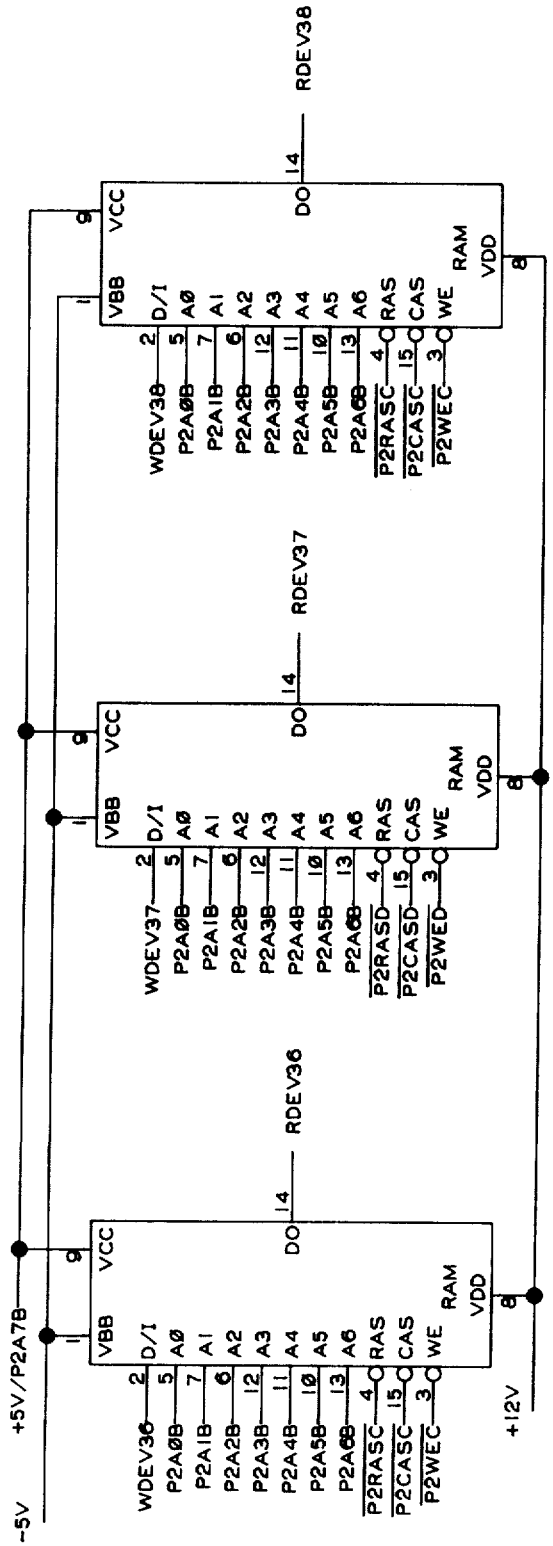
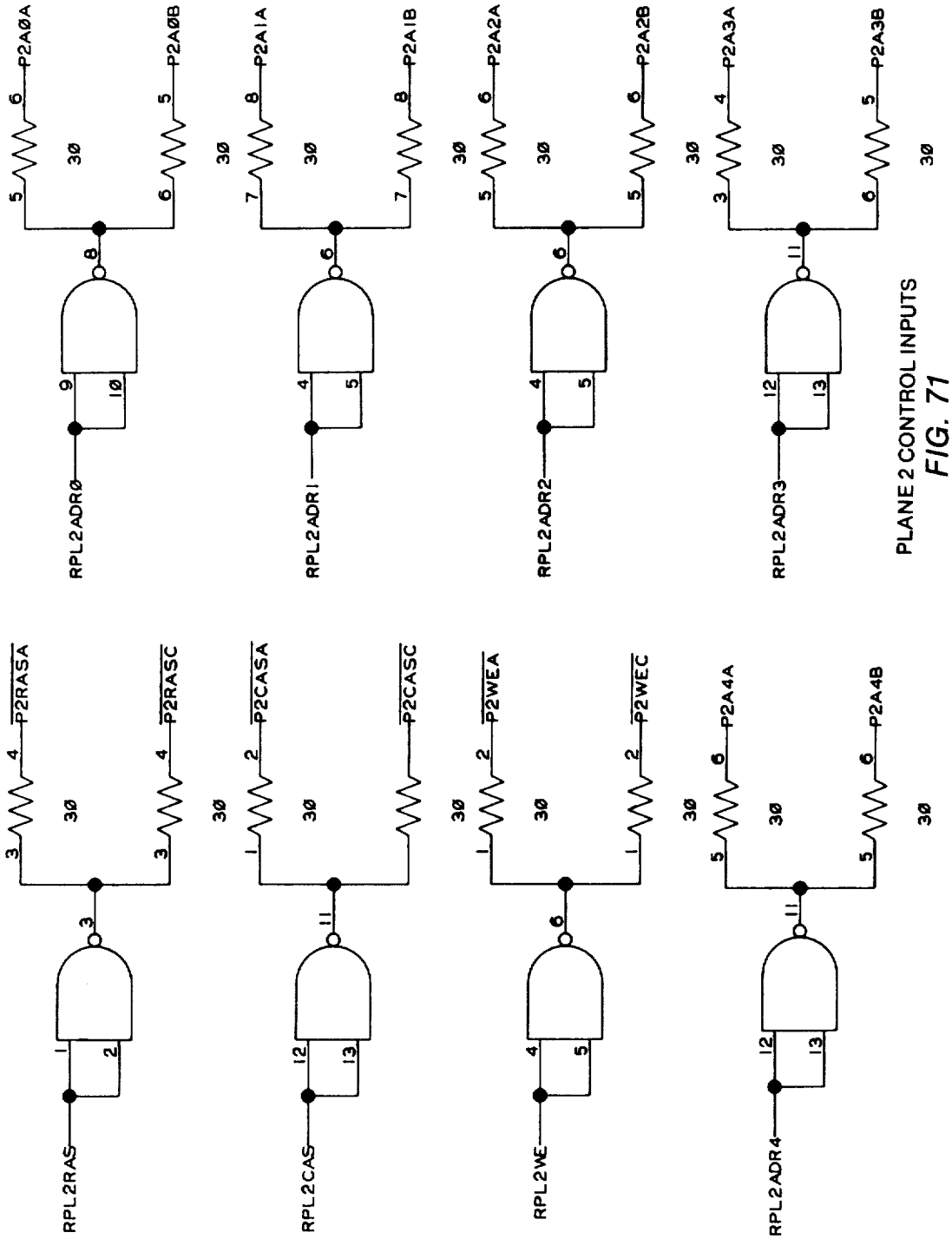

RDEV38
RDEV37
RDEV36
VCC
VBB
DO
RAM
VDD
D/I
A0
A1
A2
A3
A4
A5
A6
RAS
CAS
WE
WDEV38
P2A0B
P2A1B
P2A2B
P2A3B
P2A4B
P2A5B
P2A6B
P2RASC
P2CASC
P2WEC
WDEV37
P2RASD
P2CASD
P2WED
WDEV36
P2WEC
+5V/P2A7B
-5V
+12V

FIG. 70G



PLANE 2 CONTROL INPUTS
FIG. 71

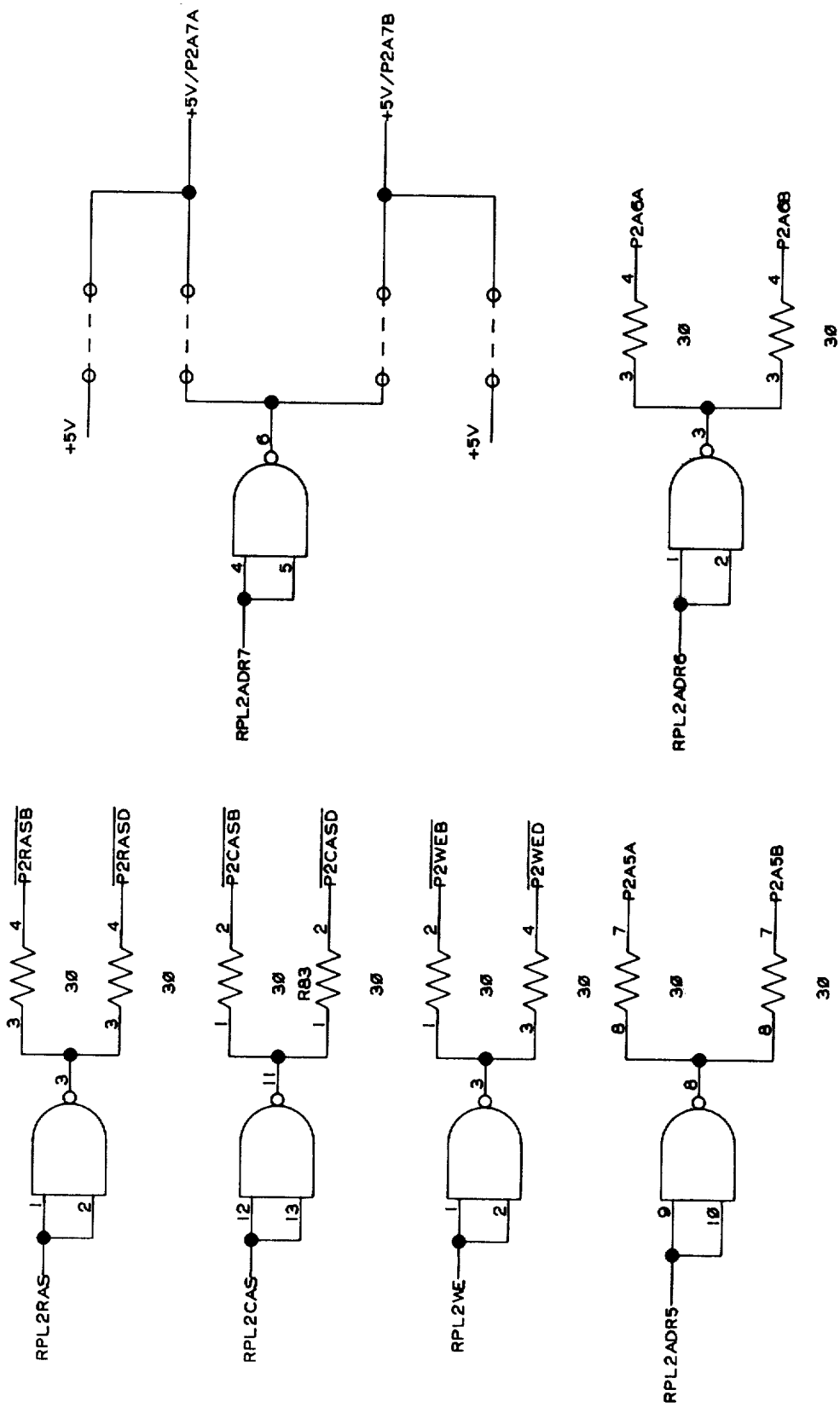
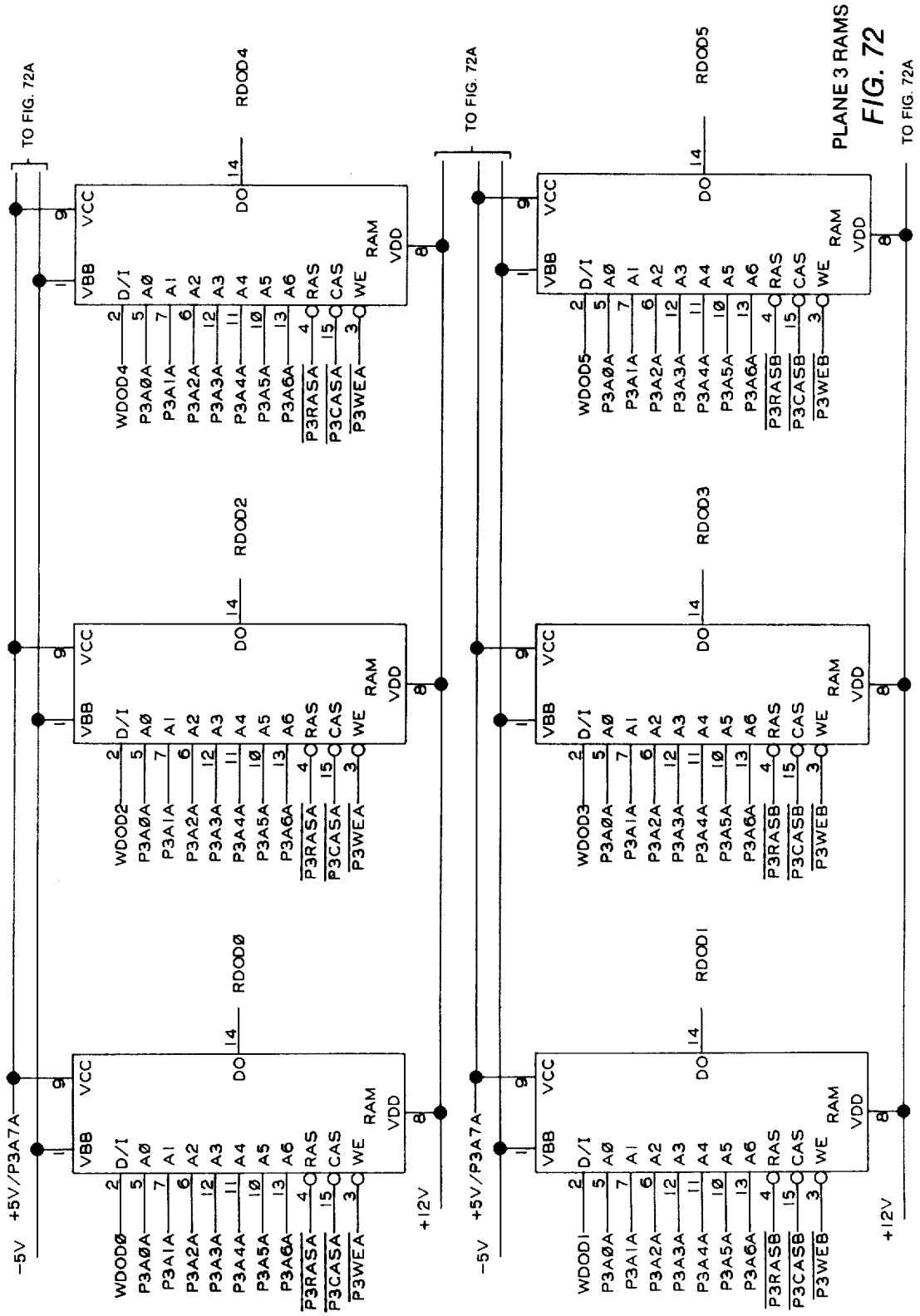

+5V
+5V/P2A7A
+5V/P2A7B
+5V
RPL2ADR7
RPL2ADR6
P2A6A
P2A6B
RPL2RAS
P2RASB
P2RASD
RPL2CAS
P2CASB
P2CASD
R83
RPL2WE
P2WEB
P2WED
RPL2ADR5
P2A5A
P2A5B
30

FIG. 71A



PLANE 3 RAMS
FIG. 72

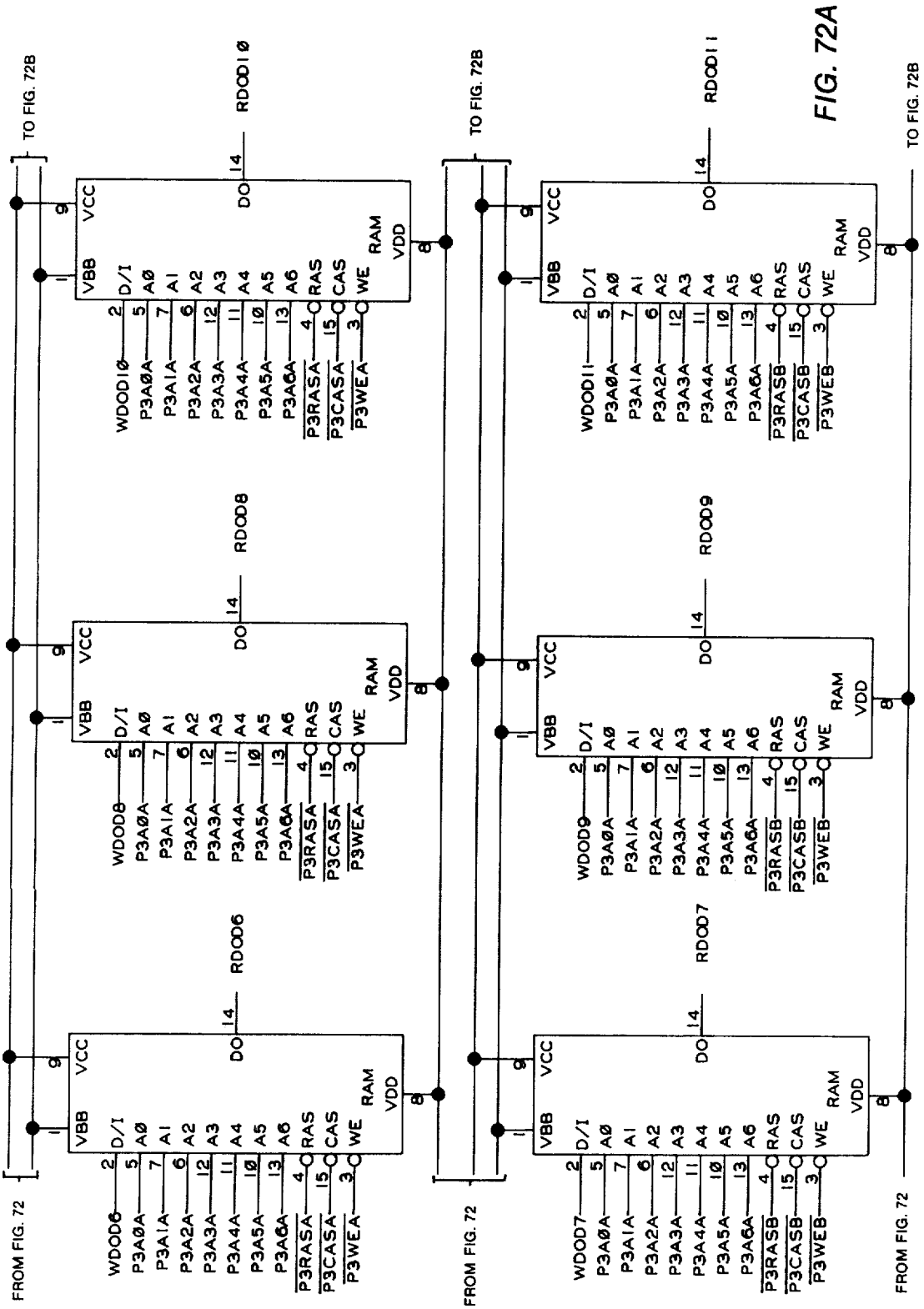


FIG. 72A
TO FIG. 72B
FROM FIG. 72
RDOD10
RDOD8
RDOD6
RDOD11
RDOD9
RDOD7
VCC
VBB
DO
RAM
VDD
D/I
A0
A1
A2
A3
A4
A5
A6
RAS
CAS
WE
WDOD10
P3A0A
P3A1A
P3A2A
P3A3A
P3A4A
P3A5A
P3A6A
P3RASA
P3CASA
P3WEA
WDOD8
WDOD6
WDOD11
P3RASB
P3CASB
P3WEB
WDOD9
WDOD7

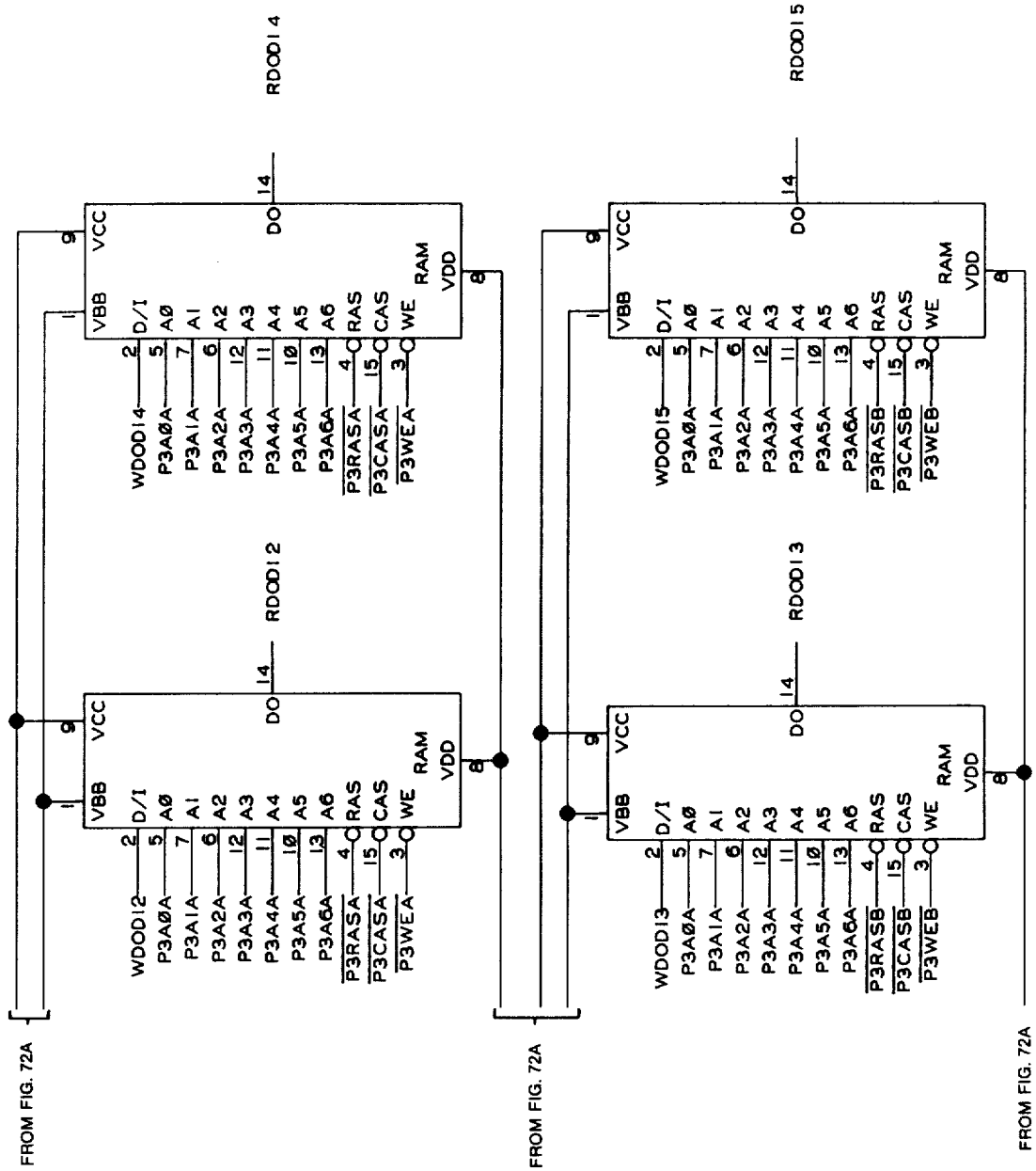

FROM FIG. 72A
VCC
VBB
D/I
A0
A1
A2
A3
A4
A5
A6
RAS
CAS
WE
DO
RAM
VDD
WDOD12
P3A0A
P3A1A
P3A2A
P3A3A
P3A4A
P3A5A
P3A6A
P3RASA
P3CASA
P3WEA
RDOD12
WDOD13
P3RASB
P3CASB
P3WEB
RDOD13
WDOD14
RDOD14
WDOD15
RDOD15
FROM FIG. 72A
FROM FIG. 72A

FIG. 72B

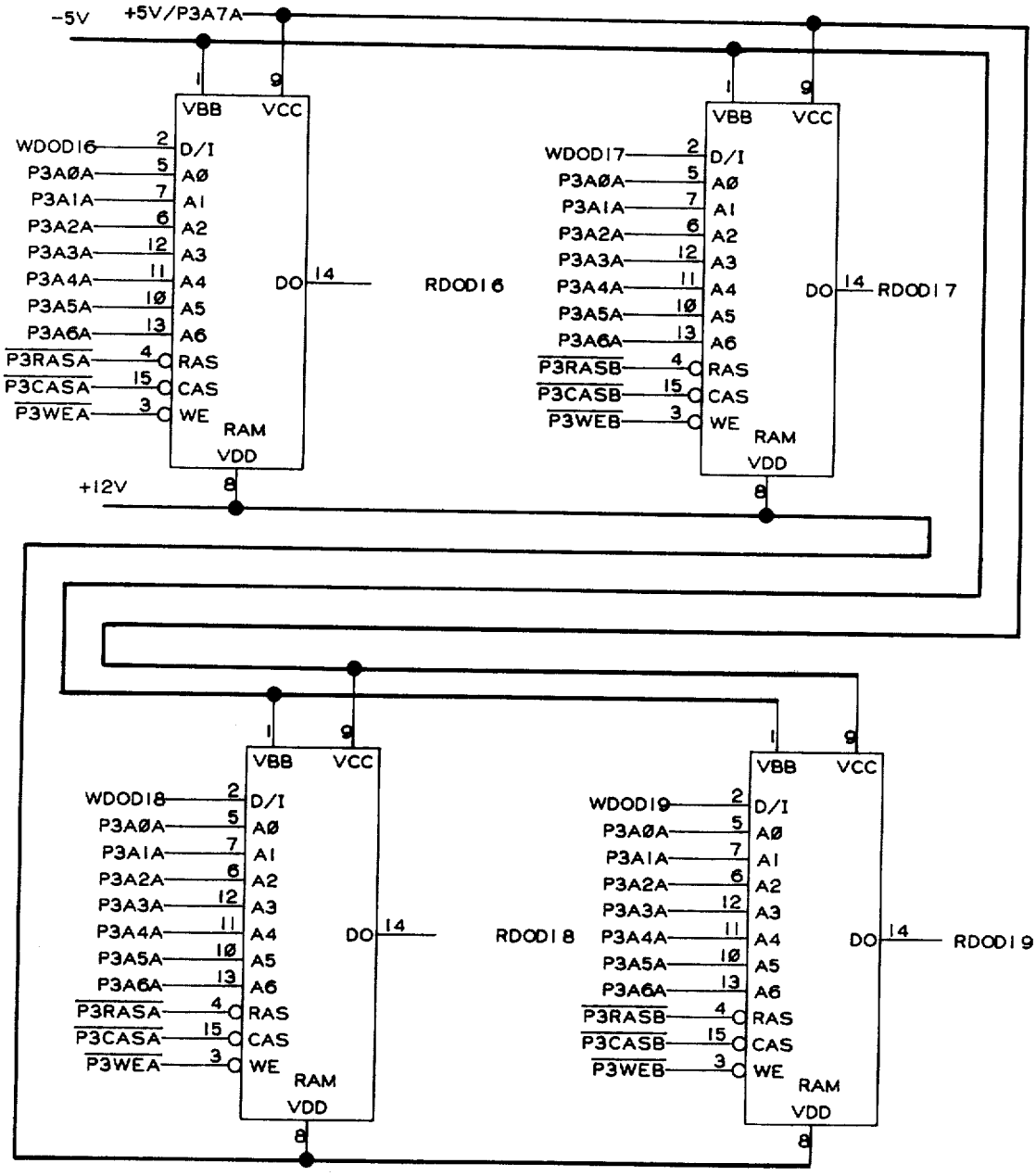

-5V
+5V/P3A7A
VBB
VCC
WDOD16
D/I
P3A0A
A0
P3A1A
A1
P3A2A
A2
P3A3A
A3
P3A4A
A4
P3A5A
A5
P3A6A
A6
P3RASA
RAS
P3CASA
CAS
P3WEA
WE
RAM
VDD
DO
RDOD16
WDOD17
P3RASB
P3CASB
P3WEB
RDOD17
+12V
WDOD18
RDOD18
WDOD19
RDOD19

FIG. 72C

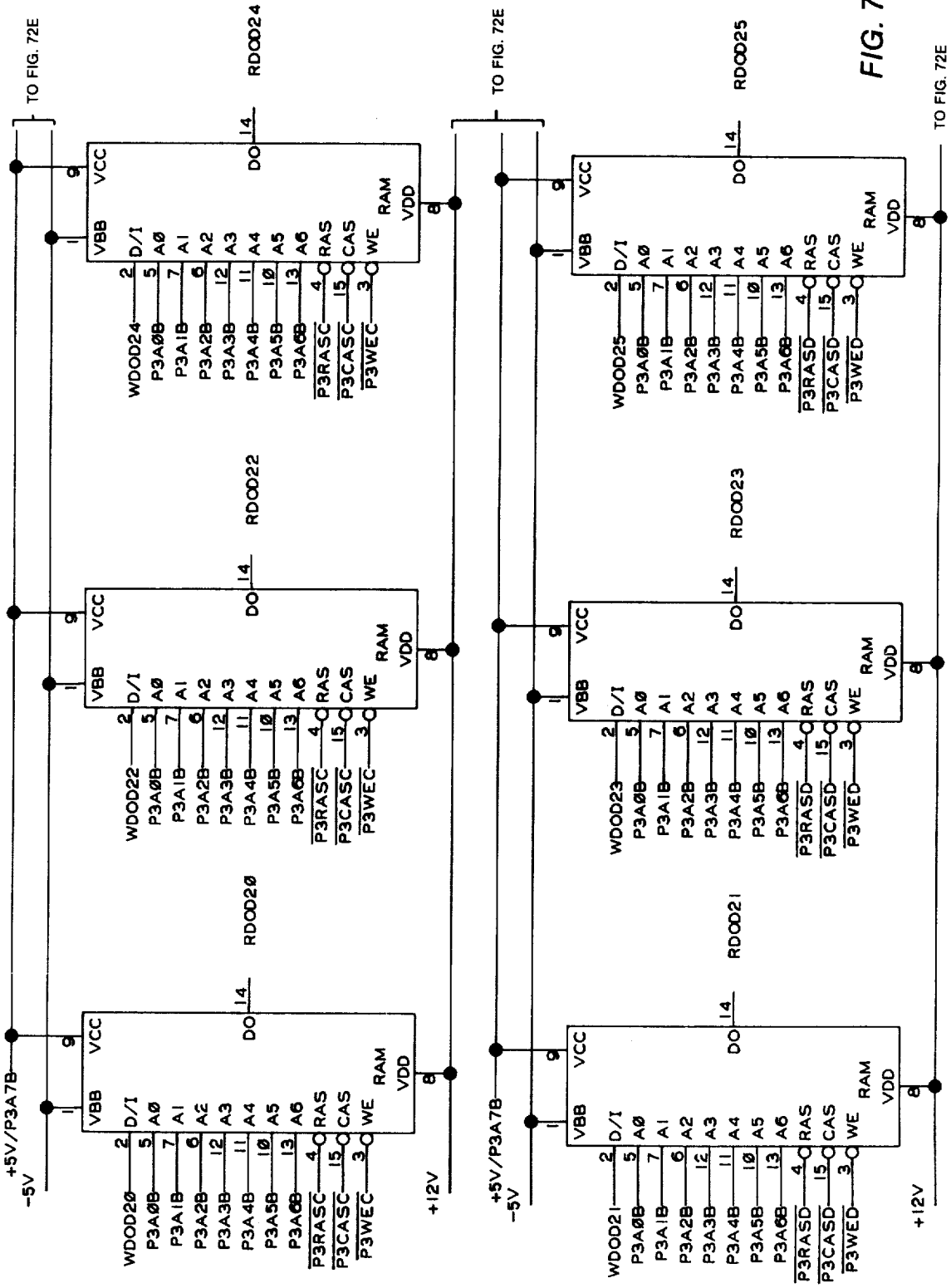

TO FIG. 72E
RDOD24
RDOD22
RDOD20
RDOD25
RDOD23
RDOD21
VCC
VBB
DO
RAM
VDD
D/I
A0
A1
A2
A3
A4
A5
A6
RAS
CAS
WE
WDOD24
P3A0B
P3A1B
P3A2B
P3A3B
P3A4B
P3A5B
P3A6B
P3RASC
P3CASC
P3WEC
WDOD22
WDOD20
WDOD25
WDOD23
WDOD21
P3RASD
P3CASD
P3WED
+5V/P3A7B
-5V
+12V

FIG. 72D

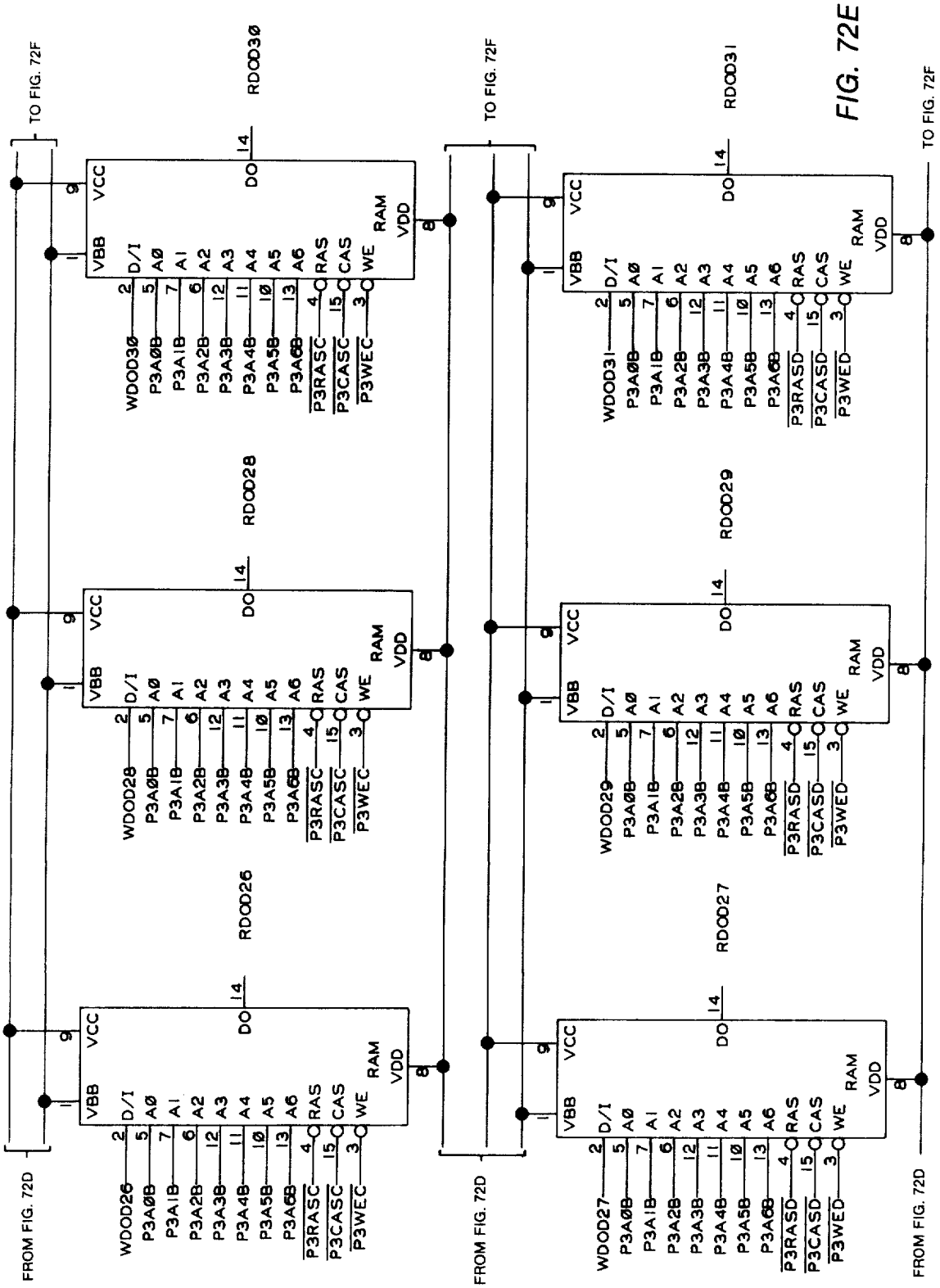

TO FIG. 72F
FROM FIG. 72D
RDOD26
RDOD28
RDOD30
RDOD27
RDOD29
RDOD31
VCC
VBB
DO
RAM
VDD
D/I
A0
A1
A2
A3
A4
A5
A6
RAS
CAS
WE
WDOD26
WDOD28
WDOD30
WDOD27
WDOD29
WDOD31
P3A0B
P3A1B
P3A2B
P3A3B
P3A4B
P3A5B
P3A6B
P3RASC
P3CASC
P3WEC
P3RASD
P3CASD
P3WED

FIG. 72E

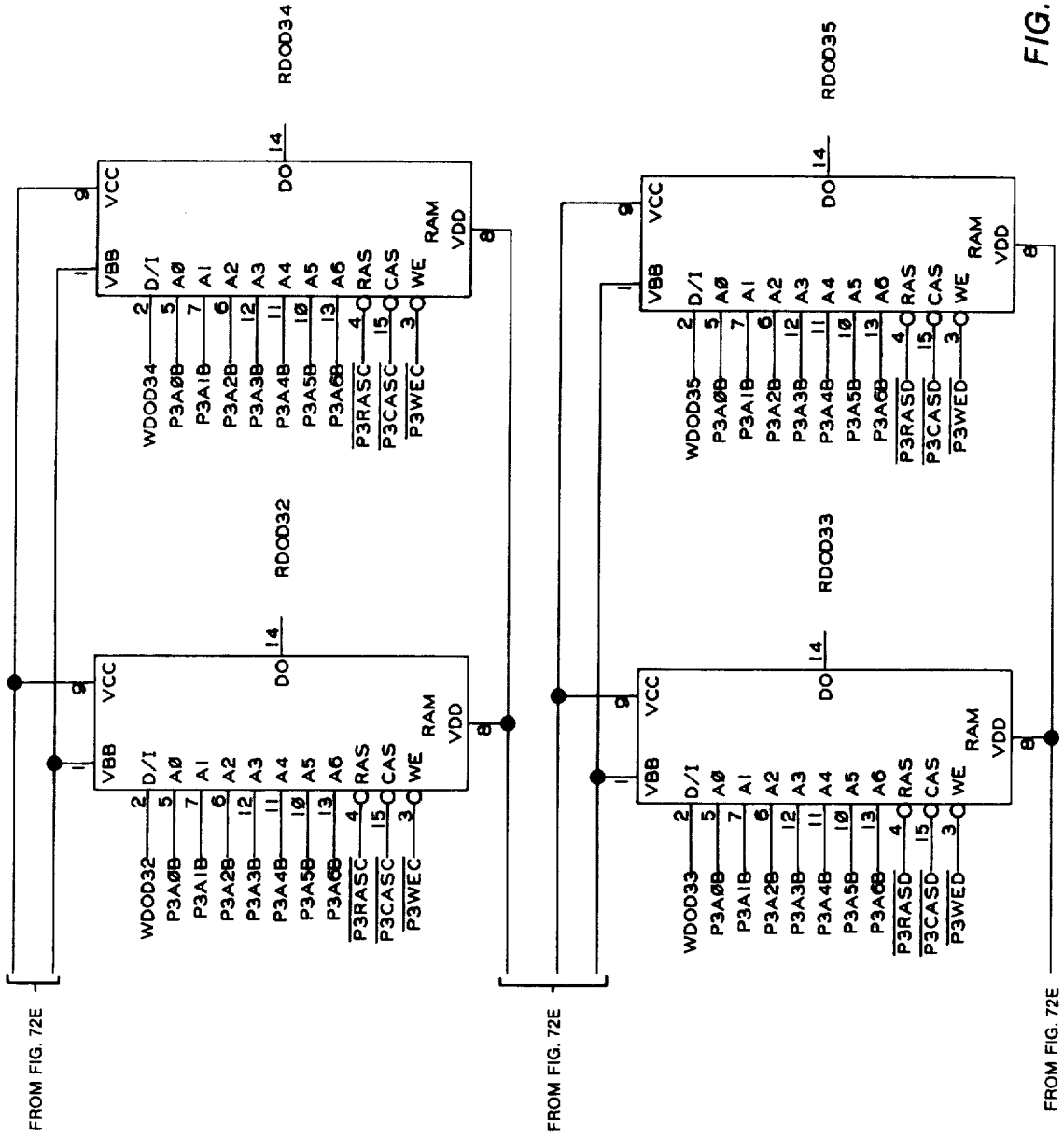

FROM FIG. 72E
RDOD32
RDOD34
RDOD33
RDOD35
VCC
VBB
DO
RAM
VDD
D/I
A0
A1
A2
A3
A4
A5
A6
RAS
CAS
WE
WDOD32
WDOD34
WDOD33
WDOD35
P3A0B
P3A1B
P3A2B
P3A3B
P3A4B
P3A5B
P3A6B
P3RASC
P3CASC
P3WEC
P3RASD
P3CASD
P3WED
FROM FIG. 72E
FROM FIG. 72E

FIG. 72F

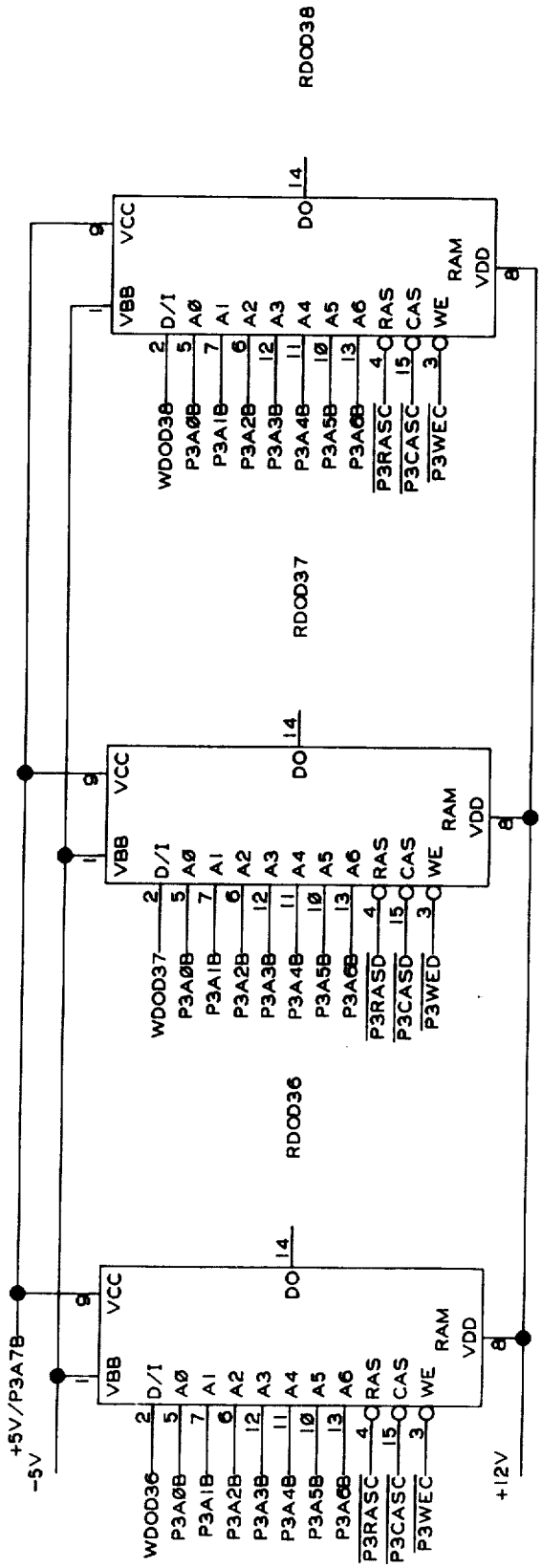
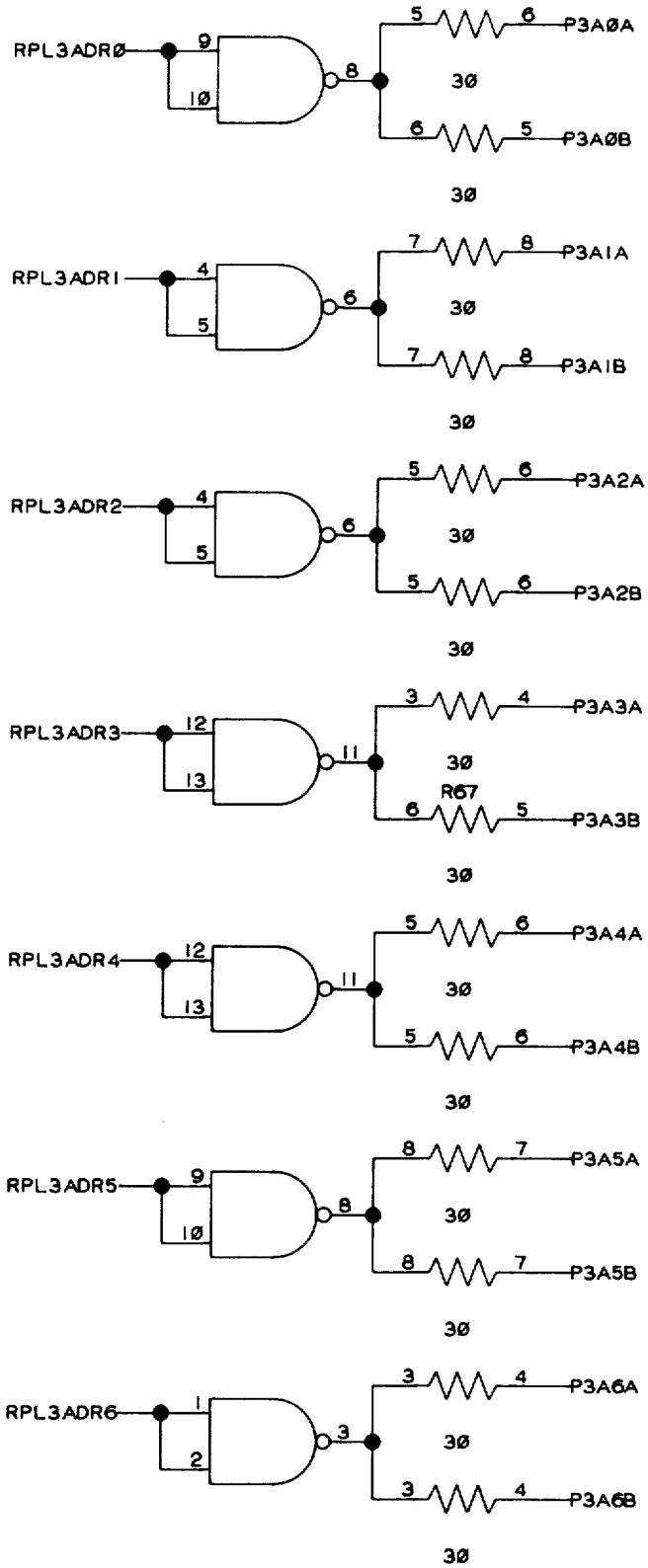

RDOD38
RDOD37
RDOD36
VCC
VBB
DO
RAM
VDD
D/I
A0
A1
A2
A3
A4
A5
A6
RAS
CAS
WE
WDOD38
P3A0B
P3A1B
P3A2B
P3A3B
P3A4B
P3A5B
P3A6B
P3RASC
P3CASC
P3WEC
WDOD37
P3RASD
P3CASD
P3WED
WDOD36
+5V/P3A7B
-5V
+12V

FIG. 72G



PLANE 3 CONTROL INPUTS

FIG. 73

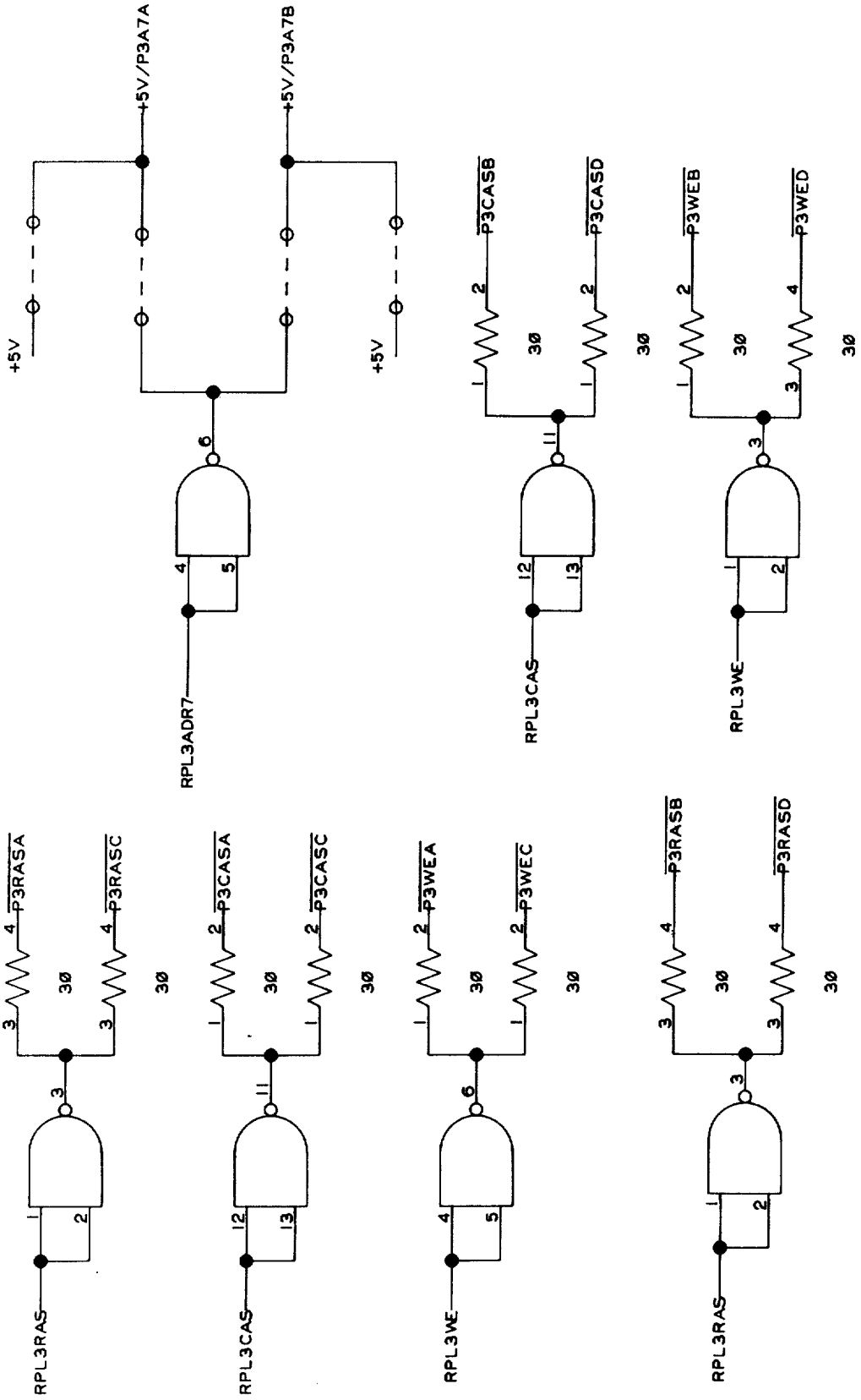
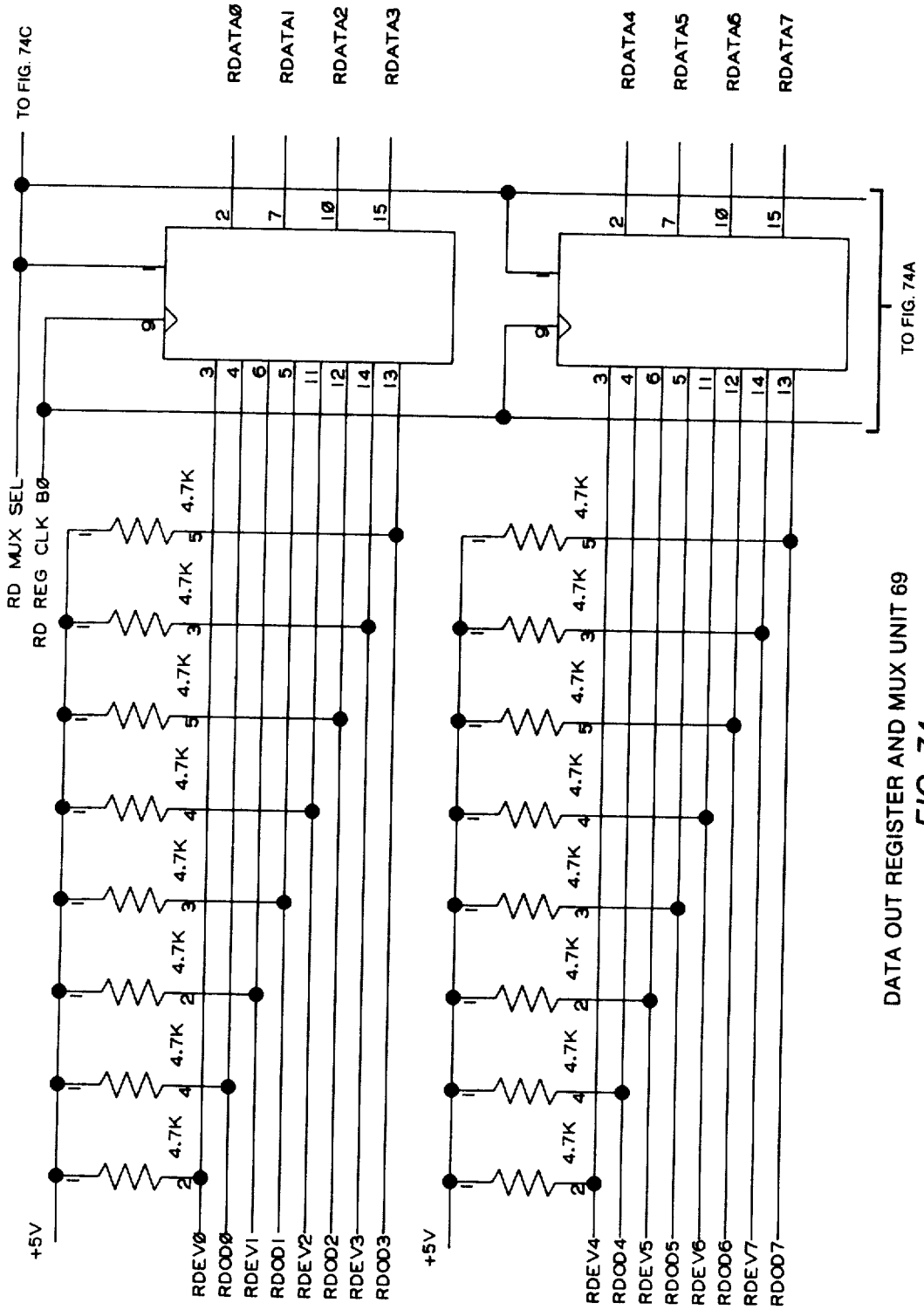

+5V
+5V/P3A7A
+5V/P3A7B
+5V
RPL3ADR7
RPL3CAS
RPL3WE
P3CASB
P3CASD
P3WEB
P3WED
RPL3RAS
RPL3CAS
RPL3WE
RPL3RAS
P3RASA
P3RASC
P3CASA
P3CASC
P3WEA
P3WEC
P3RASB
P3RASD
30

FIG. 73A



DATA OUT REGISTER AND MUX UNIT 69

FIG. 74

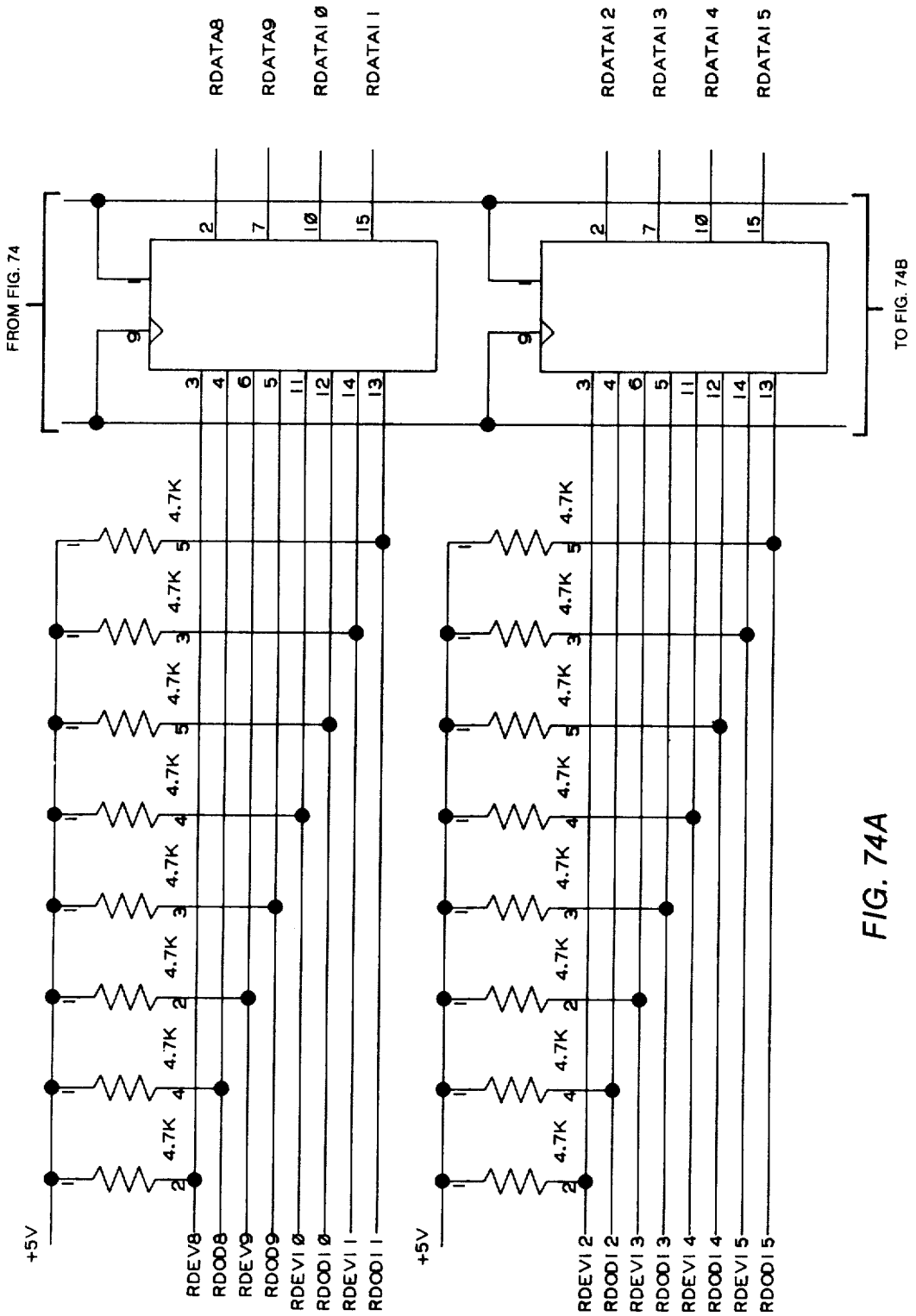

FROM FIG. 74
TO FIG. 74B
RDATA8
RDATA9
RDATAI 0
RDATAI 1
RDATAI 2
RDATAI 3
RDATAI 4
RDATAI 5
4.7K
+5V
RDEV8
RDOD8
RDEV9
RDOD9
RDEVI 0
RDODI 0
RDEVI 1
RDODI 1
RDEVI 2
RDODI 2
RDEVI 3
RDODI 3
RDEVI 4
RDODI 4
RDEVI 5
RDODI 5

FIG. 74A

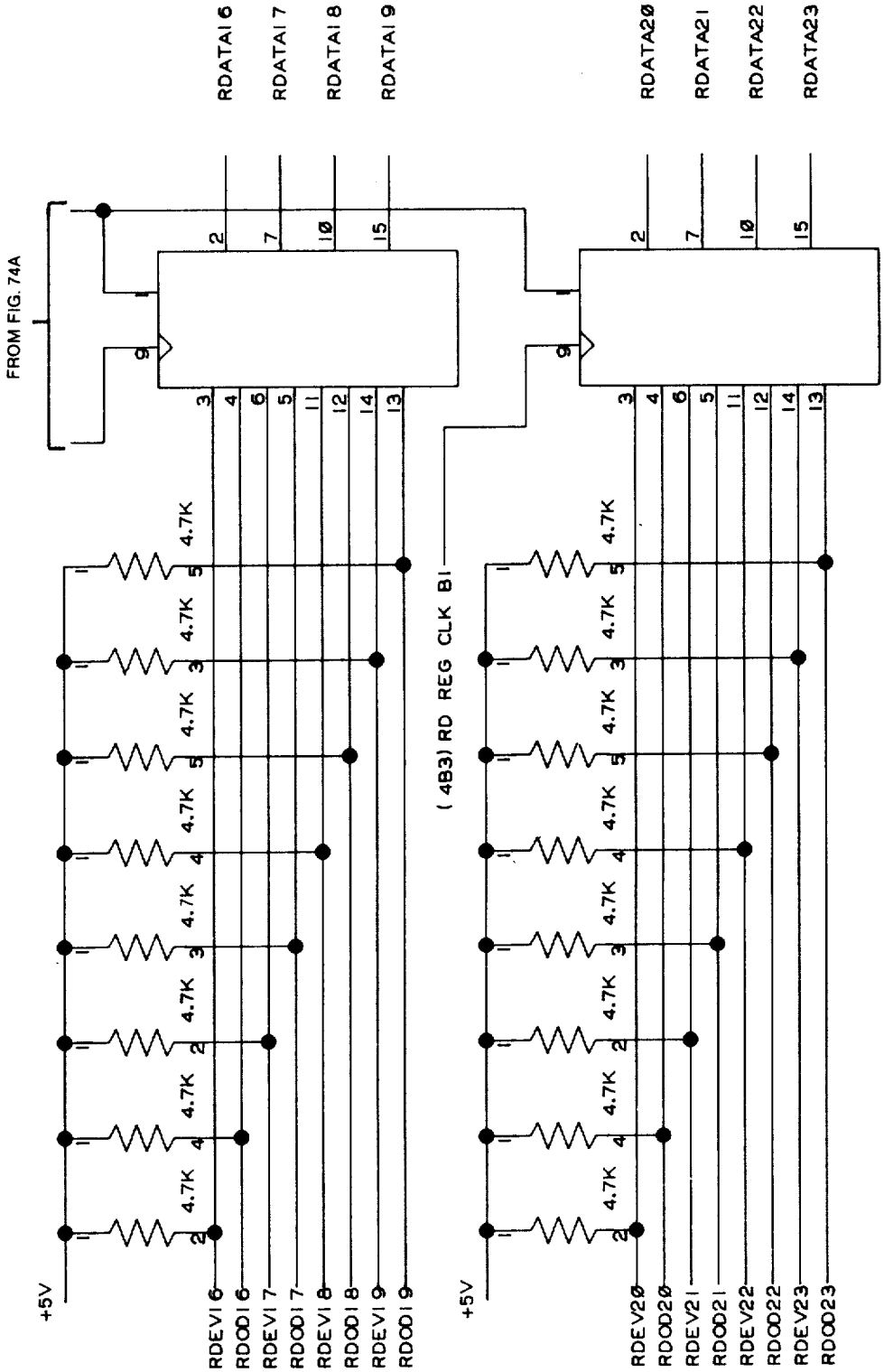

FROM FIG. 74A
RDATAI6
RDATAI7
RDATAI8
RDATAI9
RDATA20
RDATA21
RDATA22
RDATA23
4.7K
+5V
RDEVI6
RDOOI6
RDEVI7
RDOOI7
RDEVI8
RDOOI8
RDEVI9
RDOOI9
(4B3) RD REG CLK BI
RDEV20
RDOO20
RDEV21
RDOO21
RDEV22
RDOO22
RDEV23
RDOO23

FIG. 74B

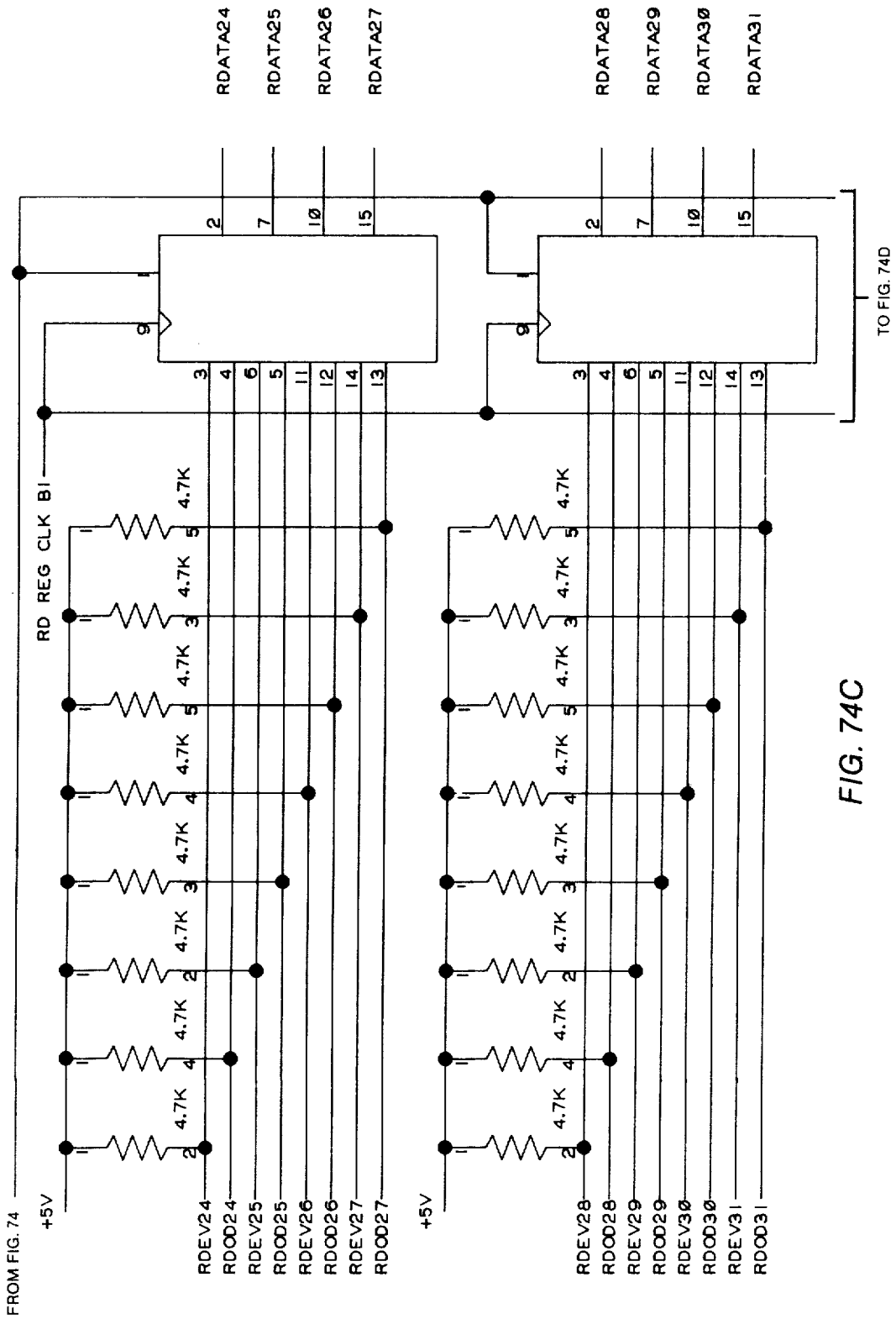

FROM FIG. 74
+5V
RD REG CLK BI
4.7K
RDEV24
RDOD24
RDEV25
RDOD25
RDEV26
RDOD26
RDEV27
RDOD27
RDATA24
RDATA25
RDATA26
RDATA27
RDEV28
RDOD28
RDEV29
RDOD29
RDEV30
RDOD30
RDEV31
RDOD31
RDATA28
RDATA29
RDATA30
RDATA31
TO FIG. 74D

FIG. 74C

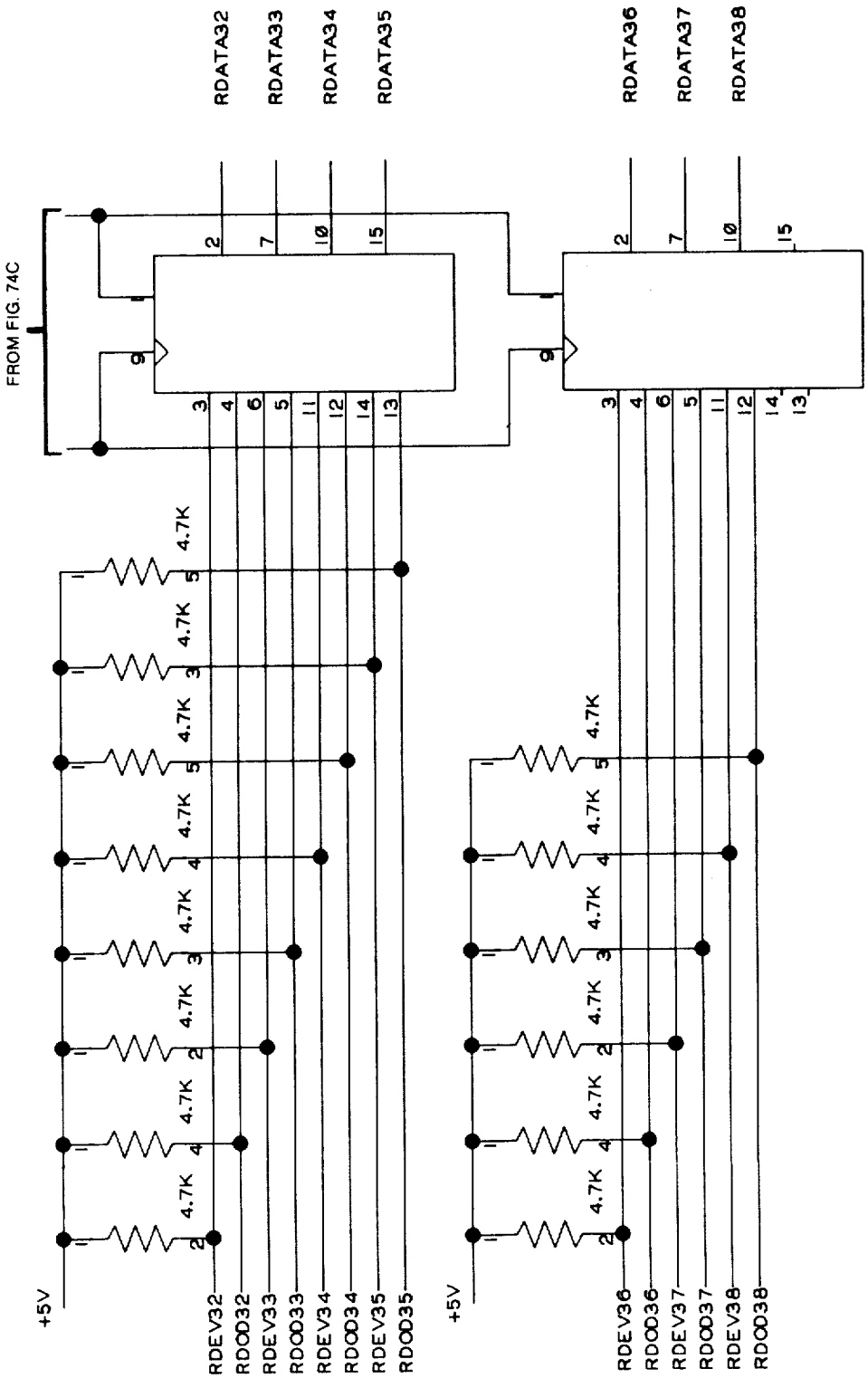

FROM FIG. 74C
RDATA32
RDATA33
RDATA34
RDATA35
RDATA36
RDATA37
RDATA38
4.7K
+5V
RDEV32
RDOD32
RDEV33
RDOD33
RDEV34
RDOD34
RDEV35
RDOD35
RDEV36
RDOD36
RDEV37
RDOD37
RDEV38
RDOD38

FIG. 74D

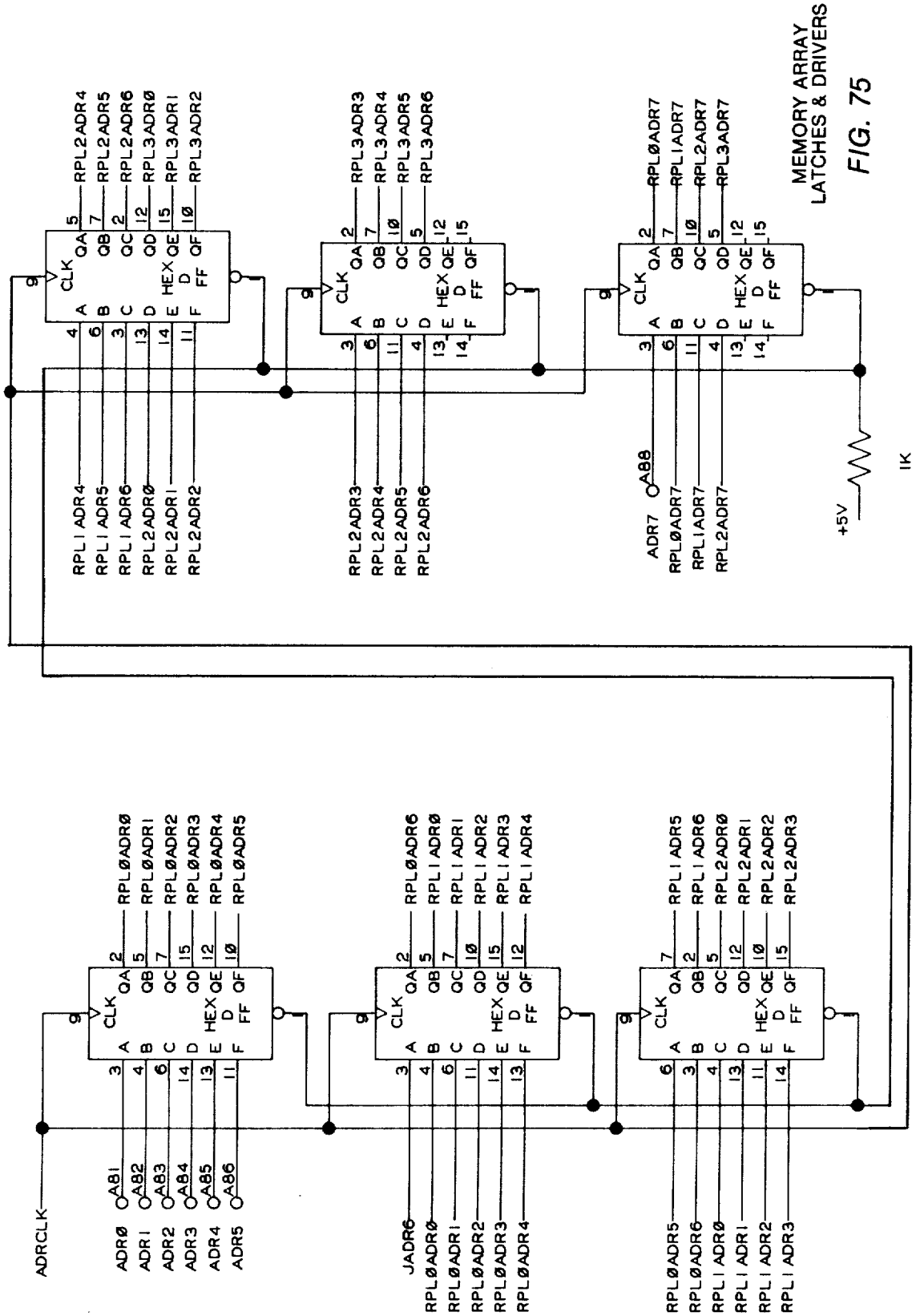
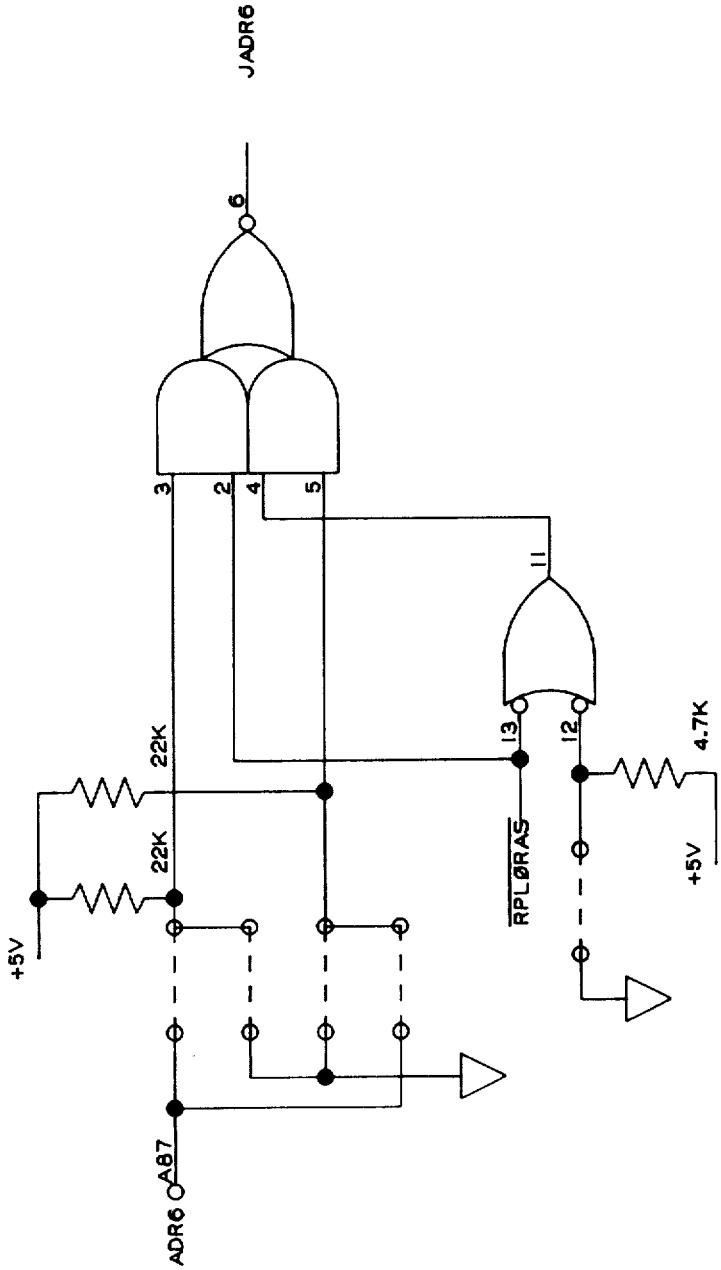
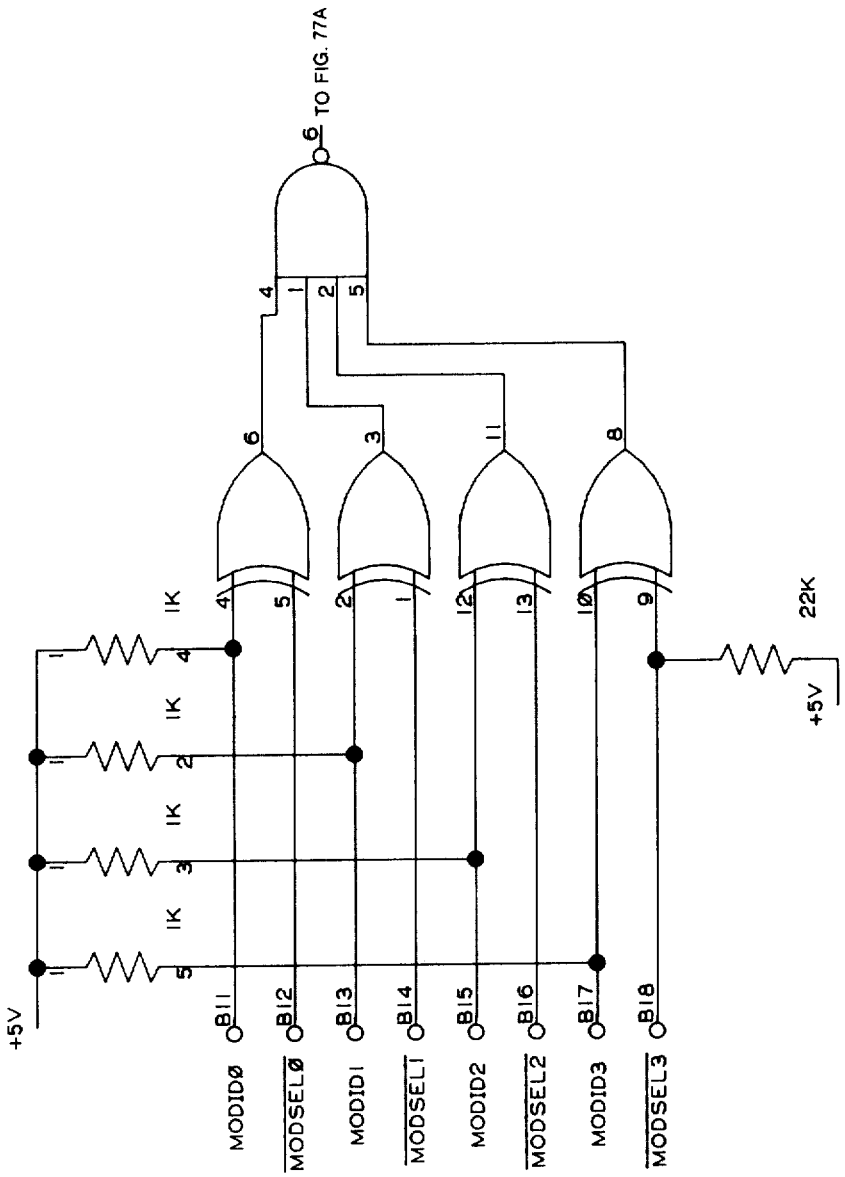

MEMORY ARRAY
LATCHES & DRIVERS
CLK
HEX
D
FF
QA
QB
QC
QD
QE
QF
RPL2ADR4
RPL2ADR5
RPL2ADR6
RPL3ADR0
RPL3ADR1
RPL3ADR2
RPL3ADR3
RPL3ADR4
RPL3ADR5
RPL3ADR6
RPL0ADR7
RPL1ADR7
RPL2ADR7
RPL3ADR7
RPL1ADR4
RPL1ADR5
RPL1ADR6
RPL2ADR0
RPL2ADR1
RPL2ADR2
RPL2ADR3
RPL2ADR4
RPL2ADR5
RPL2ADR6
A88
ADR7
+5V
1K
RPL0ADR0
RPL0ADR1
RPL0ADR2
RPL0ADR3
RPL0ADR4
RPL0ADR5
RPL0ADR6
RPL1ADR0
RPL1ADR1
RPL1ADR2
RPL1ADR3
RPL1ADR4
RPL1ADR5
RPL1ADR6
RPL2ADR0
RPL2ADR1
RPL2ADR2
RPL2ADR3
ADRCLK
A81
A82
A83
A84
A85
A86
ADR0
ADR1
ADR2
ADR3
ADR4
ADR5
JADR6

FIG. 75



RAM SEL LOGIC 68

FIG. 76



MODSEL COMPARATOR AND MEMORY MODULE CONTROL LOGIC

FIG. 77

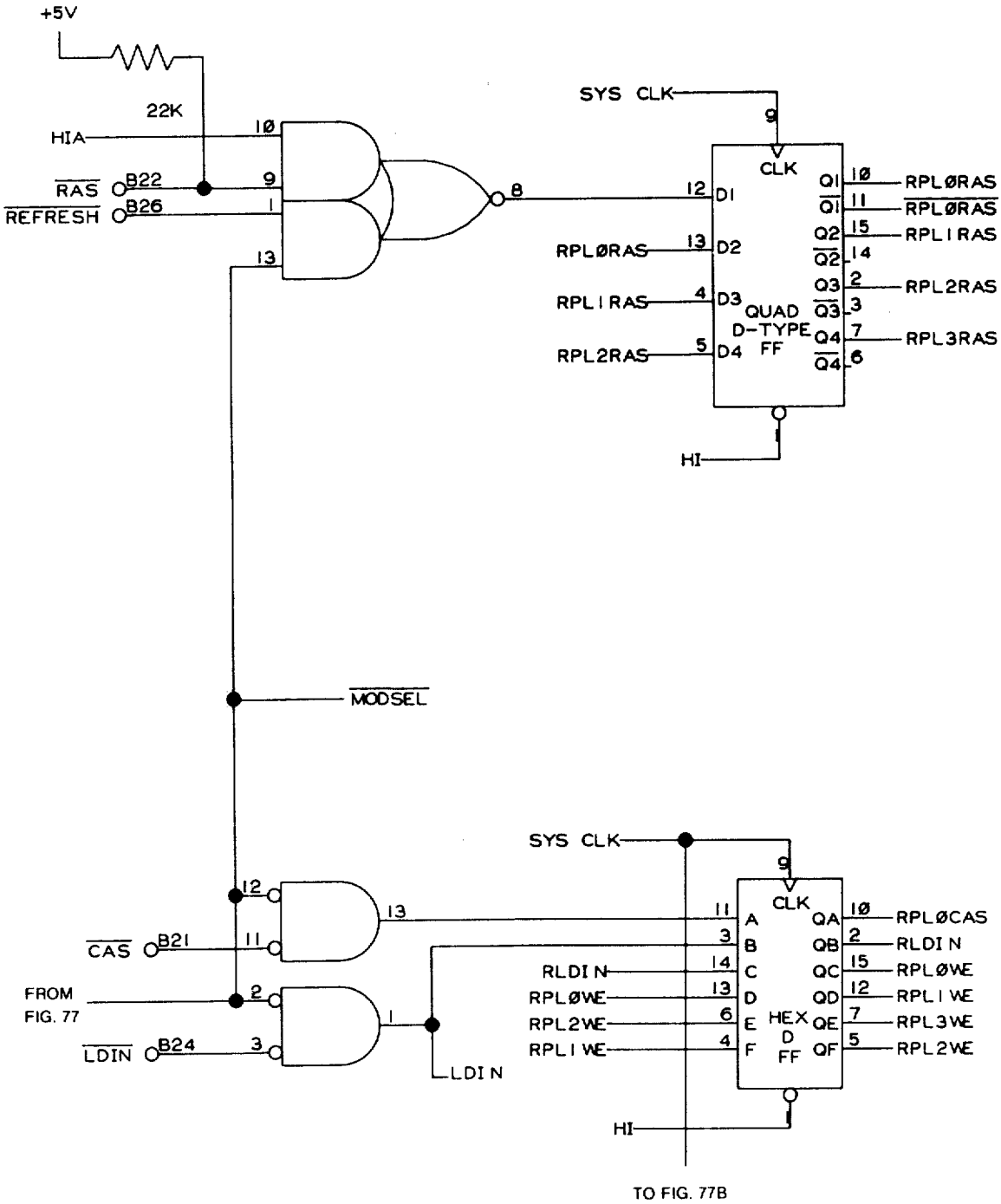

+5V
22K
HIA
RAS
B22
REFRESH
B26
SYS CLK
CLK
D1
D2
D3
D4
QUAD
D-TYPE
FF
Q1
Q2
Q3
Q4
RPLØRAS
RPL1RAS
RPL2RAS
RPL3RAS
HI
MODSEL
CAS
B21
FROM
FIG. 77
LDIN
B24
RLDIN
RPLØWE
RPL2WE
RPL1WE
A
B
C
D
E
F
HEX
D
FF
QA
QB
QC
QD
QE
QF
RPLØCAS
RLDIN
RPLØWE
RPL1WE
RPL3WE
RPL2WE
TO FIG. 77B

FIG. 77A

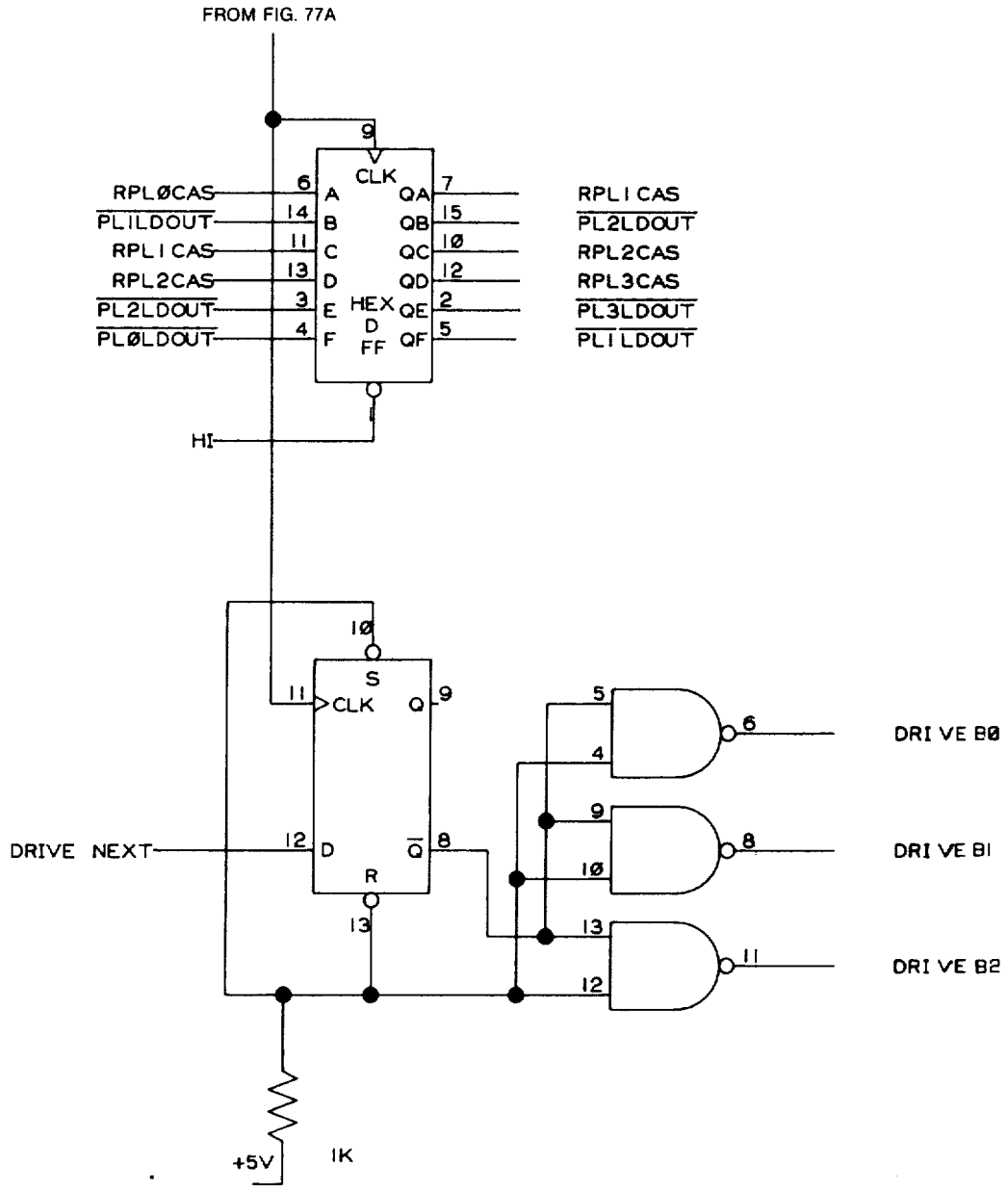

FROM FIG. 77A
CLK
RPLØCAS
PL1LDOUT
RPL1CAS
RPL2CAS
PL2LDOUT
PLØLDOUT
A
B
C
D
E
F
HEX
D
FF
QA
QB
QC
QD
QE
QF
RPL1CAS
PL2LDOUT
RPL2CAS
RPL3CAS
PL3LDOUT
PL1LDOUT
HI
S
CLK
Q
D
R
DRIVE NEXT
DRIVE B0
DRIVE B1
DRIVE B2
+5V
1K

FIG. 77B

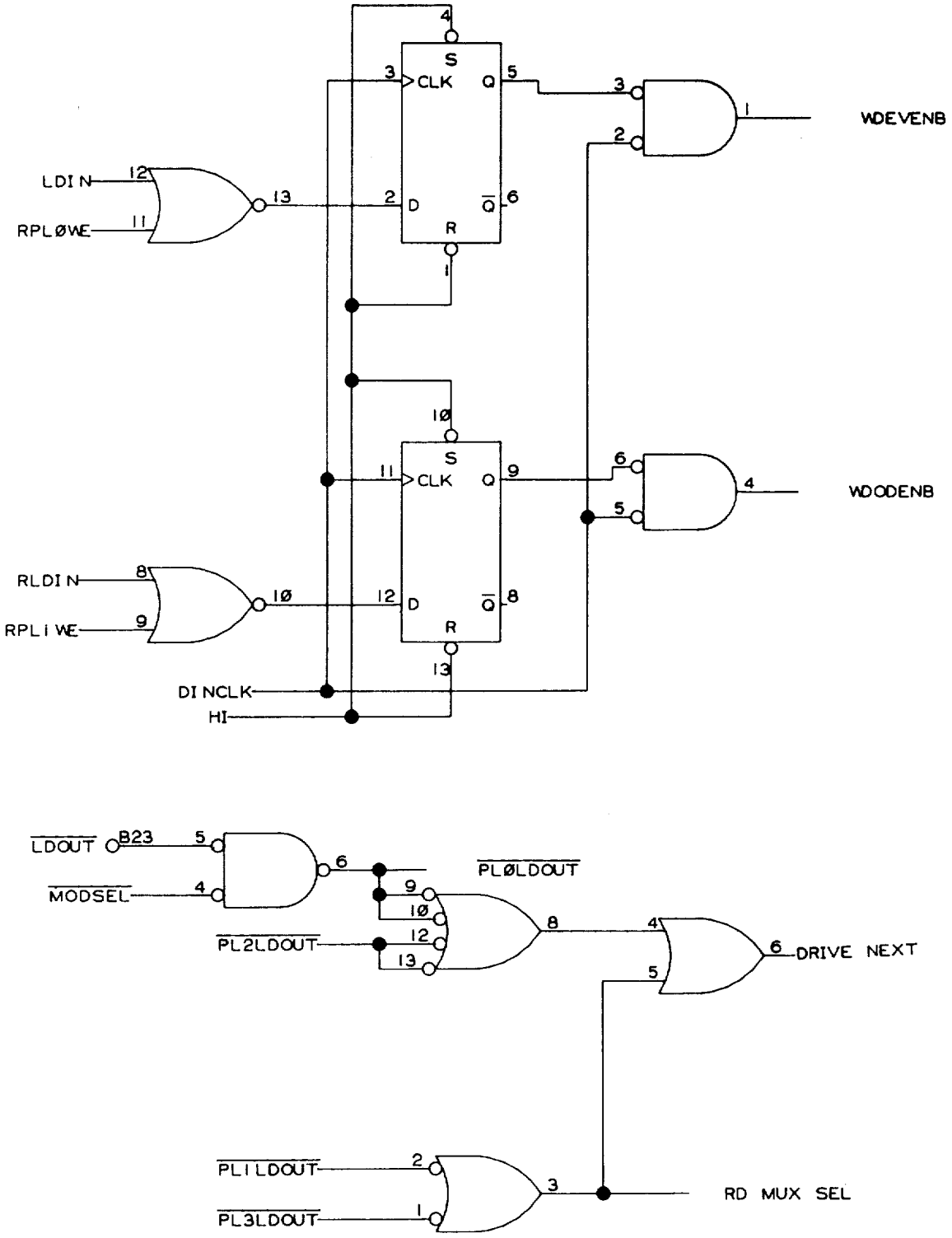
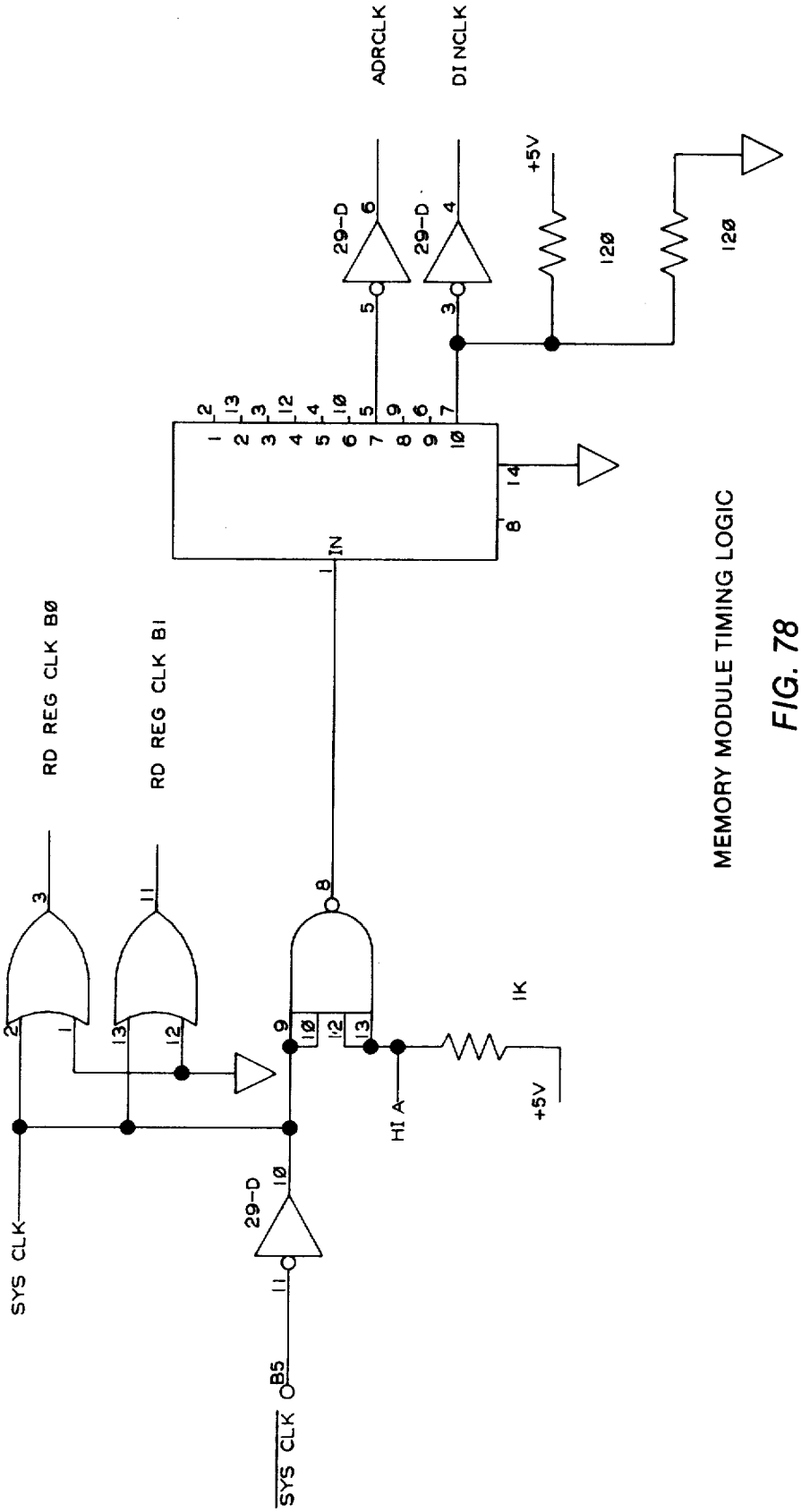

LDIN
RPLØWE
RLDIN
RPLIWE
DINCLK
HI
S
CLK
D
R
Q
WDEVENB
WDODENB
LDOUT
B23
MODSEL
PL2LDOUT
PLØLDOUT
DRIVE NEXT
PL1LDOUT
PL3LDOUT
RD MUX SEL

FIG. 77C



MEMORY MODULE TIMING LOGIC

FIG. 78

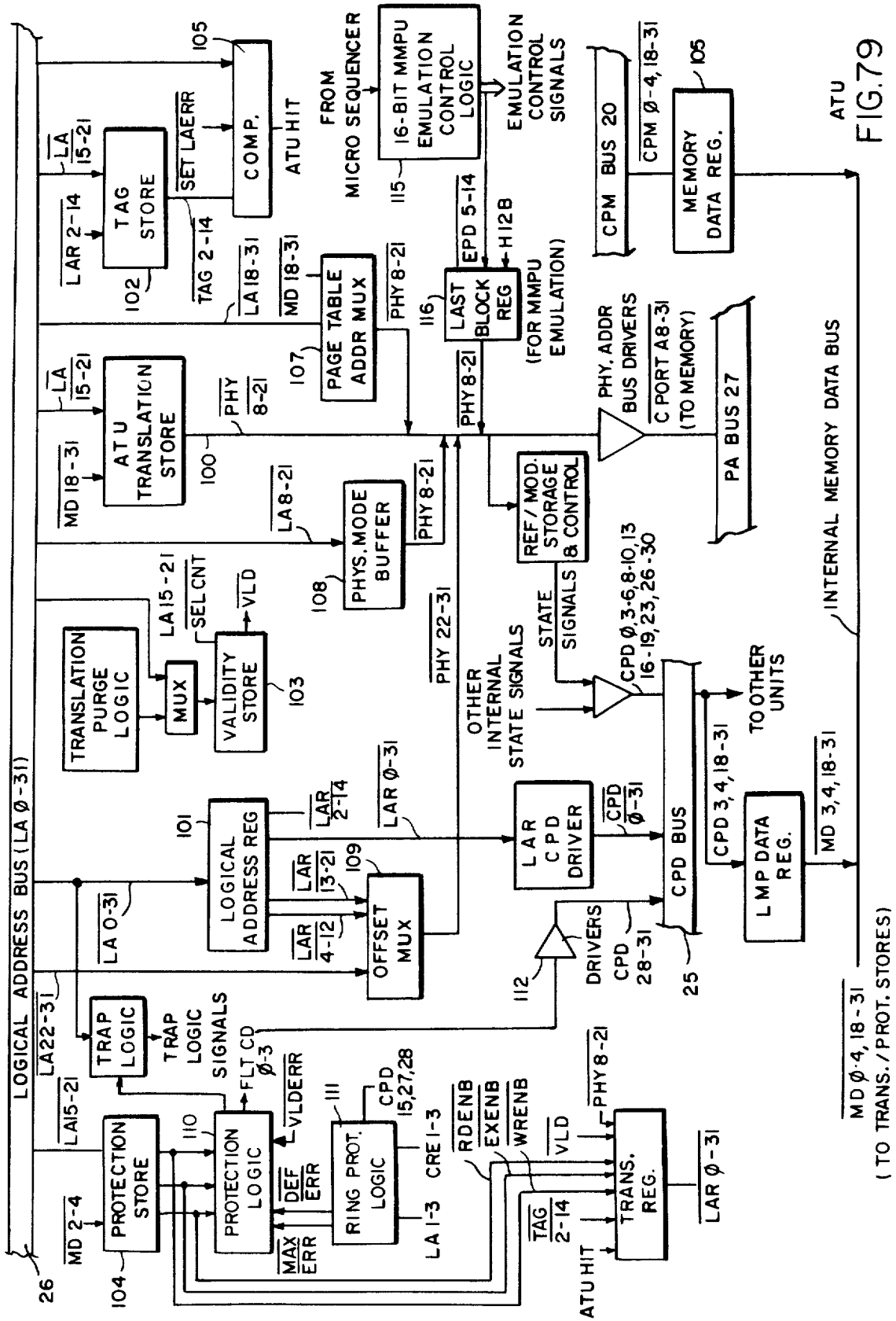

LOGICAL ADDRESS BUS (LA Ø-31)
26
MD 2-4
PROTECTION STORE
104
LA15-21
LA22-31
TRAP LOGIC
TRAP LOGIC SIGNALS
110
PROTECTION LOGIC
FLT CD Ø-3
VLDERR
DEF ERR
MAX ERR
111
RING PROT. LOGIC
CPD 15,27,28
LA 1-3
CRE 1-3
RDENB
EXENB
WRENB
TAG 2-14
ATU HIT
TRANS. REG.
PHY 8-21
VLD
LAR Ø-31
LA 0-31
101
LOGICAL ADDRESS REG
LAR 2-14
LAR 13-21
LAR 4-12
109
OFFSET MUX
LAR Ø-31
112
DRIVERS
CPD 28-31
25
CPD BUS
LAR CPD DRIVER
CPD Ø-31
OTHER INTERNAL STATE SIGNALS
PHY 22-31
TRANSLATION PURGE LOGIC
MUX
LA 15-21
SEL CNT
VALIDITY STORE
VLD
103
108
PHYS. MODE BUFFER
LA 8-21
PHY 8-21
MD 18-31
ATU TRANSLATION STORE
LA 15-21
100
PHY 8-21
107
PAGE TABLE ADDR MUX
LA 18-31
MD 18-31
PHY 8-21
LAR 2-14
TAG STORE
102
LA 15-21
SET LAERR
TAG 2-14
105
COMP.
ATU HIT
FROM MICRO SEQUENCER
115
16-BIT MMPU EMULATION CONTROL LOGIC
EMULATION CONTROL SIGNALS
EPD 5-14
H12B
116
LAST BLOCK REG
(FOR MMPU EMULATION)
PHY 8-21
REF/MOD. STORAGE & CONTROL
STATE SIGNALS
CPD Ø,3-6,8-10,13 16-19,23,26-30
PHY, ADDR BUS DRIVERS
C PORT A8-31 (TO MEMORY)
PA BUS 27
CPM BUS 20
CPM Ø-4, 18-31
105
MEMORY DATA REG.
ATU
INTERNAL MEMORY DATA BUS
TO OTHER UNITS
CPD 3,4,18-31
LMP DATA REG.
MD 3,4,18-31
MD Ø-4, 18-31 (TO TRANS./PROT. STORES)

FIG. 79

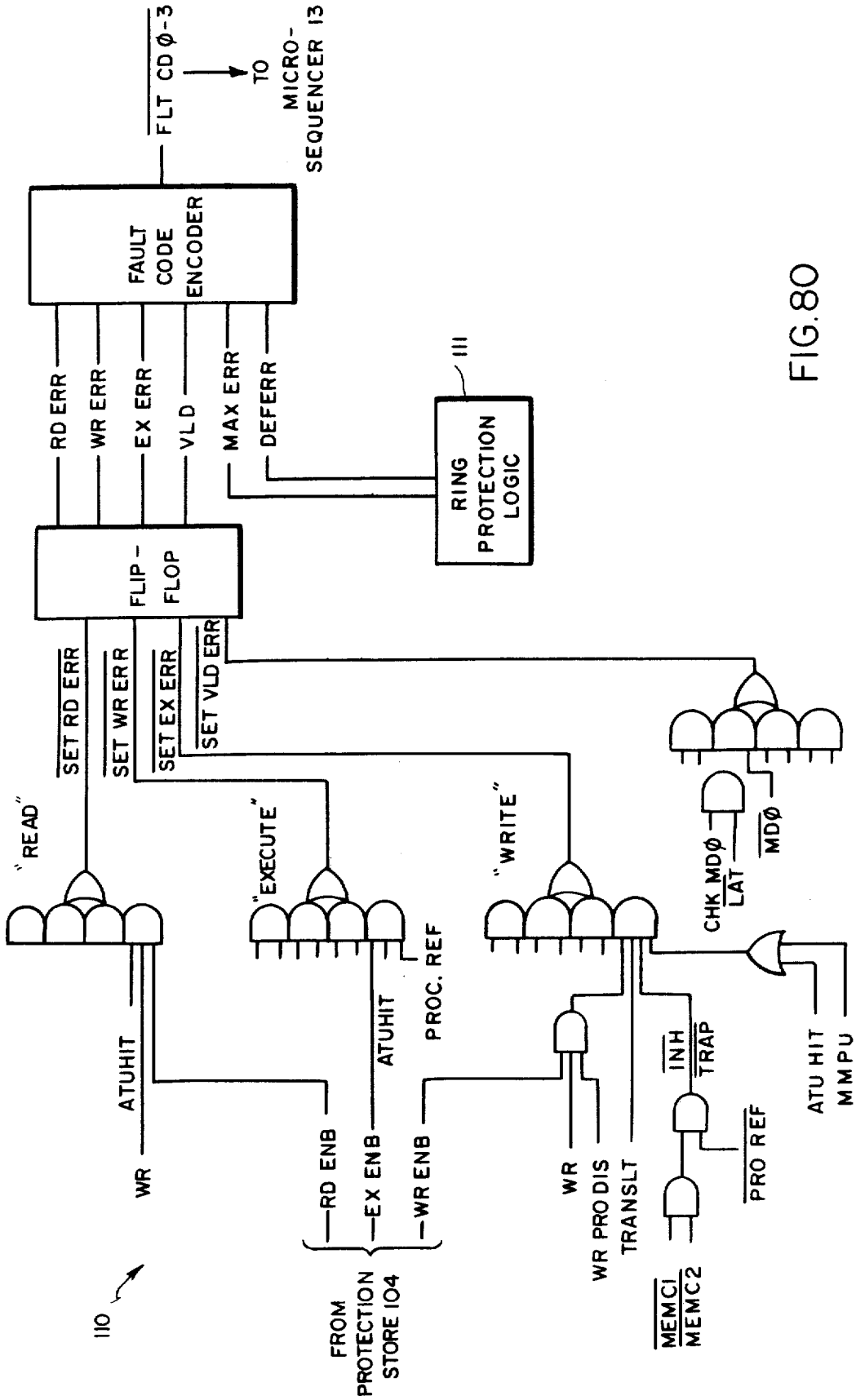

FLT CD Ø-3
TO MICRO-SEQUENCER 13
FAULT CODE ENCODER
RD ERR
WR ERR
EX ERR
VLD
MAX ERR
DEFERR
111
RING PROTECTION LOGIC
FLIP-FLOP
SET RD ERR
SET WR ERR
SET EX ERR
SET VLD ERR
"READ"
"EXECUTE"
"WRITE"
CHK MDØ
LAT
MDØ
ATUHIT
WR
RD ENB
EX ENB
WR ENB
PROC. REF
INH
TRAP
PRO REF
WR PRO DIS
TRANSLT
ATU HIT
MMPU
MEMC1
MEMC2
110
FROM PROTECTION STORE 104

FIG. 80

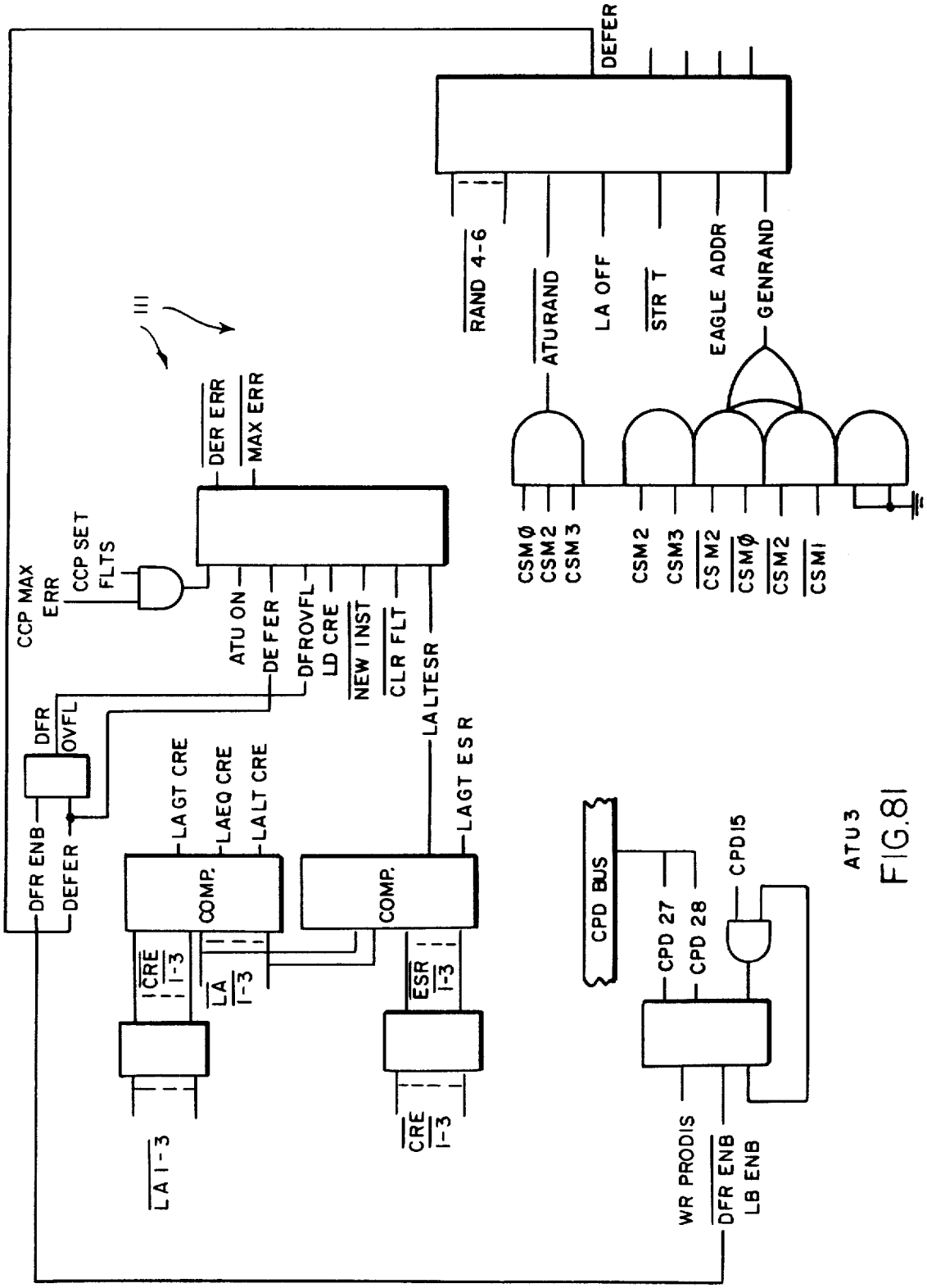
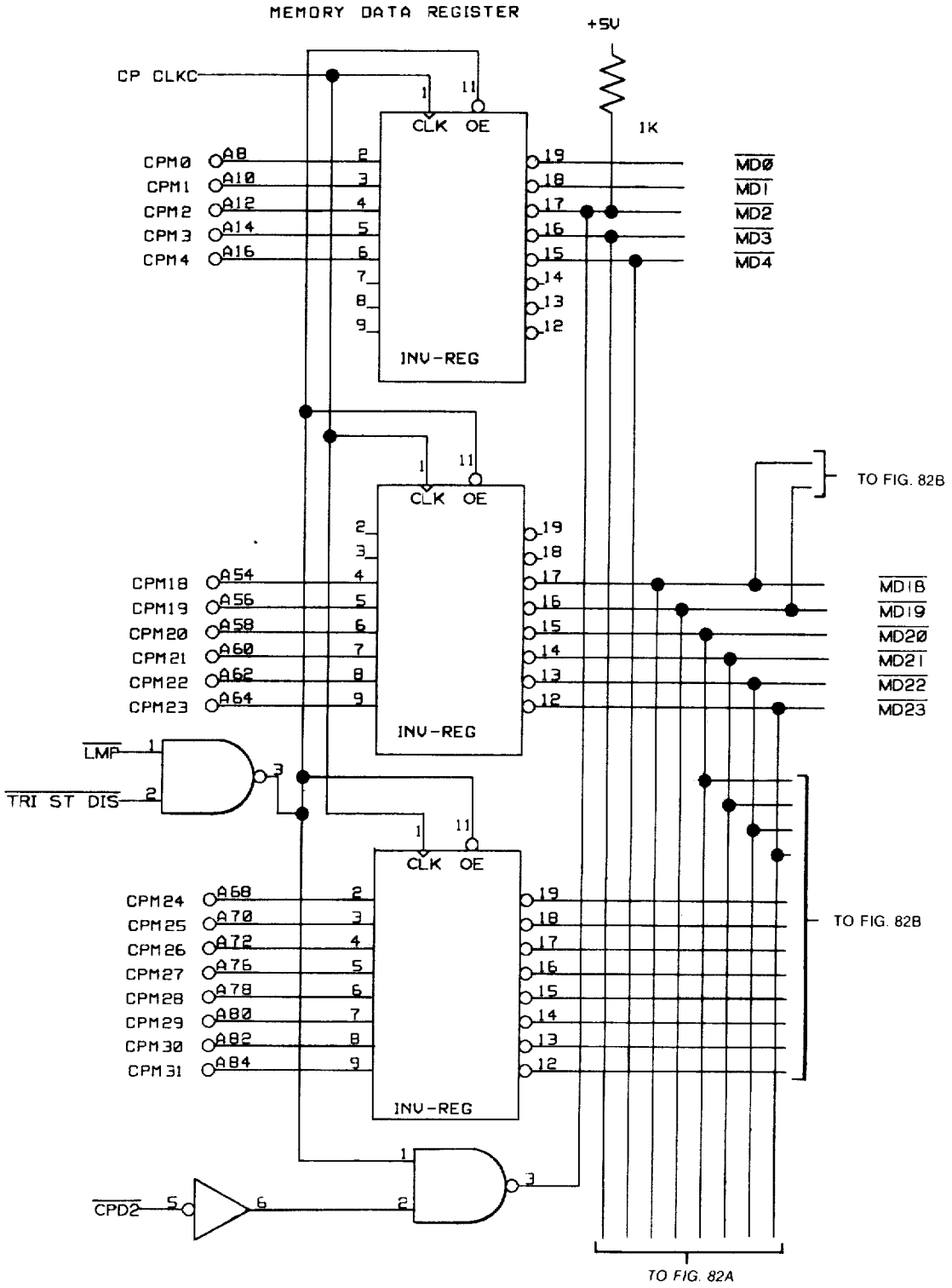

DEFER
RAND 4-6
ATURAND
LA OFF
STR T
EAGLE ADDR
GENRAND
CSMØ
CSM2
CSM3
CSM2
CSM3
CSM2
CSMØ
CSM2
CSM1
DER ERR
MAX ERR
CCP MAX ERR
CCP SET FLTS
ATU ON
DEFER
DFROVFL
LD CRE
NEW INST
CLR FLT
LALTESR
DFR
OVFL
DFR ENB
DEFER
LAGT CRE
LAEQ CRE
LALT CRE
COMP.
COMP.
LAGT ESR
CRE 1-3
LA 1-3
ESR 1-3
CRE 1-3
LA 1-3
CPD BUS
CPD 27
CPD 28
CPD15
WR PRODIS
DFR ENB
LB ENB
ATU3

FIG.81



MEMORY DATA REGISTER 106
LMP DATA REGISTER
TRANSLATION STORE 100
PHYSICAL PAGE TABLE MUX 107
PHYSICAL MODE BUFFER 108
LAST BLOCK REGISTER 116

FIG. 82

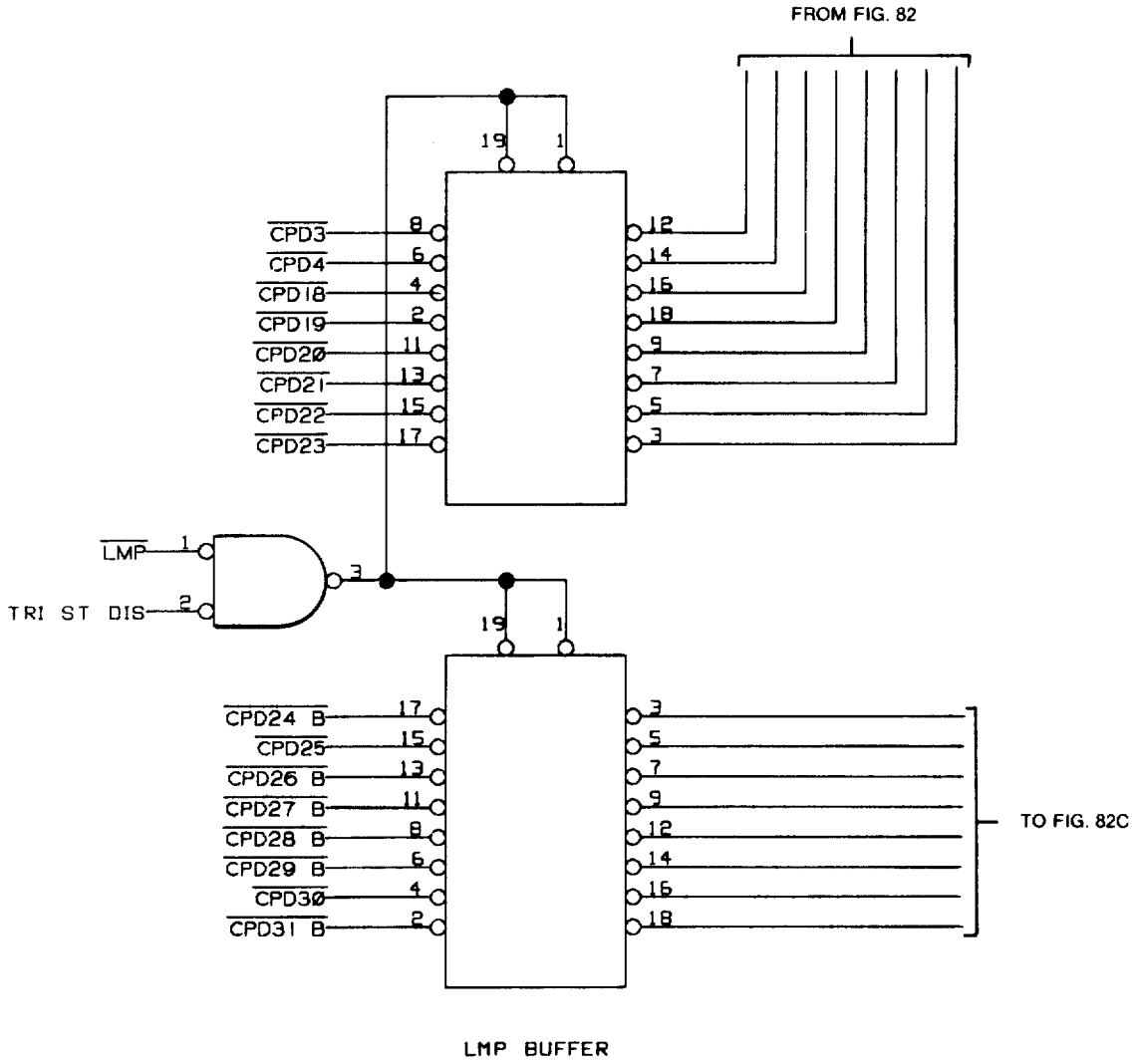

FROM FIG. 82
CPD3
CPD4
CPD18
CPD19
CPD20
CPD21
CPD22
CPD23
LMP
TRI ST DIS
CPD24 B
CPD25
CPD26 B
CPD27 B
CPD28 B
CPD29 B
CPD30
CPD31 B
TO FIG. 82C
LMP BUFFER

FIG. 82A

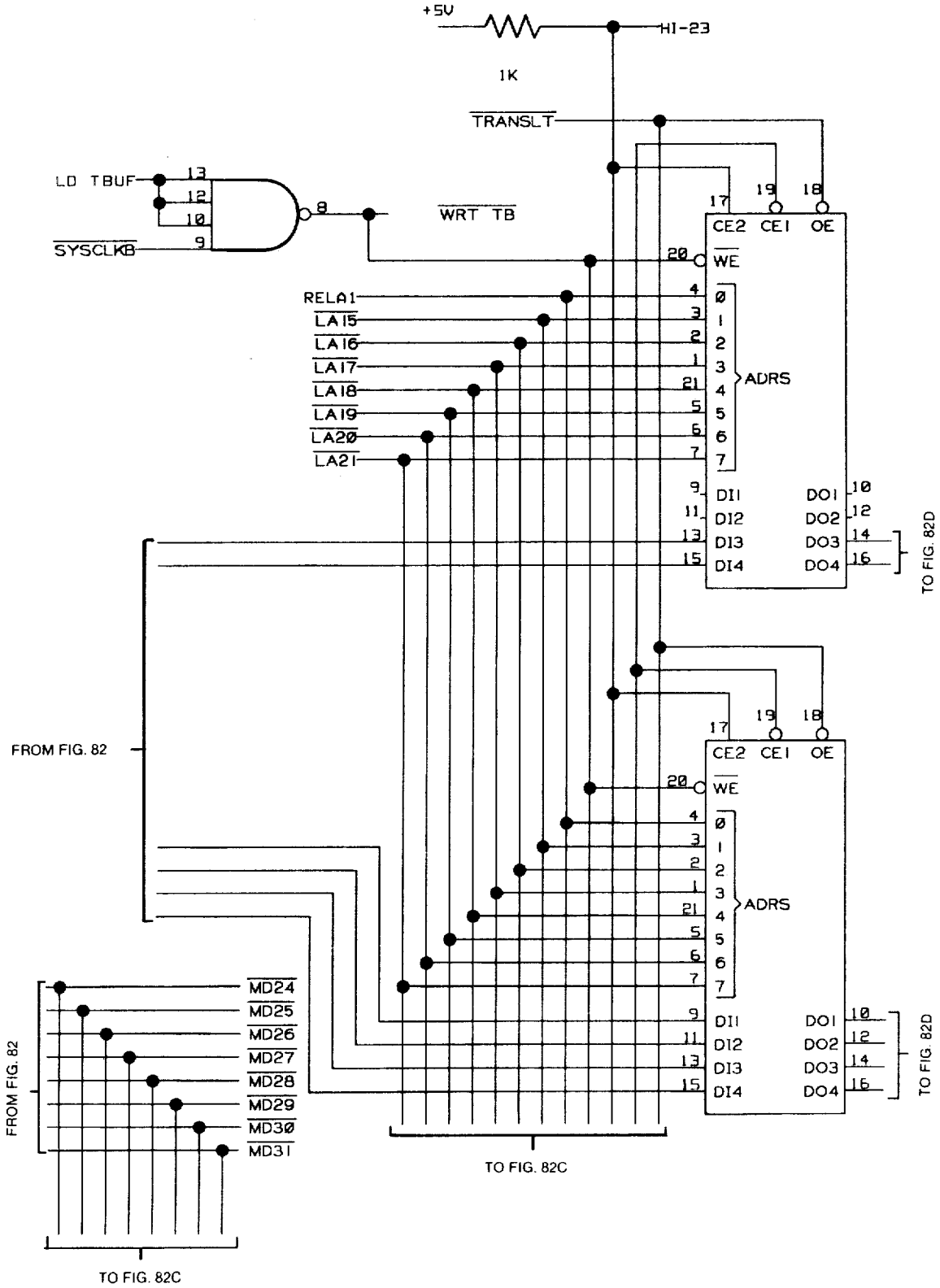

+5V
HI-23
1K
TRANSLT
LD TBUF
SYSCLKB
WRT TB
CE2
CE1
OE
WE
RELA1
LA15
LA16
LA17
LA18
LA19
LA20
LA21
ADRS
DI1
DI2
DI3
DI4
DO1
DO2
DO3
DO4
TO FIG. 82D
FROM FIG. 82
MD24
MD25
MD26
MD27
MD28
MD29
MD30
MD31
TO FIG. 82C

FIG. 82B

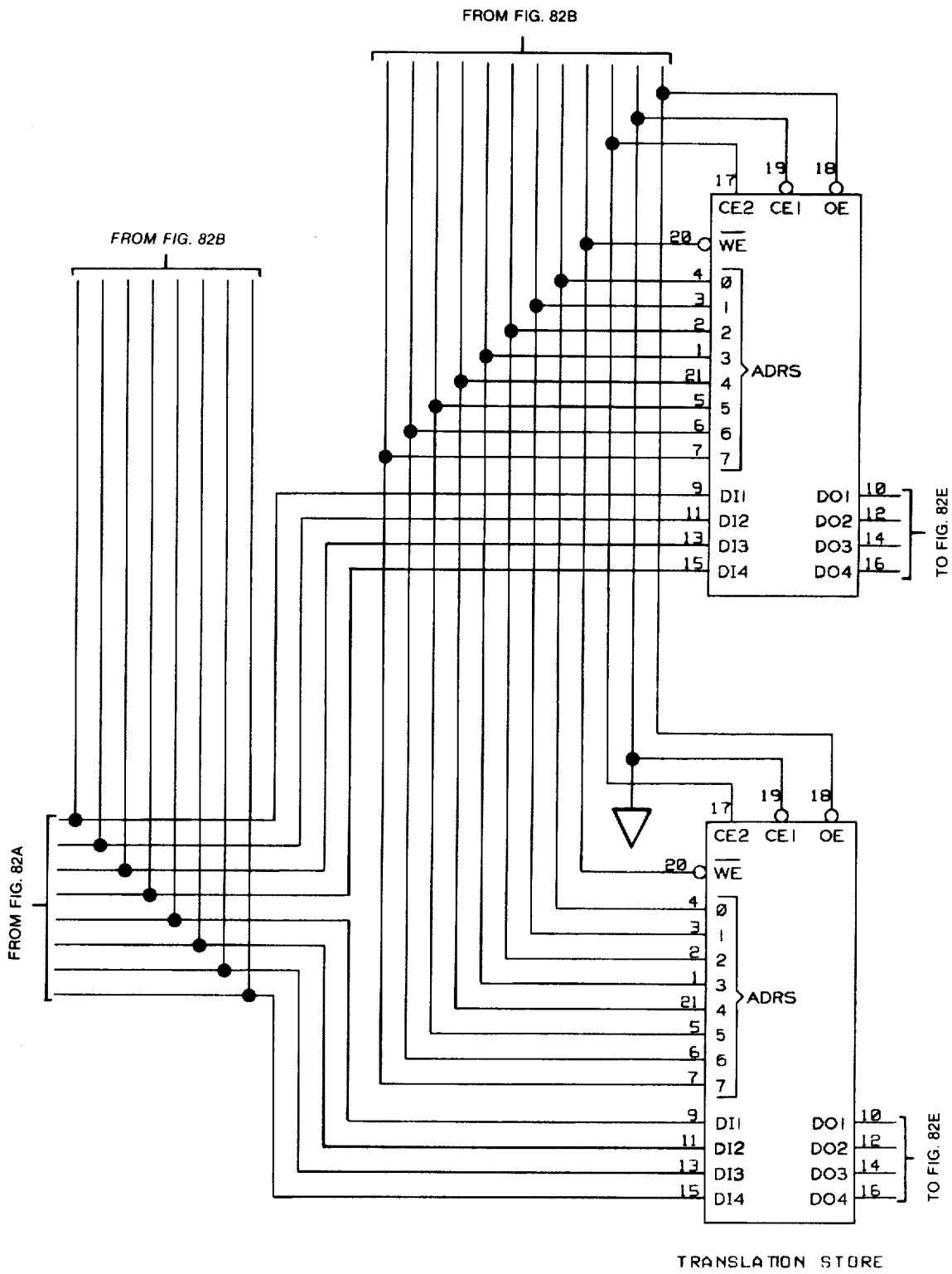

FROM FIG. 82B
FROM FIG. 82B
FROM FIG. 82A
CE2
CE1
OE
WE
ADRS
DI1
DI2
DI3
DI4
DO1
DO2
DO3
DO4
TO FIG. 82E
TO FIG. 82E
TRANSLATION STORE

FIG. 82C

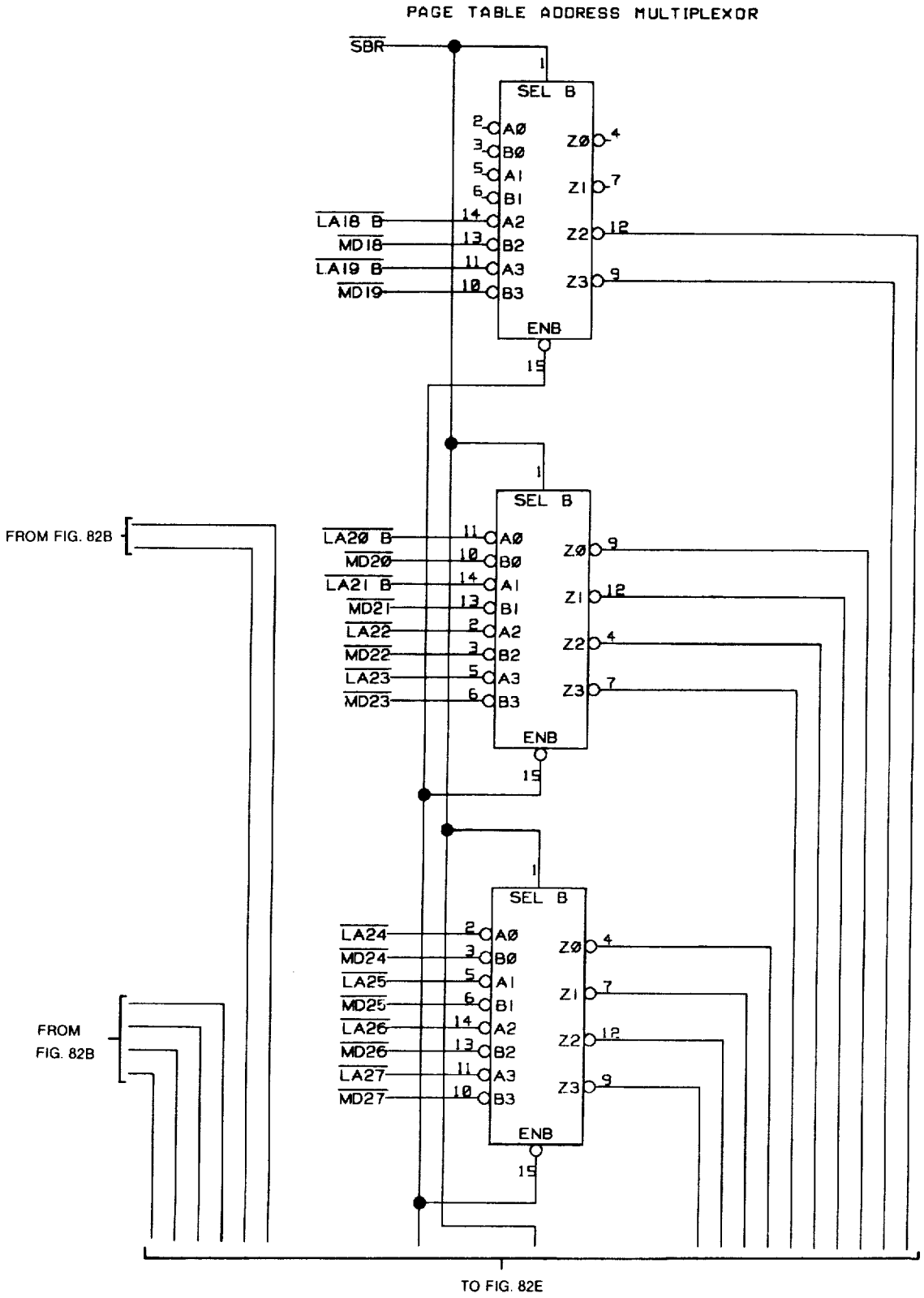

PAGE TABLE ADDRESS MULTIPLEXOR
SBR
SEL B
A0
B0
A1
B1
A2
B2
A3
B3
Z0
Z1
Z2
Z3
ENB
LA18 B
MD18
LA19 B
MD19
LA20 B
MD20
LA21 B
MD21
LA22
MD22
LA23
MD23
LA24
MD24
LA25
MD25
LA26
MD26
LA27
MD27
FROM FIG. 82B
FROM FIG. 82B
TO FIG. 82E

FIG. 82D

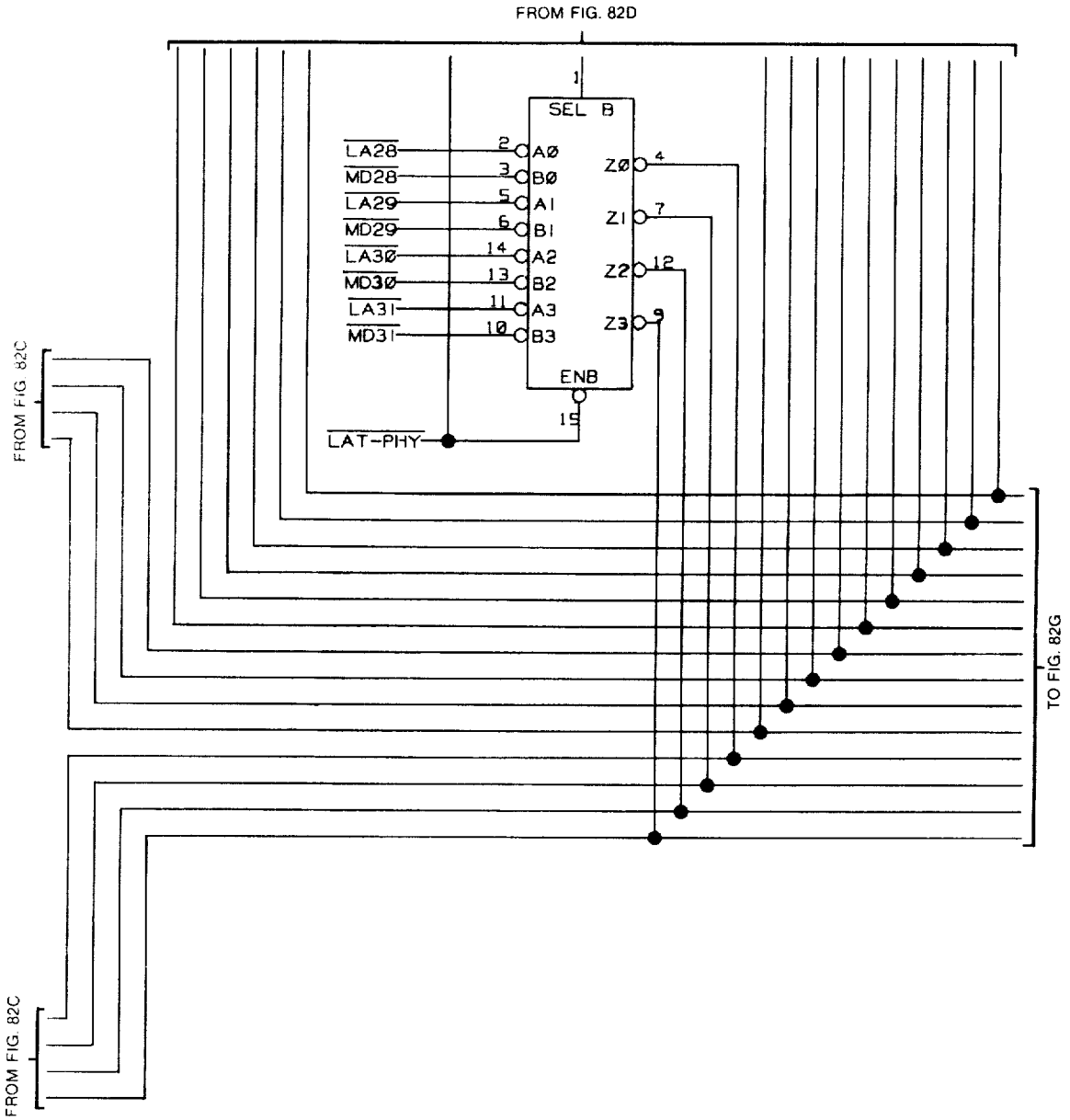

FROM FIG. 82D
SEL B
1
LA28
MD28
LA29
MD29
LA30
MD30
LA31
MD31
2
3
5
6
14
13
11
10
A0
B0
A1
B1
A2
B2
A3
B3
Z0
Z1
Z2
Z3
4
7
12
9
ENB
15
LAT-PHY
FROM FIG. 82C
FROM FIG. 82C
TO FIG. 82G

FIG. 82E

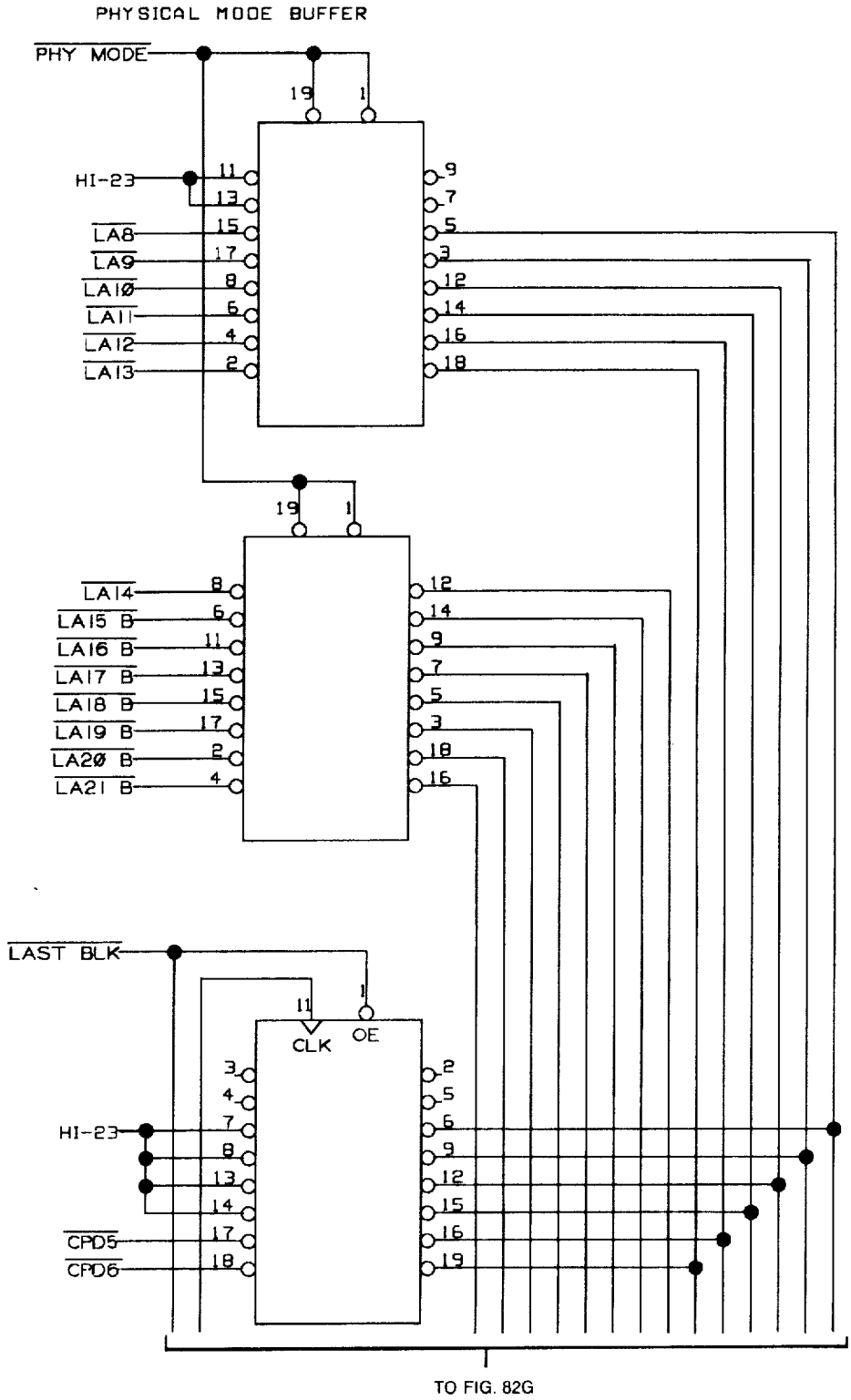

PHYSICAL MODE BUFFER
PHY MODE
HI-23
LA8
LA9
LA10
LA11
LA12
LA13
LA14
LA15 B
LA16 B
LA17 B
LA18 B
LA19 B
LA20 B
LA21 B
LAST BLK
CLK
OE
HI-23
CPD5
CPD6
TO FIG. 82G

FIG. 82F

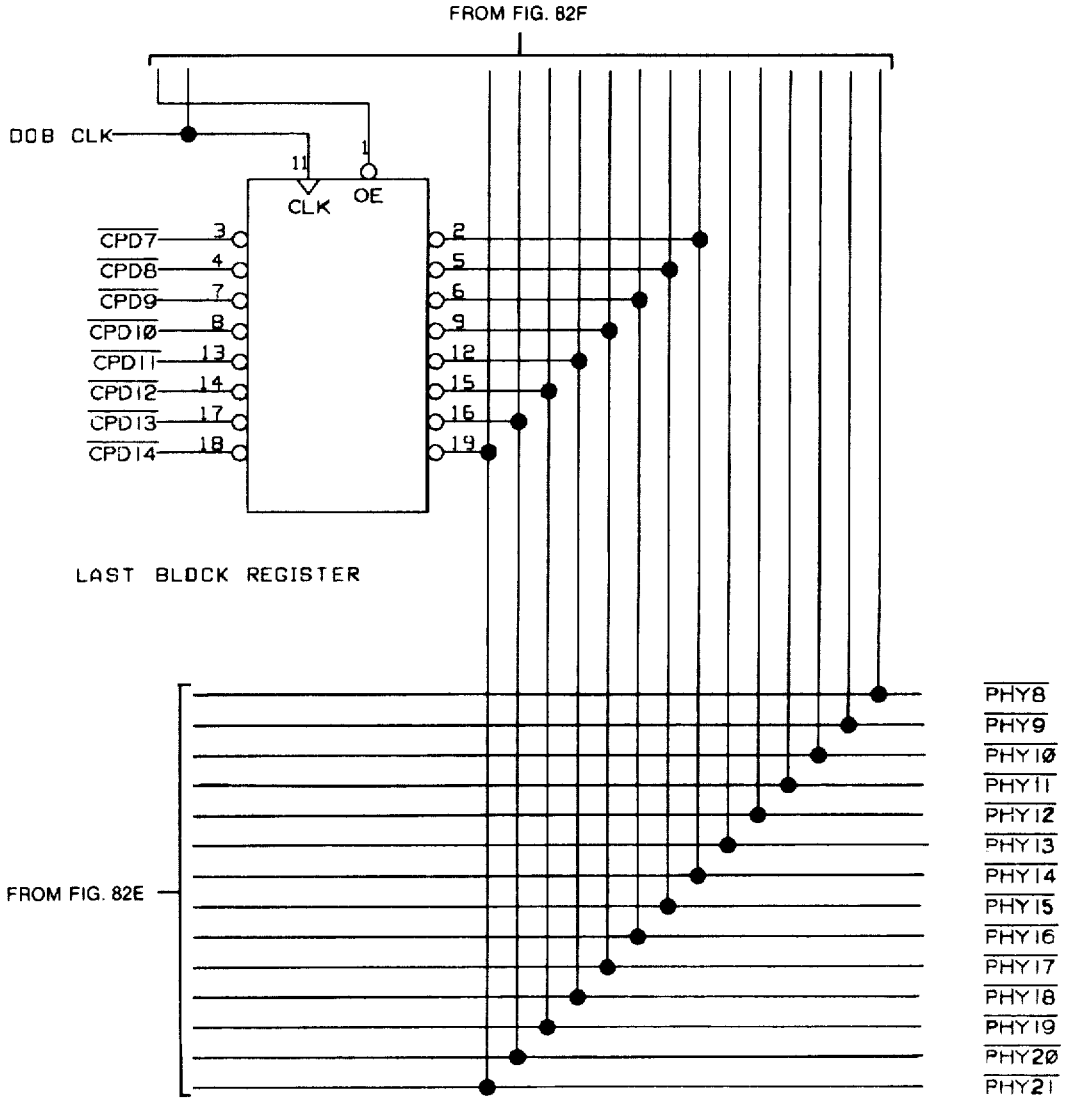
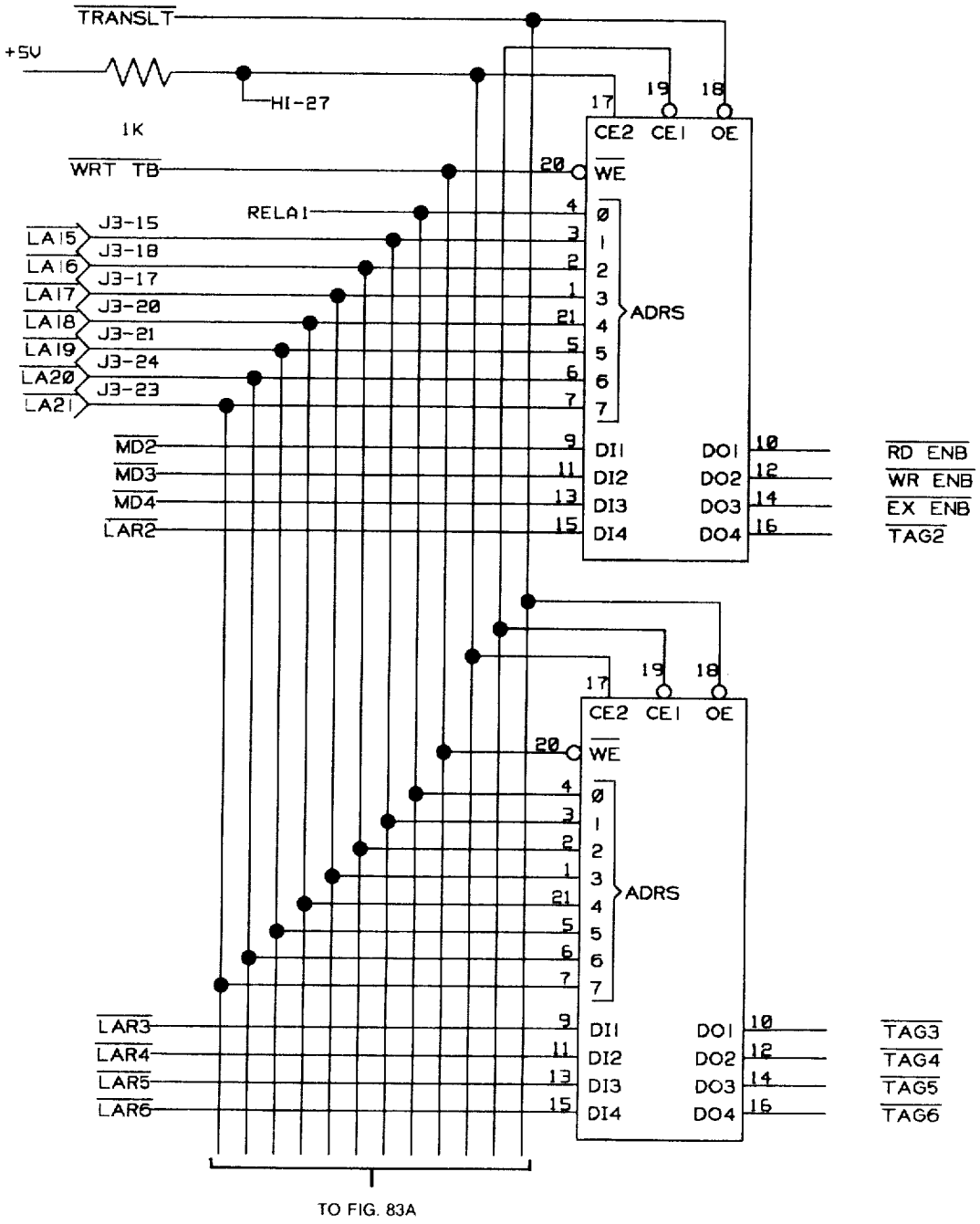

FROM FIG. 82F
DOB CLK
11
1
CLK
OE
CPD7
CPD8
CPD9
CPD10
CPD11
CPD12
CPD13
CPD14
3
4
7
8
13
14
17
18
2
5
6
9
12
15
16
19
LAST BLOCK REGISTER
FROM FIG. 82E
PHY8
PHY9
PHY10
PHY11
PHY12
PHY13
PHY14
PHY15
PHY16
PHY17
PHY18
PHY19
PHY20
PHY21

FIG. 82G



TAG STORE 102 AND PROTECTION STORE 104

FIG. 83

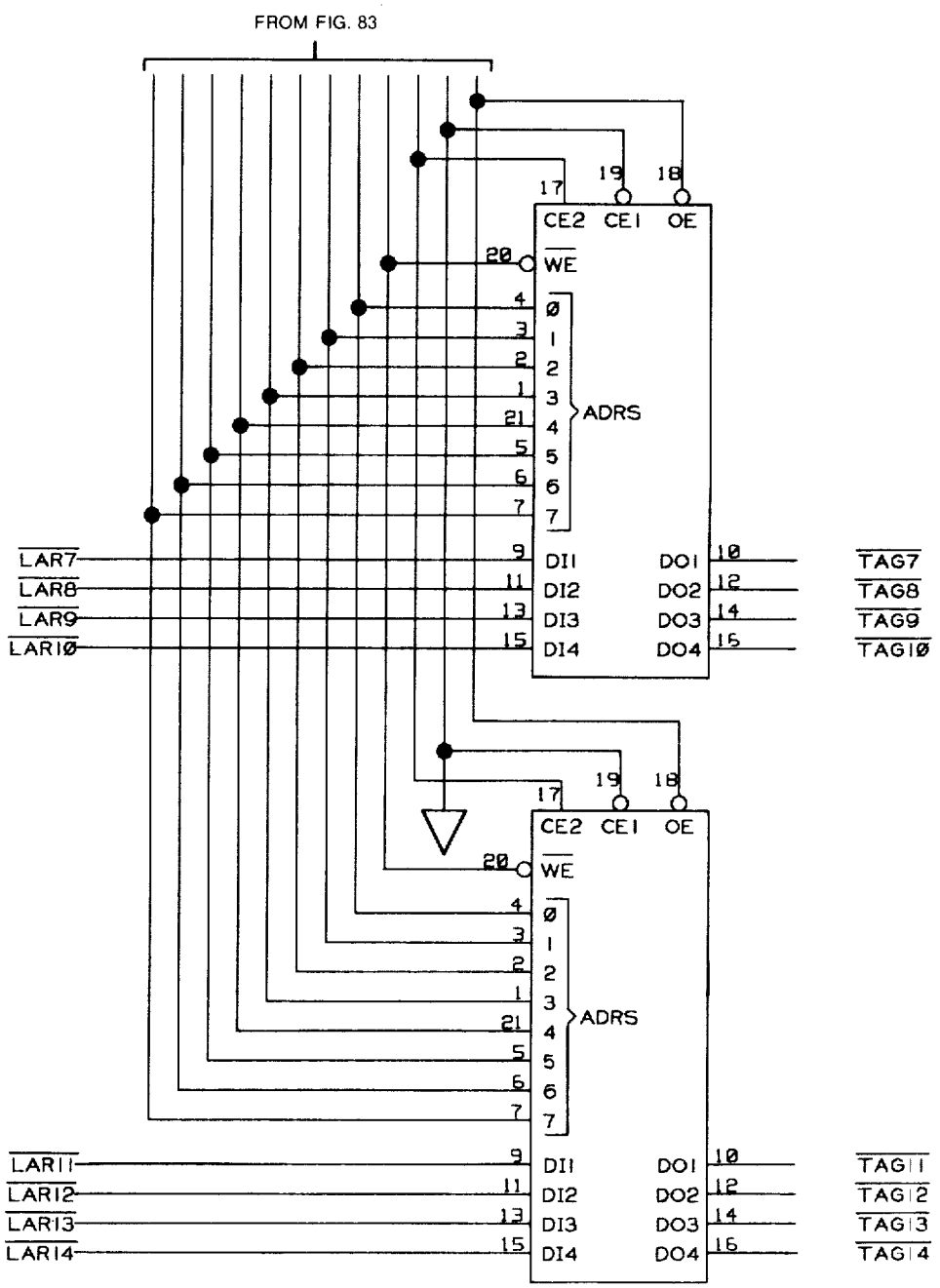
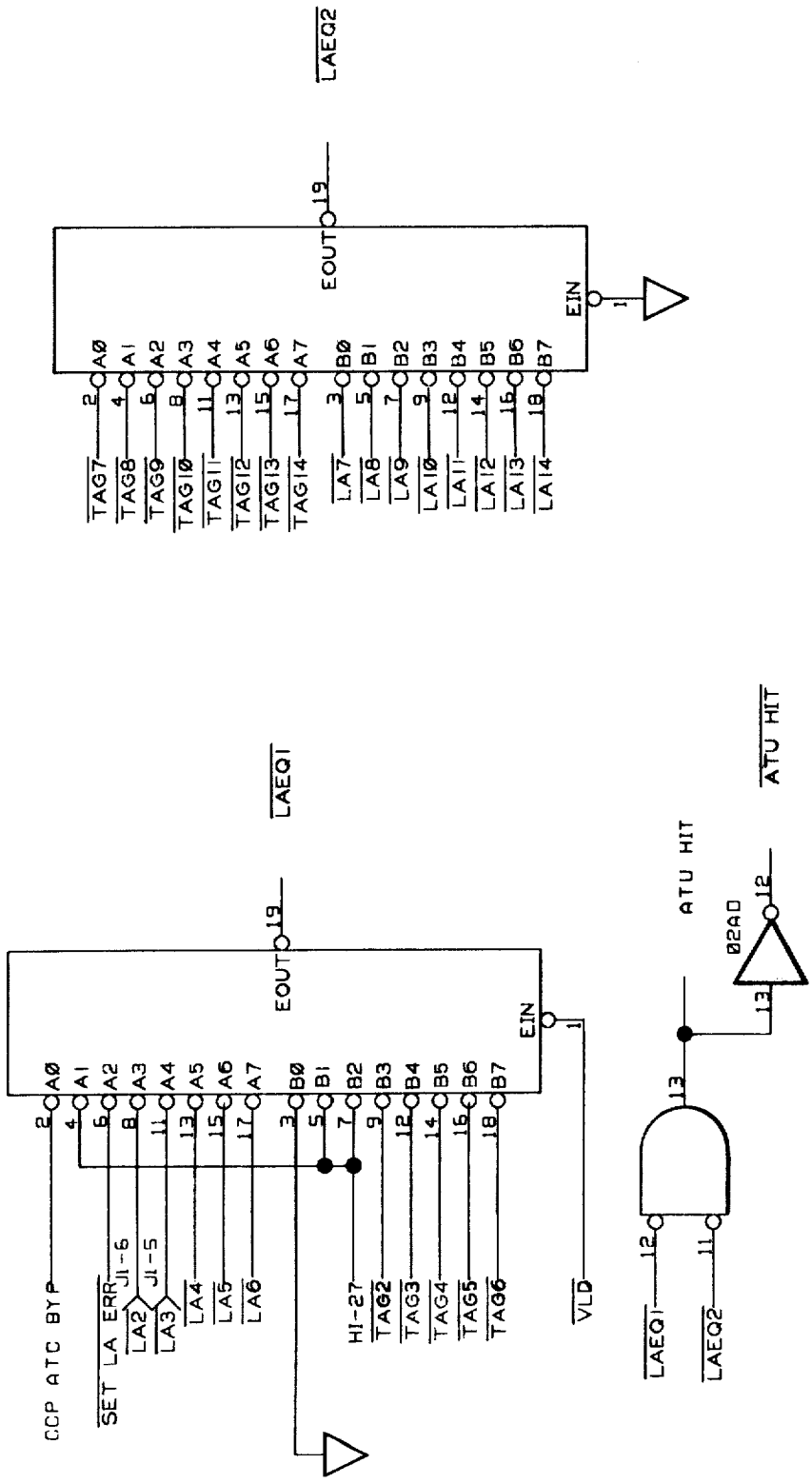
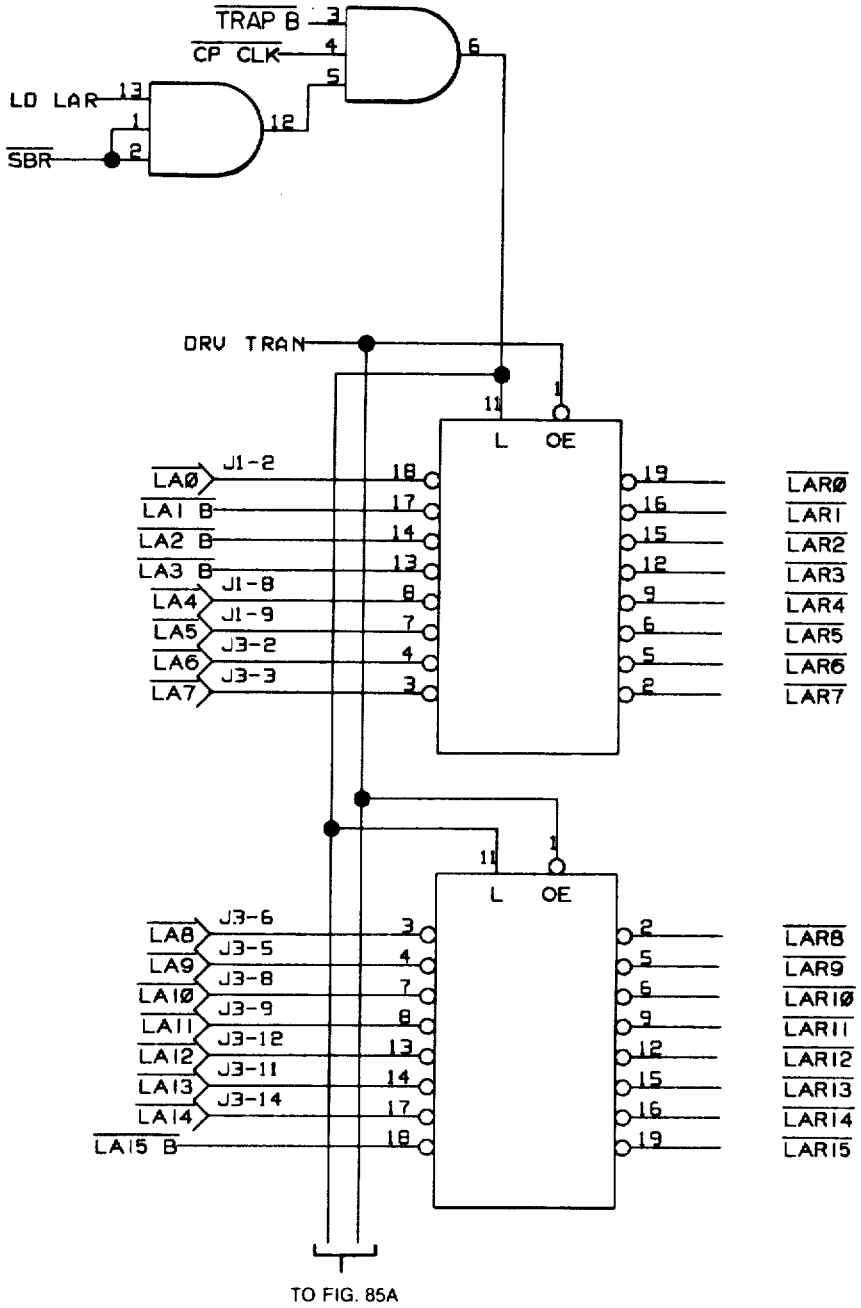

FROM FIG. 83
CE2
CE1
OE
WE
ADRS
DI1
DI2
DI3
DI4
DO1
DO2
DO3
DO4
LAR7
LAR8
LAR9
LAR10
TAG7
TAG8
TAG9
TAG10
LAR11
LAR12
LAR13
LAR14
TAG11
TAG12
TAG13
TAG14

FIG. 83A



TAG COMPARATOR UNIT 105

FIG. 84



LOGICAL ADDRESS REGISTER 101

FIG. 85

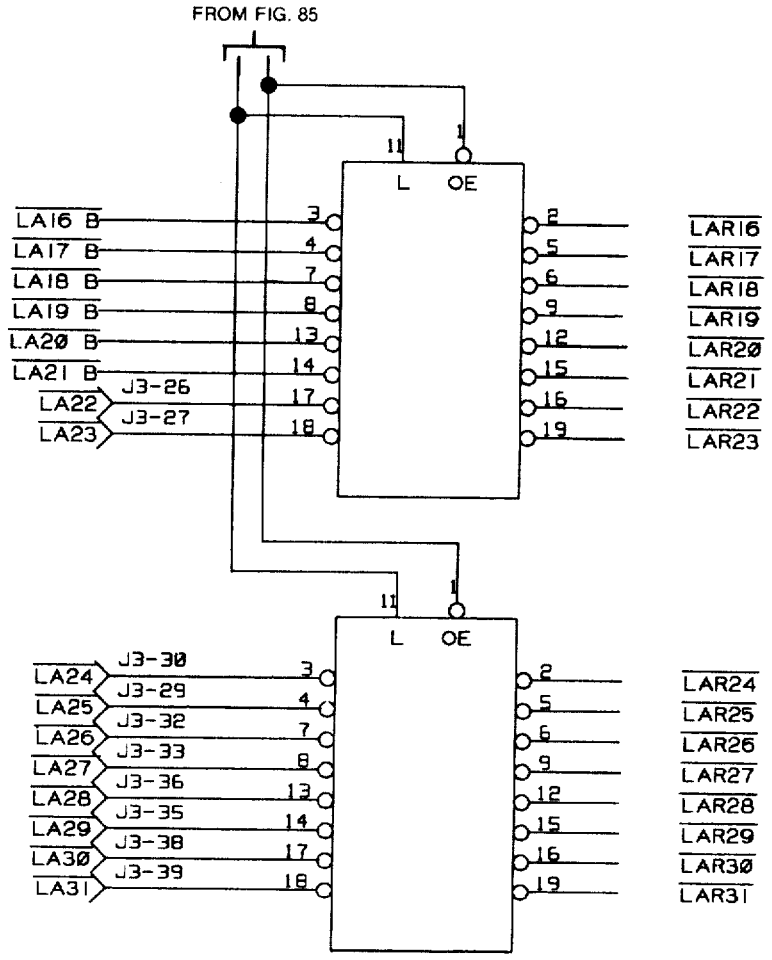
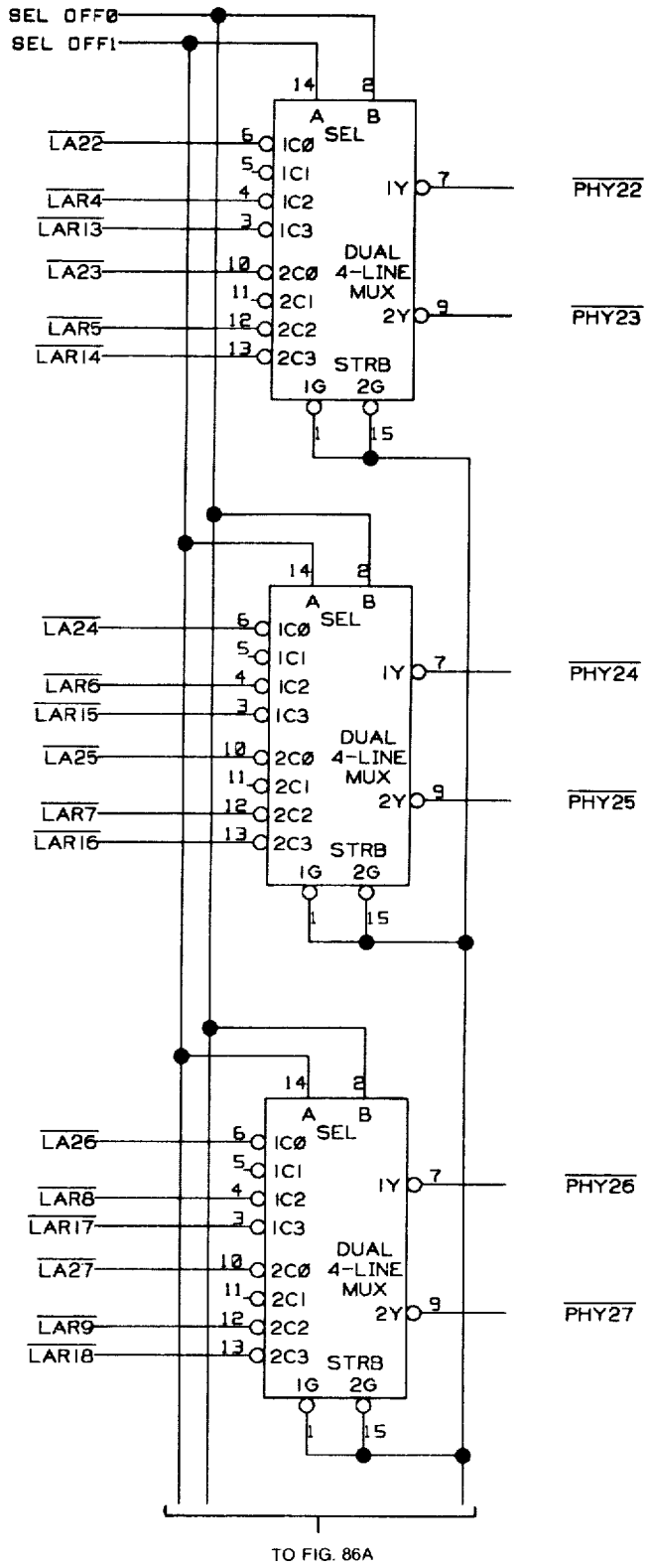

FROM FIG. 85
L
OE
LA16 B
LA17 B
LA18 B
LA19 B
LA20 B
LA21 B
LA22
LA23
J3-26
J3-27
LAR16
LAR17
LAR18
LAR19
LAR20
LAR21
LAR22
LAR23
LA24
LA25
LA26
LA27
LA28
LA29
LA30
LA31
J3-30
J3-29
J3-32
J3-33
J3-36
J3-35
J3-38
J3-39
LAR24
LAR25
LAR26
LAR27
LAR28
LAR29
LAR30
LAR31

FIG. 85A



PHYSICAL ADDRESS OFFSET MULTIPLEXOR 109

FIG. 86

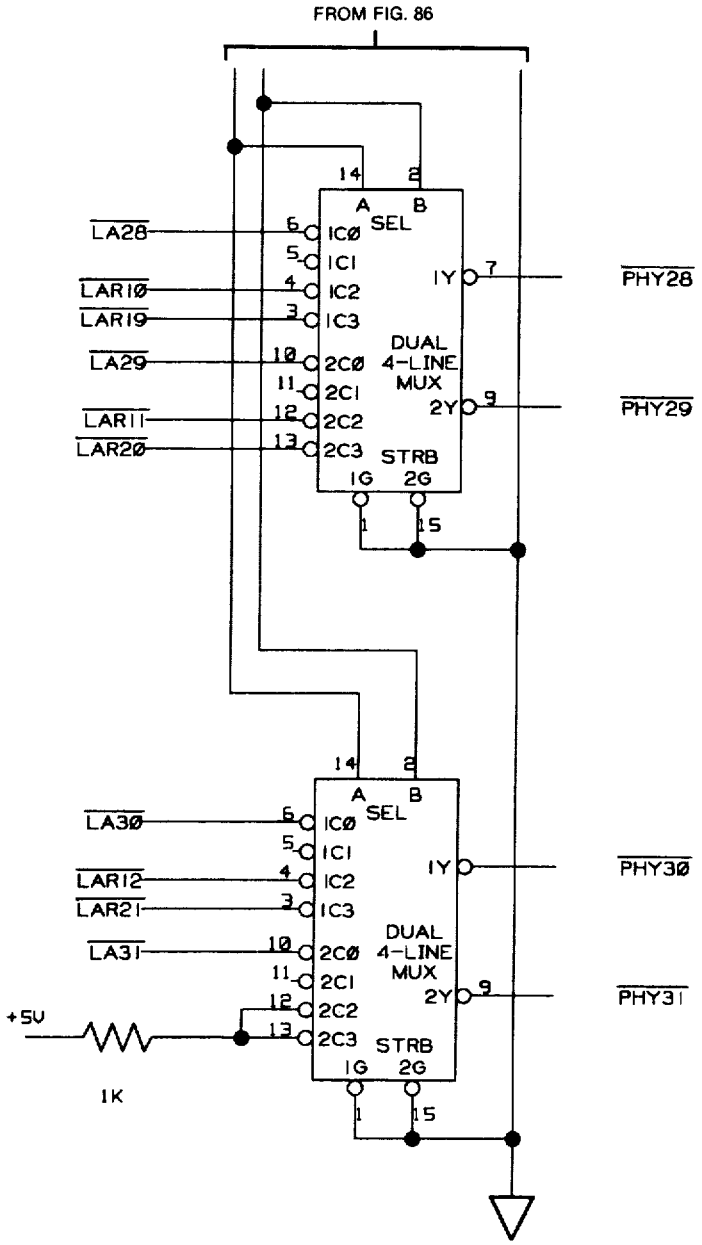
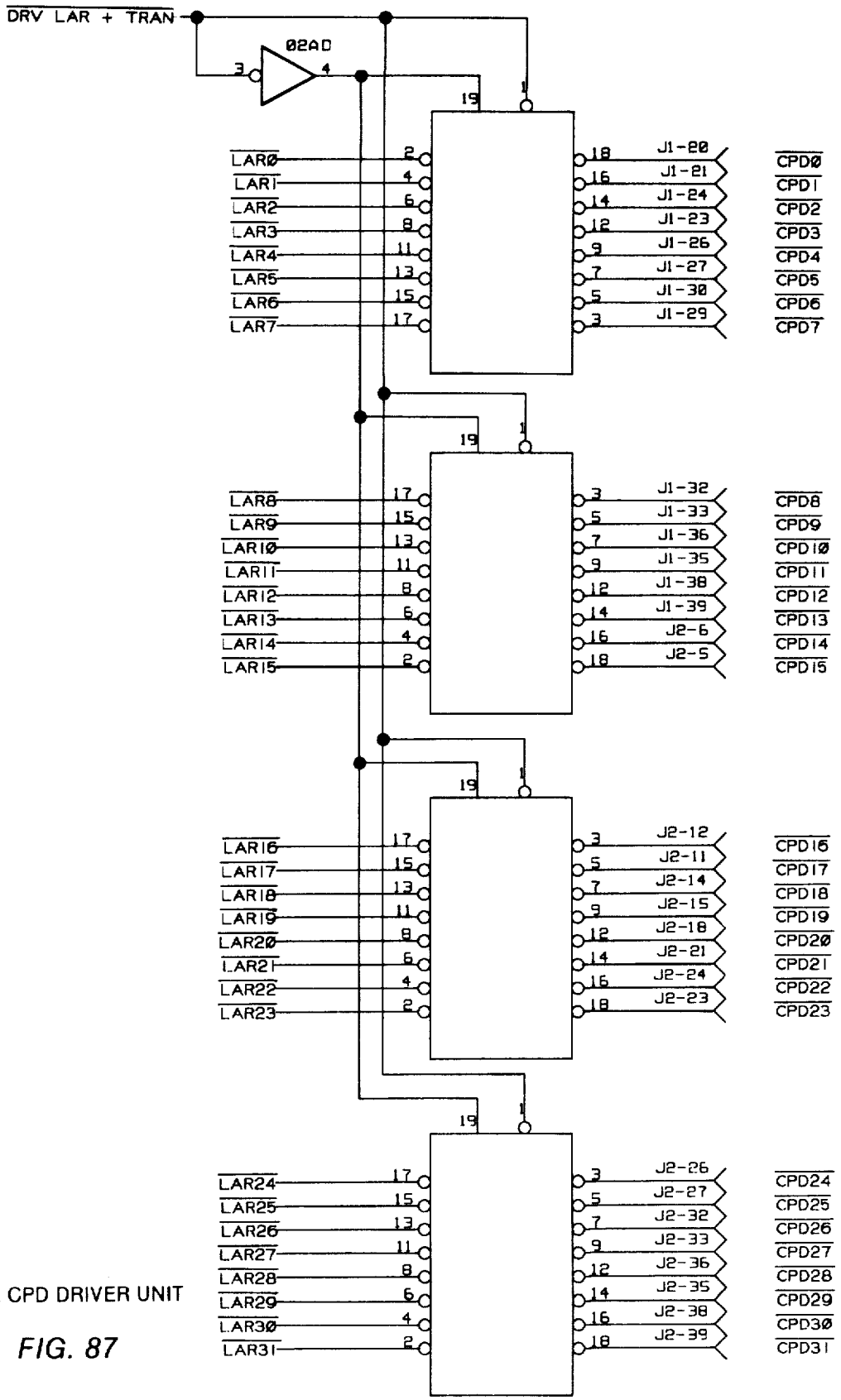
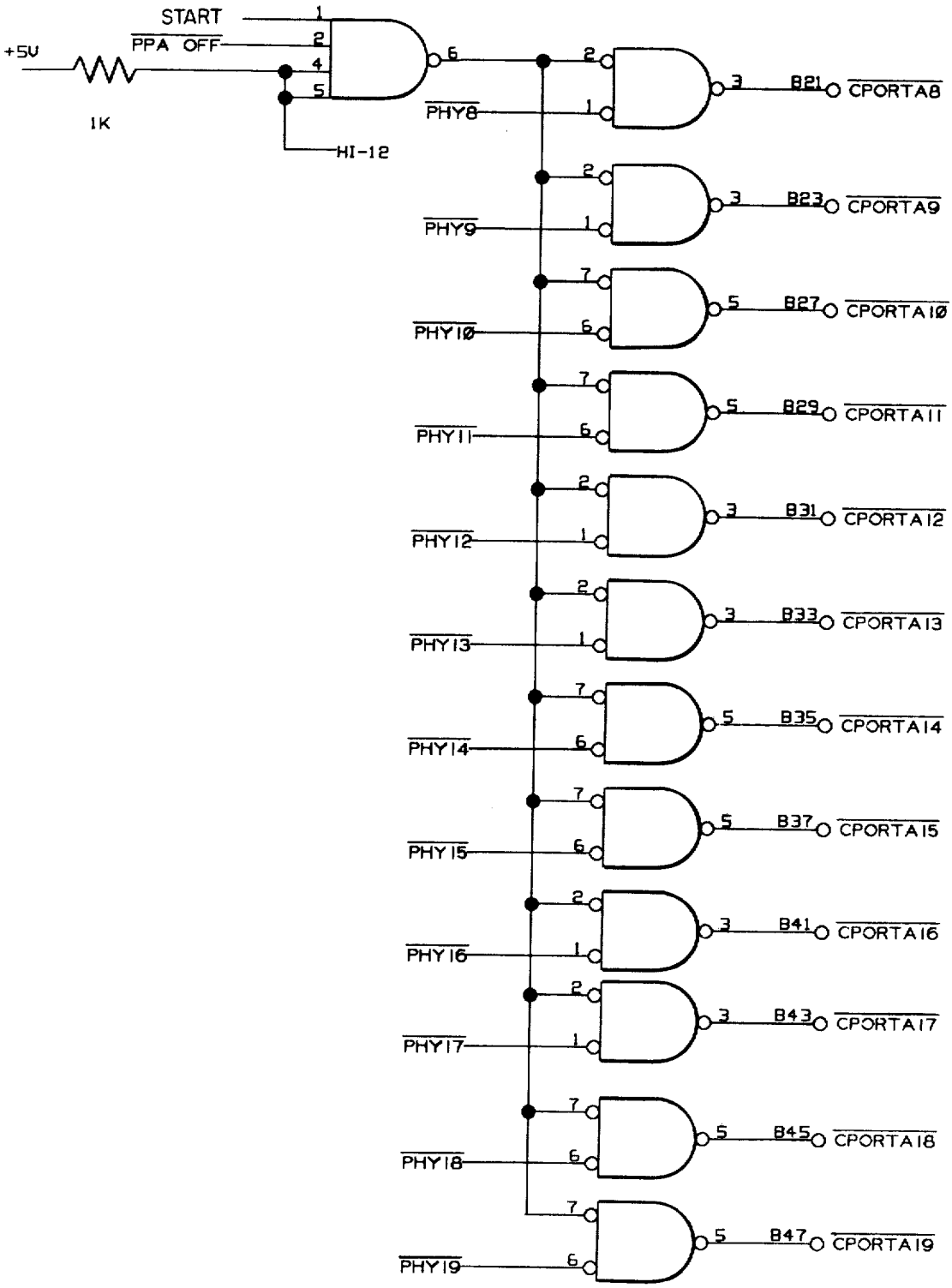

FROM FIG. 86
14
2
A
B
SEL
LA28
6
1C0
5
1C1
LAR10
4
1C2
LAR19
3
1C3
1Y
7
PHY28
LA29
10
2C0
DUAL
4-LINE
MUX
11
2C1
LAR11
12
2C2
2Y
9
PHY29
LAR20
13
2C3
STRB
1G
2G
1
15
14
2
A
B
SEL
LA30
6
1C0
5
1C1
LAR12
4
1C2
LAR21
3
1C3
1Y
PHY30
LA31
10
2C0
DUAL
4-LINE
MUX
11
2C1
12
2C2
2Y
9
PHY31
+5V
13
2C3
STRB
1G
2G
1K
1
15

FIG. 86A



LAR CPD DRIVER UNIT

FIG. 87



PHYSICAL ADDRESS BUS DRIVERS

FIG. 88

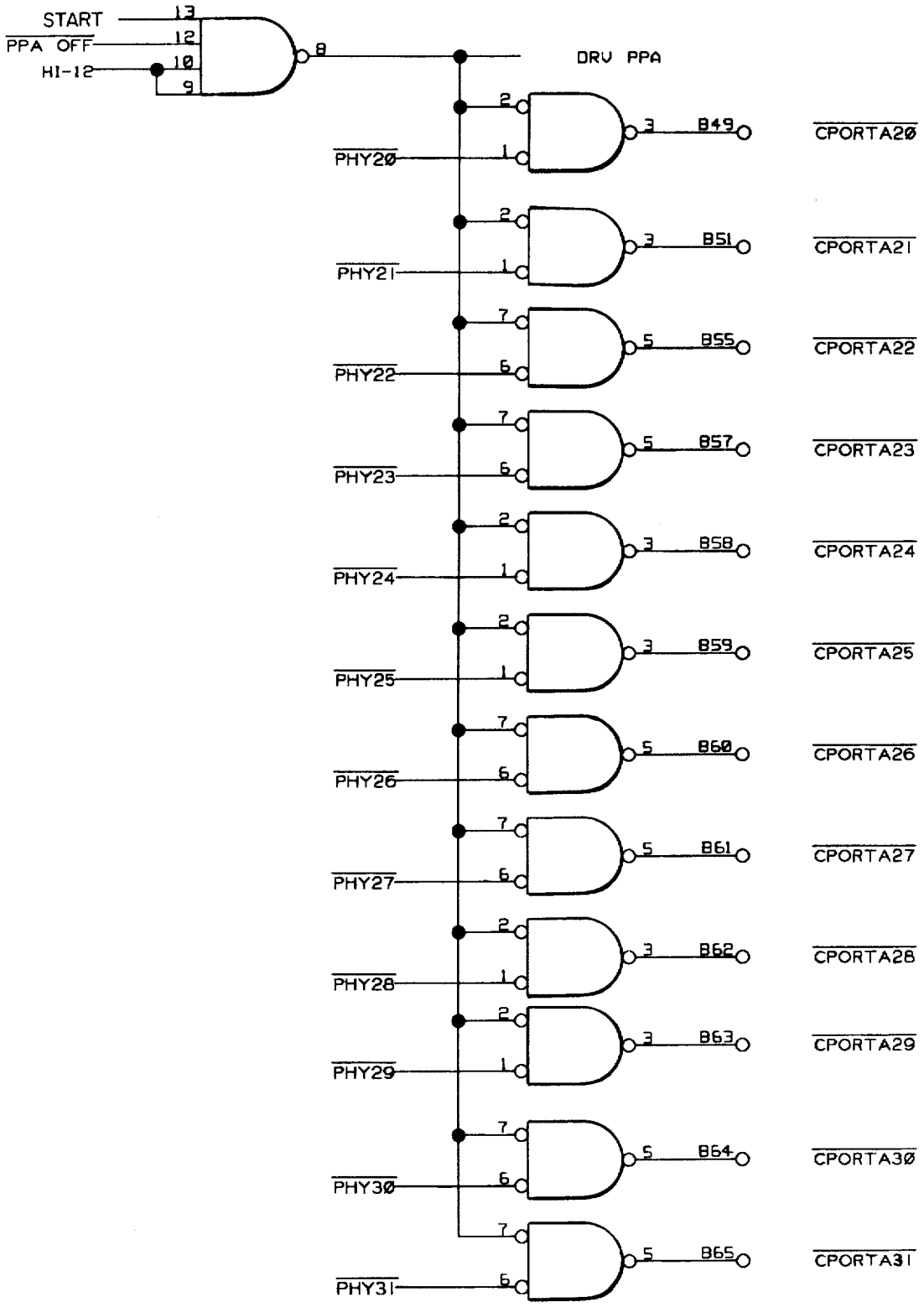

START
PPA OFF
HI-12
13
12
10
9
8
DRV PPA
PHY20
PHY21
PHY22
PHY23
PHY24
PHY25
PHY26
PHY27
PHY28
PHY29
PHY30
PHY31
B49
B51
B55
B57
B58
B59
B60
B61
B62
B63
B64
B65
CPORTA20
CPORTA21
CPORTA22
CPORTA23
CPORTA24
CPORTA25
CPORTA26
CPORTA27
CPORTA28
CPORTA29
CPORTA30
CPORTA31

FIG. 88A

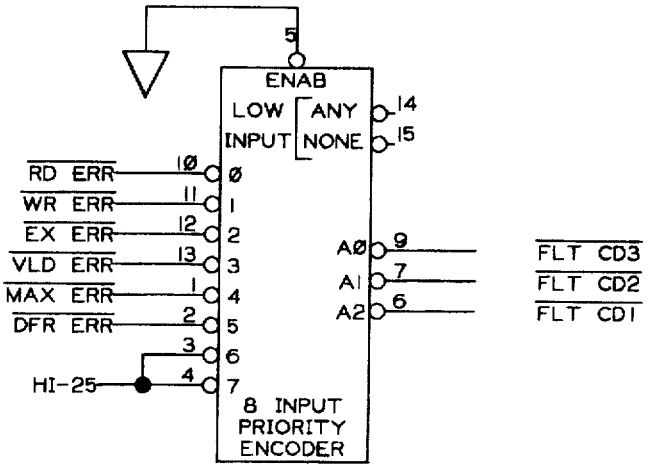
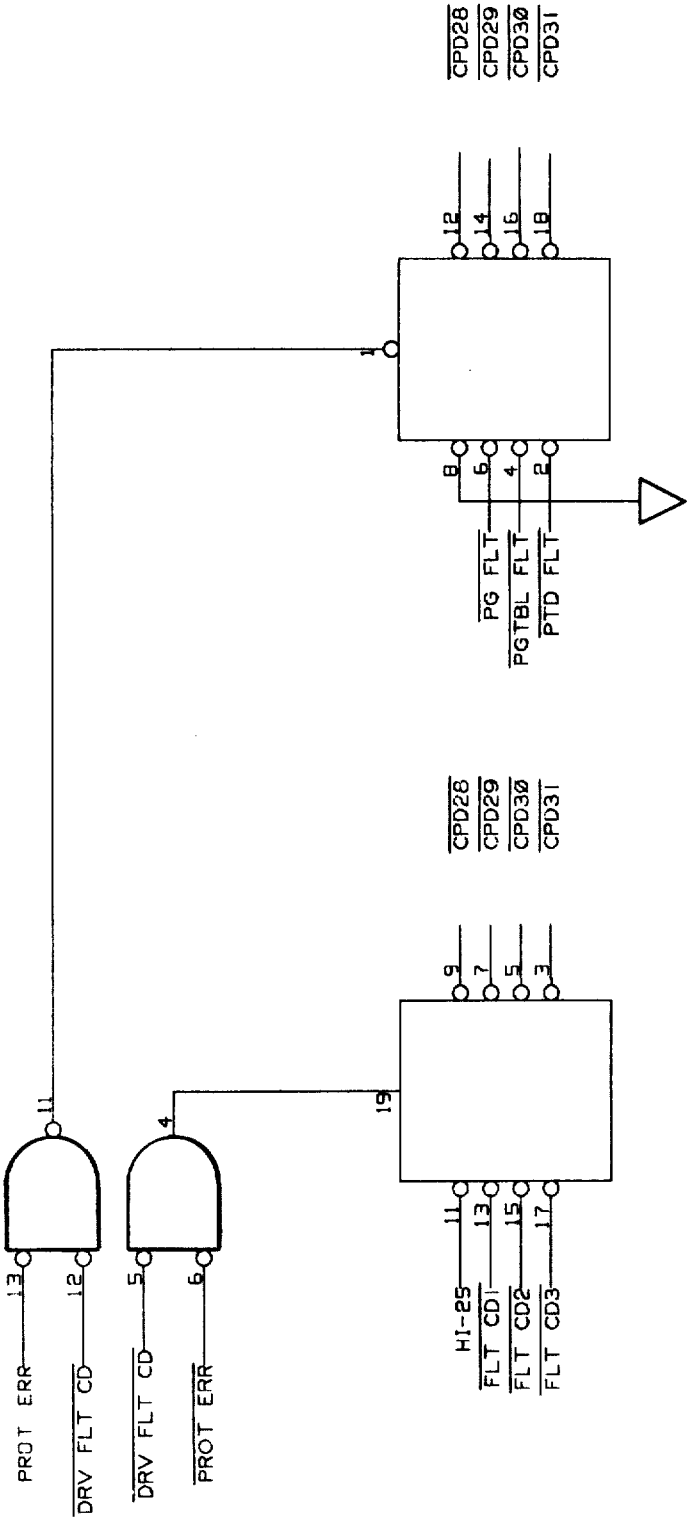
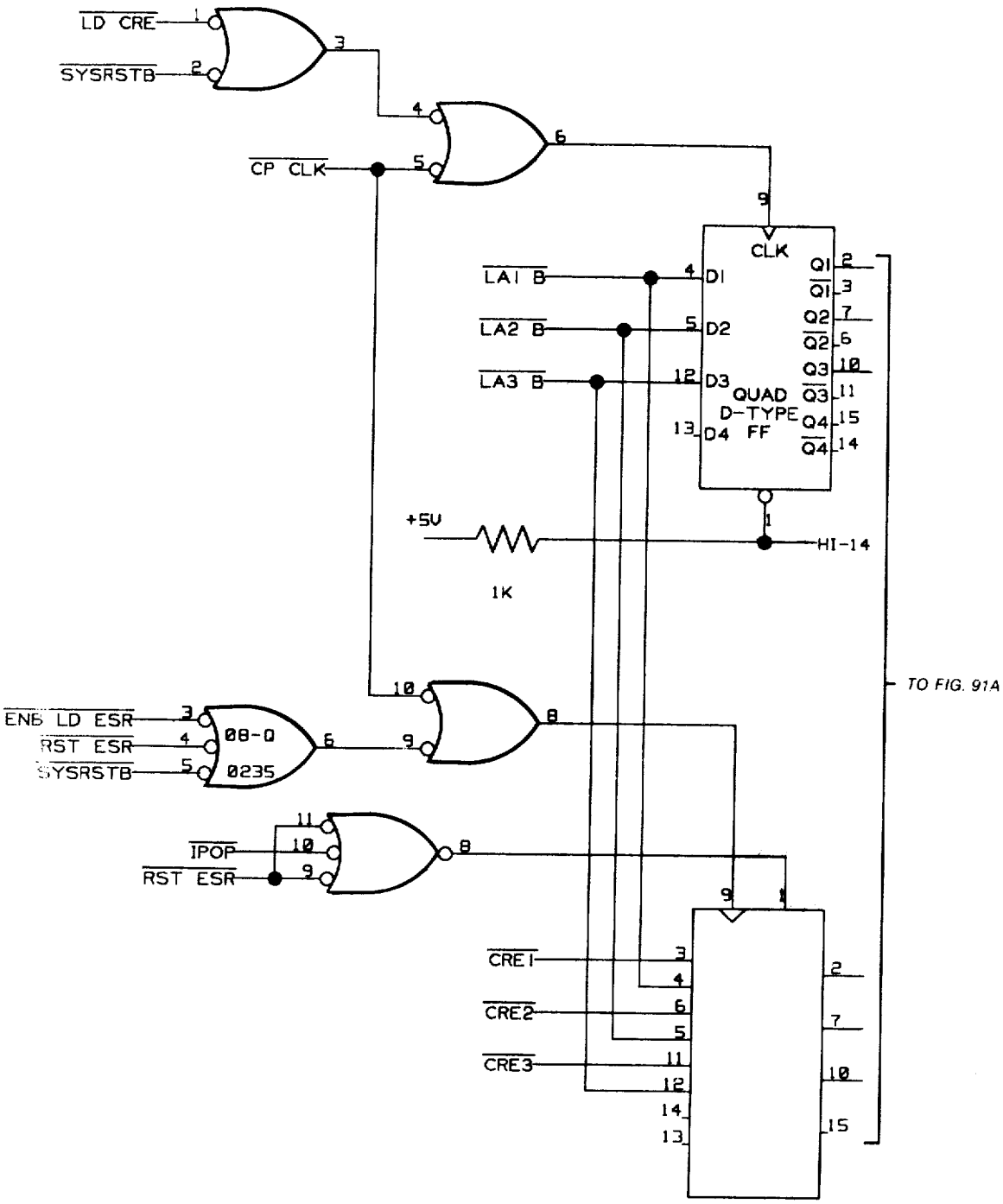

5
ENAB
LOW
INPUT
ANY
NONE
14
15
RD ERR
WR ERR
EX ERR
VLD ERR
MAX ERR
DFR ERR
HI-25
10
11
12
13
1
2
3
4
0
1
2
3
4
5
6
7
A0
A1
A2
9
7
6
FLT CD3
FLT CD2
FLT CD1
8 INPUT
PRIORITY
ENCODER

FIG. 89



FLT CD DRIVERS 112

FIG. 90



RING PROTECTION LOGIC III

FIG. 91

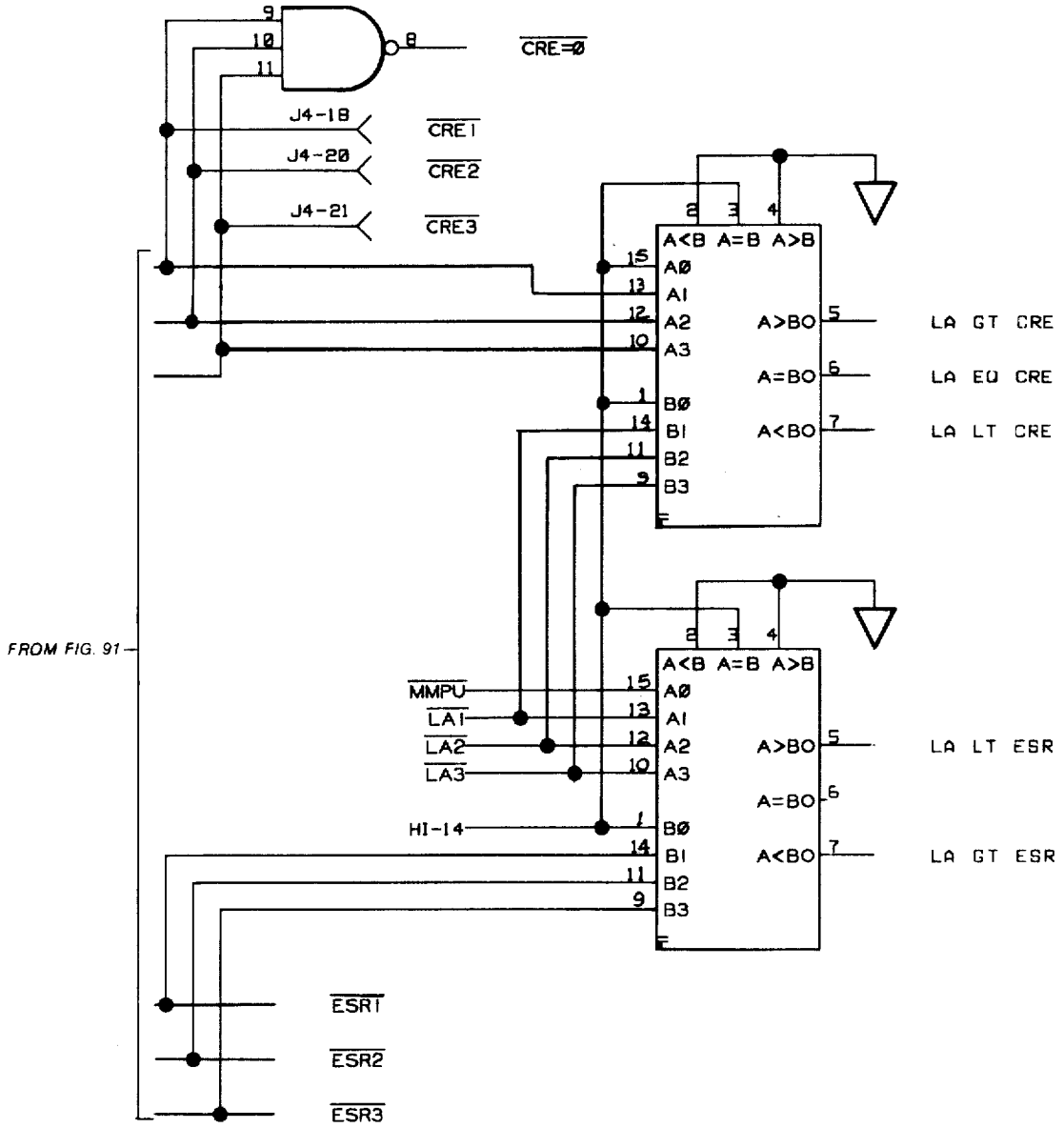

9
10
11
8
CRE=Ø
J4-18
CRE1
J4-20
CRE2
J4-21
CRE3
A<B A=B A>B
AØ
A1
A2
A3
BØ
B1
B2
B3
A>BO
A=BO
A<BO
LA GT CRE
LA EQ CRE
LA LT CRE
FROM FIG. 91
MMPU
LA1
LA2
LA3
HI-14
LA LT ESR
LA GT ESR
ESR1
ESR2
ESR3

FIG. 91A

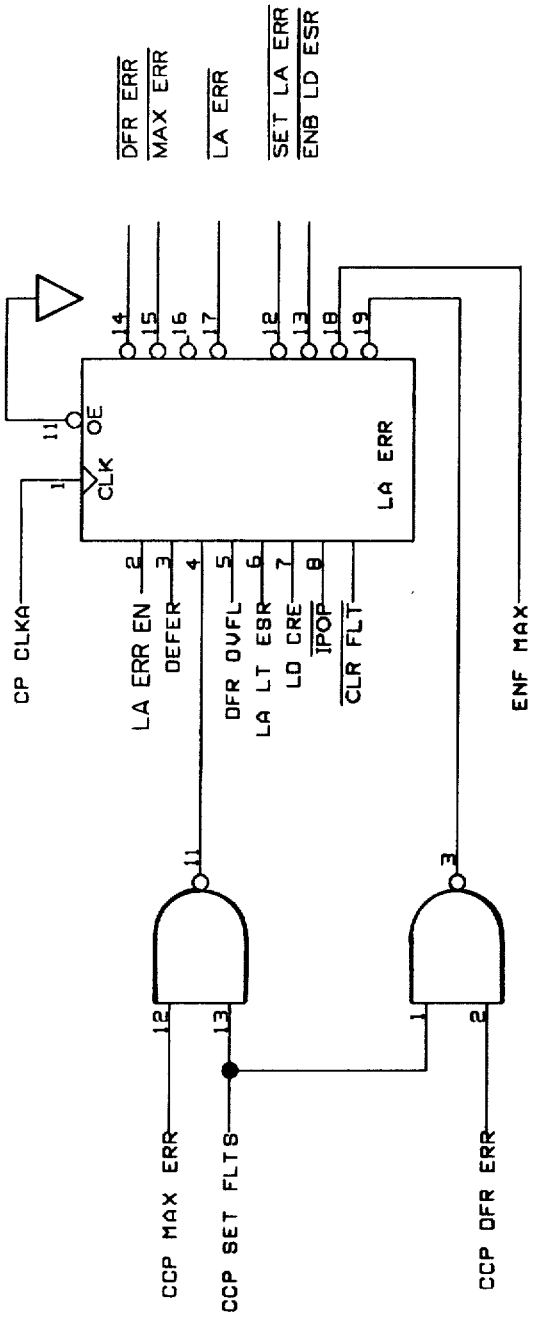
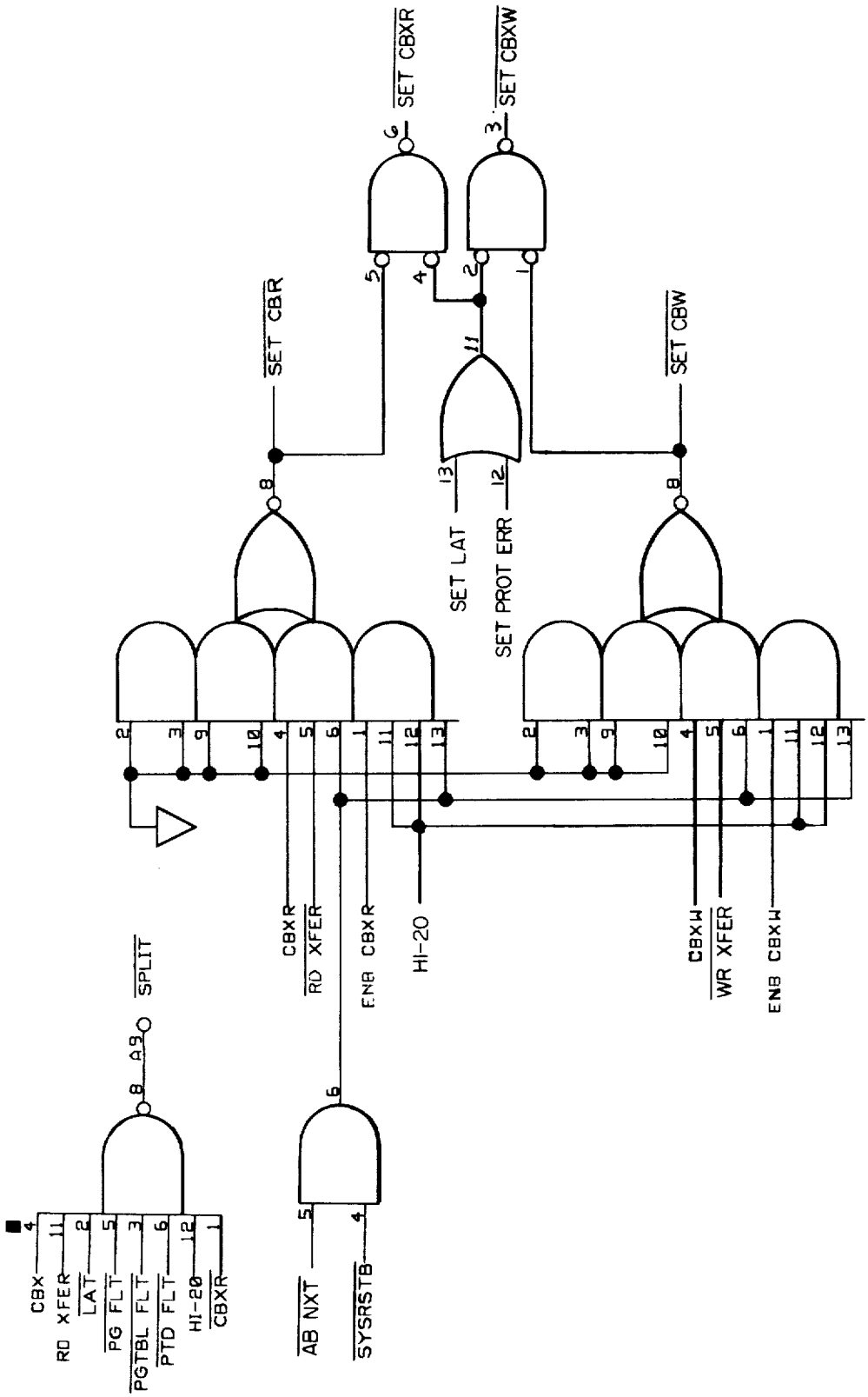

DFR ERR
MAX ERR
LA ERR
SET LA ERR
ENB LD ESR
14
15
16
17
12
13
18
19
11
OE
CLK
1
LA ERR
CP CLKA
2
3
4
5
6
7
8
LA ERR EN
DEFER
DFR OVFL
LA LT ESR
LD CRE
IPOP
CLR FLT
ENF MAX
11
3
12
13
1
2
CCP MAX ERR
CCP SET FLTS
CCP DFR ERR

FIG. 91B



FAULT DETECTION AND CACHE BLOCK CROSSING TRAP LOGIC

FIG. 92

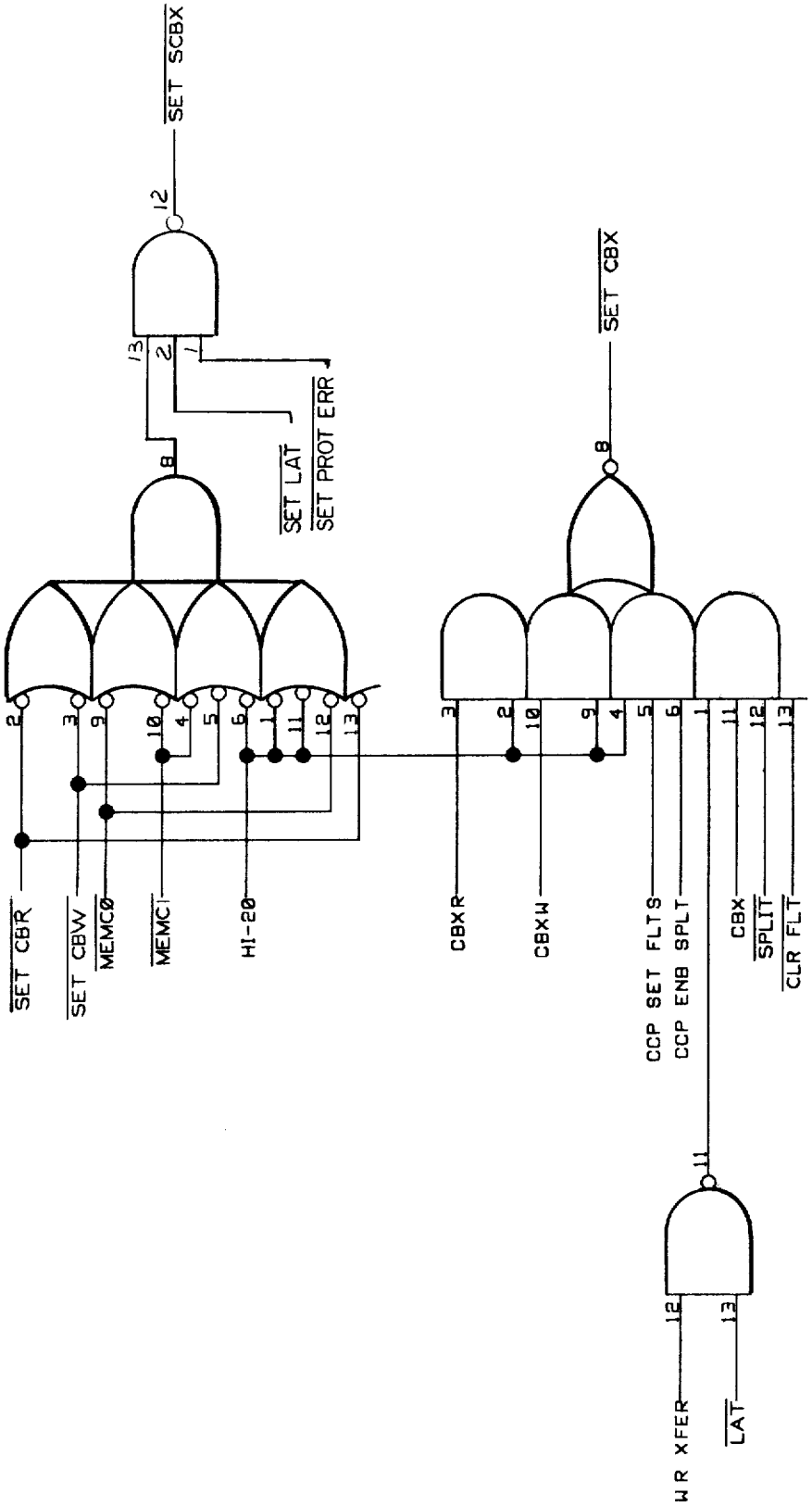

SET SCBX
SET CBX
SET LAT
SET PROT ERR
SET CBR
SET CBW
MEMC0
MEMC1
HI-20
CBXR
CBXW
CCP SET FLTS
CCP ENB SPLT
CBX
SPLIT
CLR FLT
WR XFER
LAT

FIG. 92A

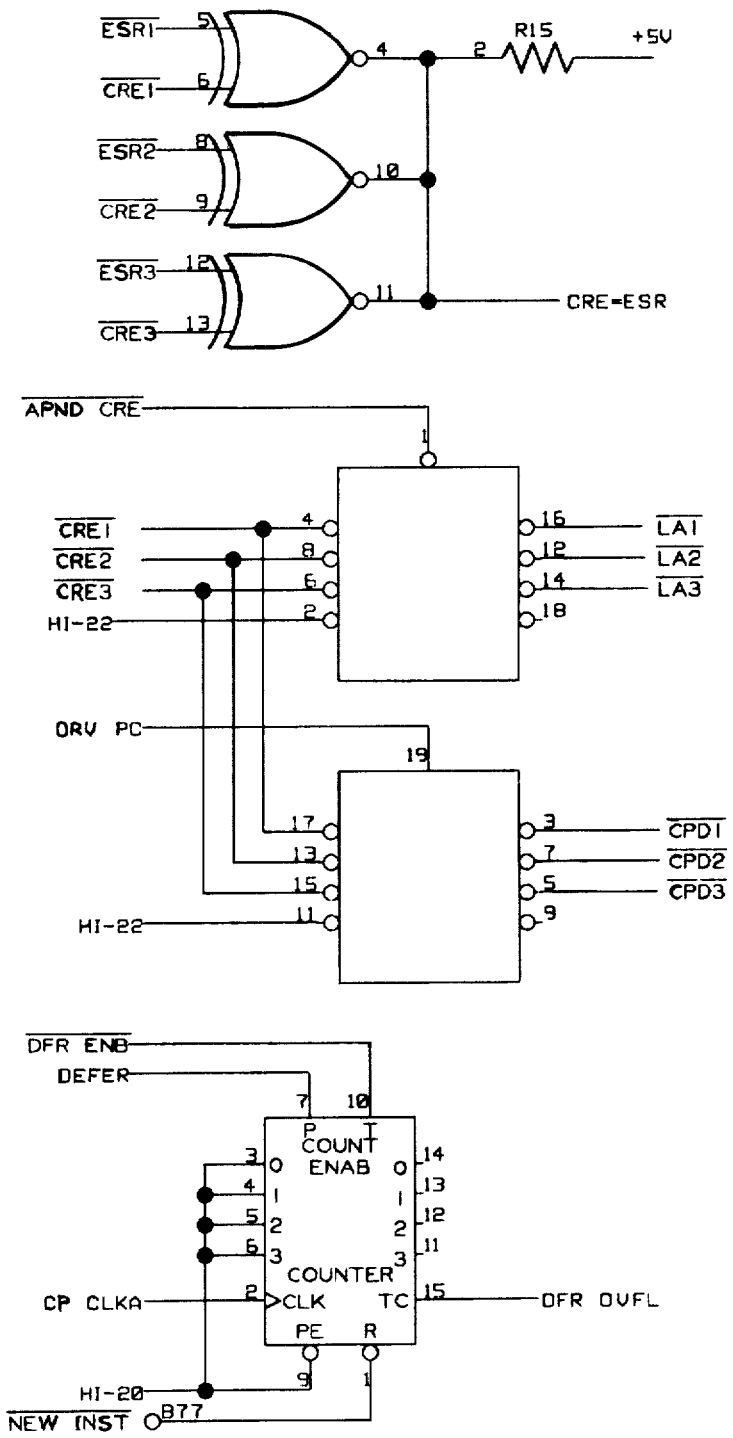

ESR1
CRE1
ESR2
CRE2
ESR3
CRE3
R15
+5V
CRE=ESR
APND CRE
CRE1
CRE2
CRE3
HI-22
LA1
LA2
LA3
ORV PC
HI-22
CPD1
CPD2
CPD3
DFR ENB
DEFER
P
T
COUNT
ENAB
COUNTER
CP CLKA
CLK
TC
PE
R
DFR OVFL
HI-20
NEW INST
B77

FIG. 92B

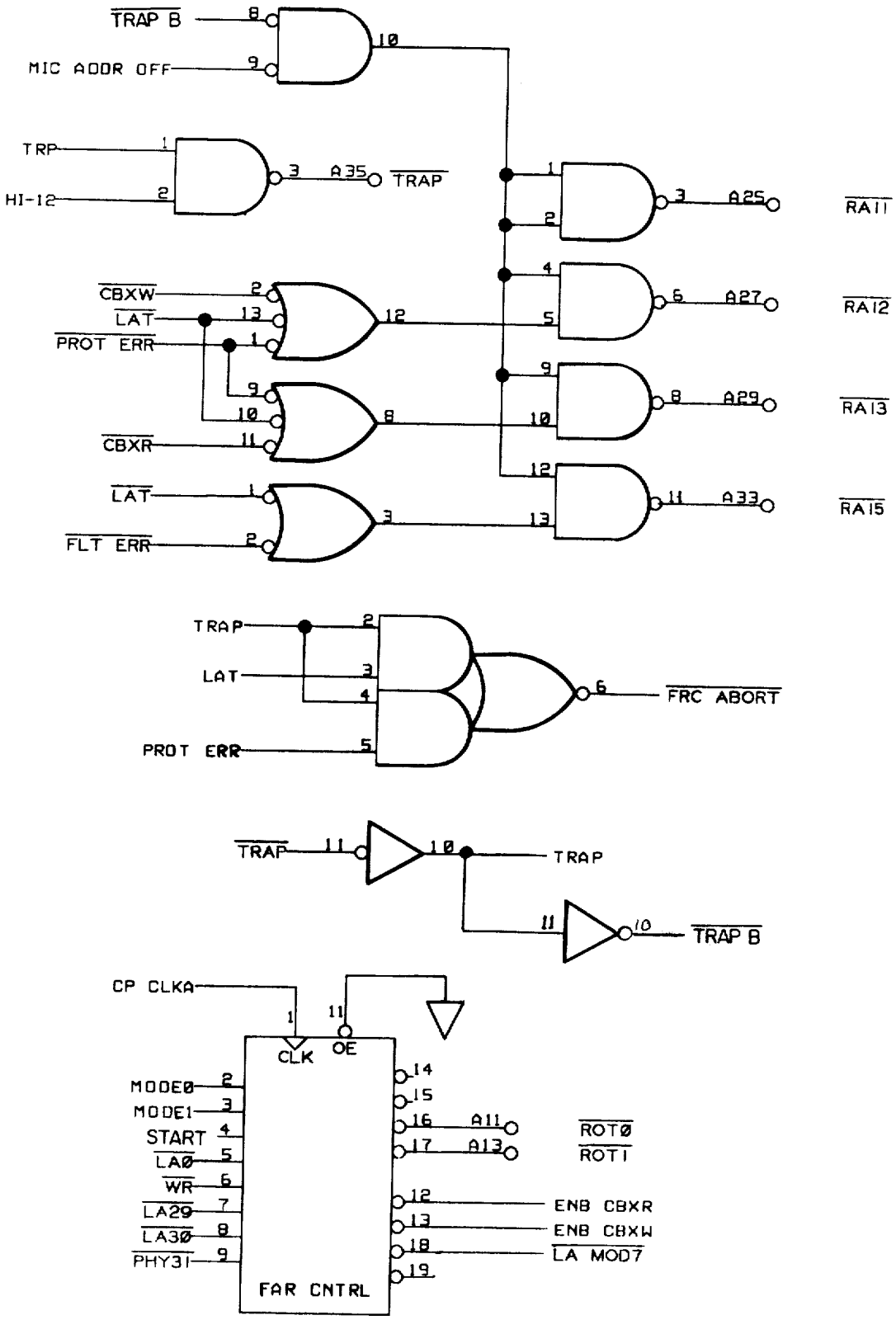

TRAP B
MIC ADDR OFF
TRP
HI-12
A35
TRAP
CBXW
LAT
PROT ERR
CBXR
LAT
FLT ERR
A25
RA11
A27
RA12
A29
RA13
A33
RA15
TRAP
LAT
PROT ERR
FRC ABORT
TRAP
TRAP
TRAP B
CP CLKA
CLK
OE
MODE0
MODE1
START
LA0
WR
LA29
LA30
PHY31
FAR CNTRL
A11
A13
ROT0
ROT1
ENB CBXR
ENB CBXW
LA MOD7

FIG. 92C

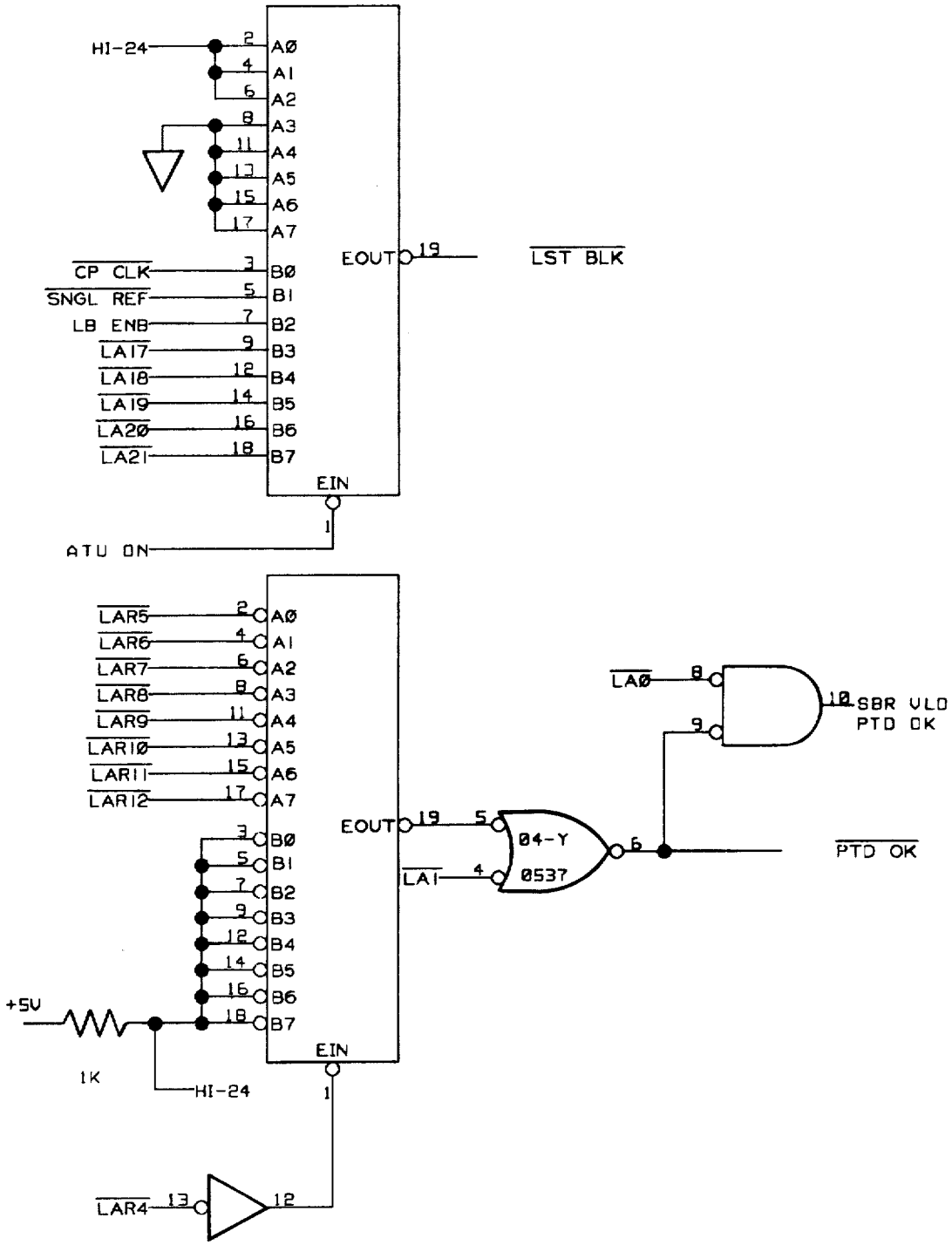
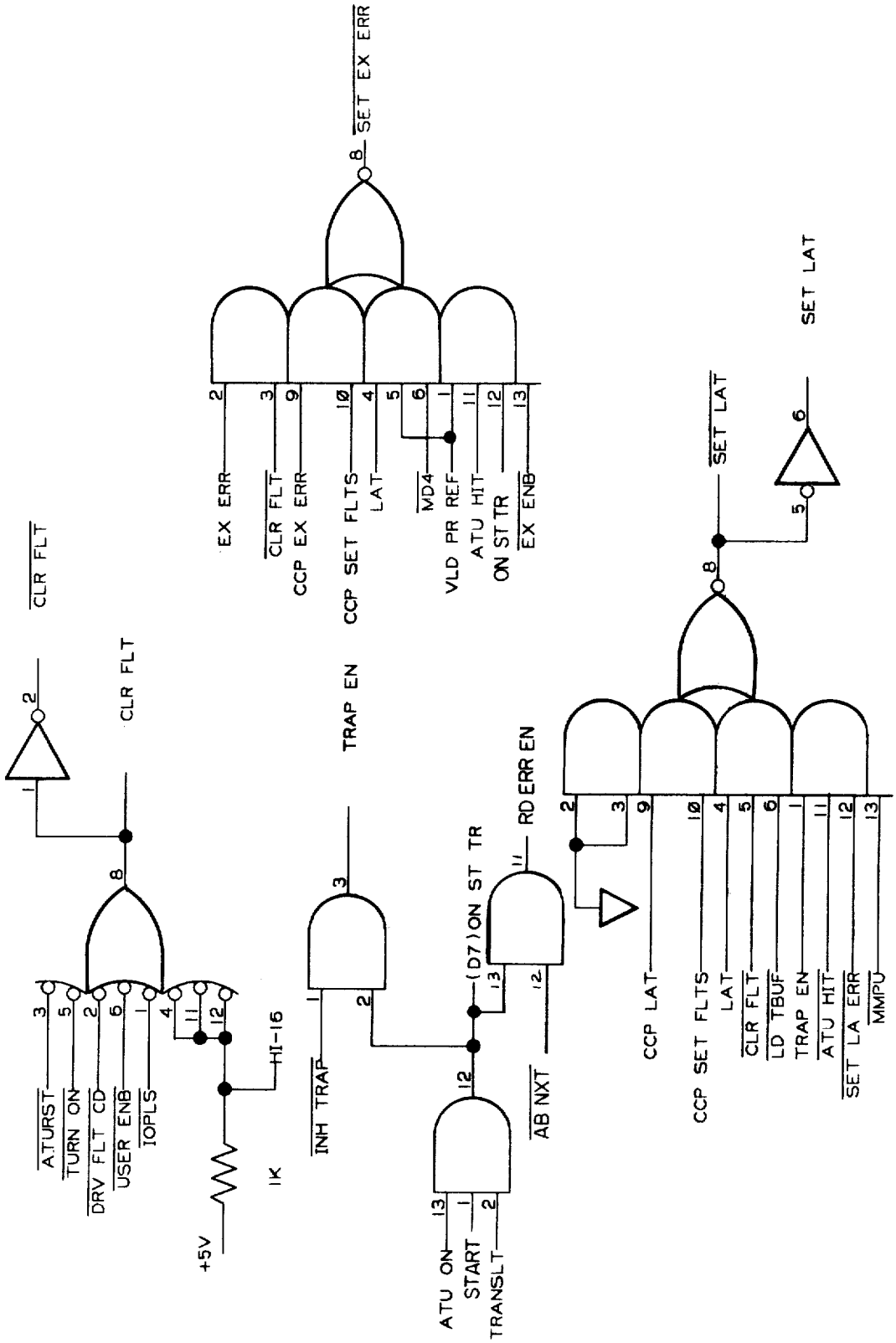

HI-24
A0
A1
A2
A3
A4
A5
A6
A7
CP CLK
SNGL REF
LB ENB
LA17
LA18
LA19
LA20
LA21
B0
B1
B2
B3
B4
B5
B6
B7
EOUT
LST BLK
EIN
ATU ON
LAR5
LAR6
LAR7
LAR8
LAR9
LAR10
LAR11
LAR12
LA0
SBR VLD
PTD OK
04-Y
0537
LA1
PTD OK
+5V
1K
HI-24
LAR4

FIG. 92D



FAULT DETECTION TRAP LOGIC

FIG. 93

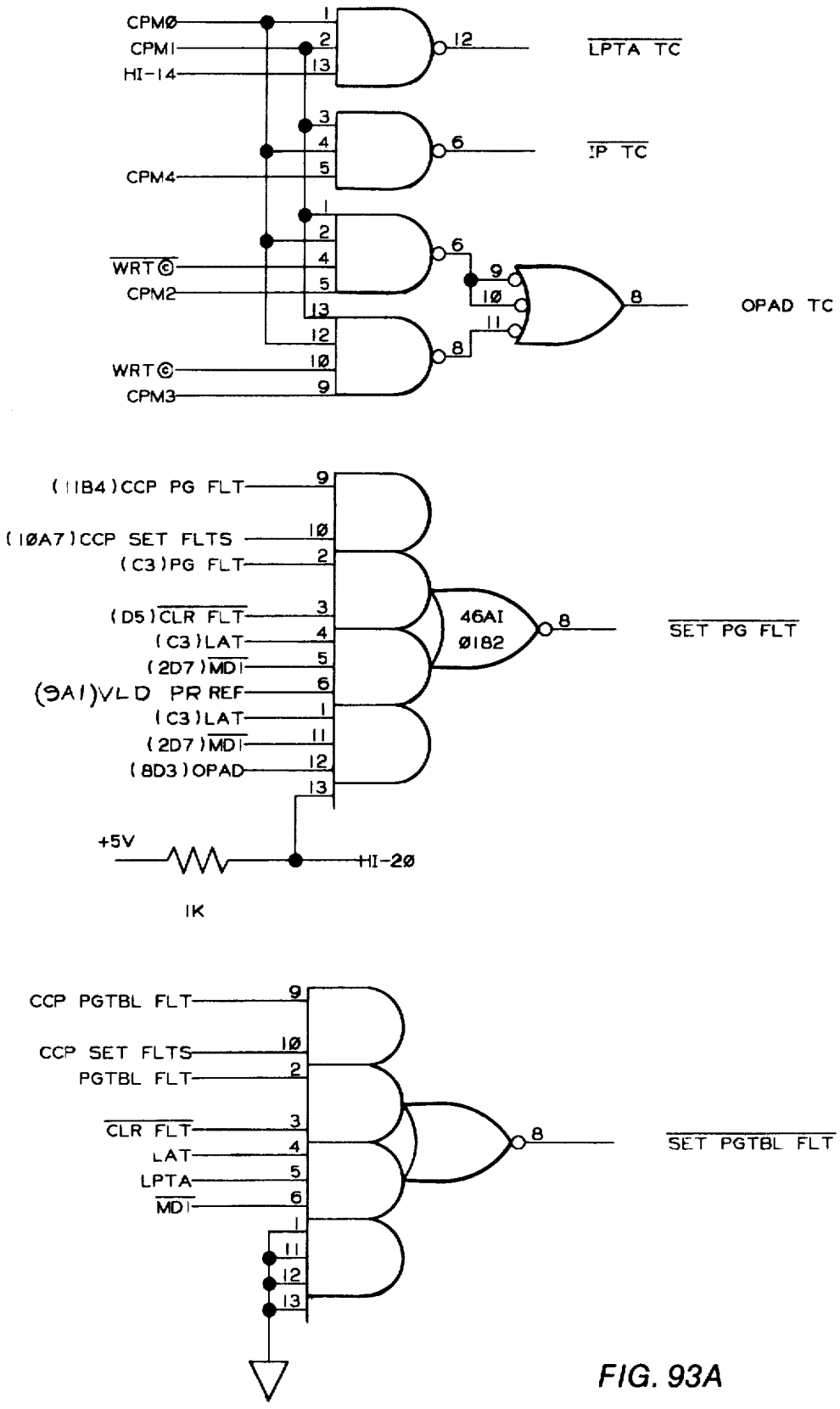

CPMØ
CPMI
HI-14
CPM4
WRT©
CPM2
WRT©
CPM3
LPTA TC
IP TC
OPAD TC
(11B4)CCP PG FLT
(1ØA7)CCP SET FLTS
(C3)PG FLT
(D5)CLR FLT
(C3)LAT
(2D7)MDI
(9AI)VLD PR REF
(C3)LAT
(2D7)MDI
(8D3)OPAD
46AI
Ø182
SET PG FLT
+5V
HI-2Ø
IK
CCP PGTBL FLT
CCP SET FLTS
PGTBL FLT
CLR FLT
LAT
LPTA
MDI
SET PGTBL FLT

FIG. 93A

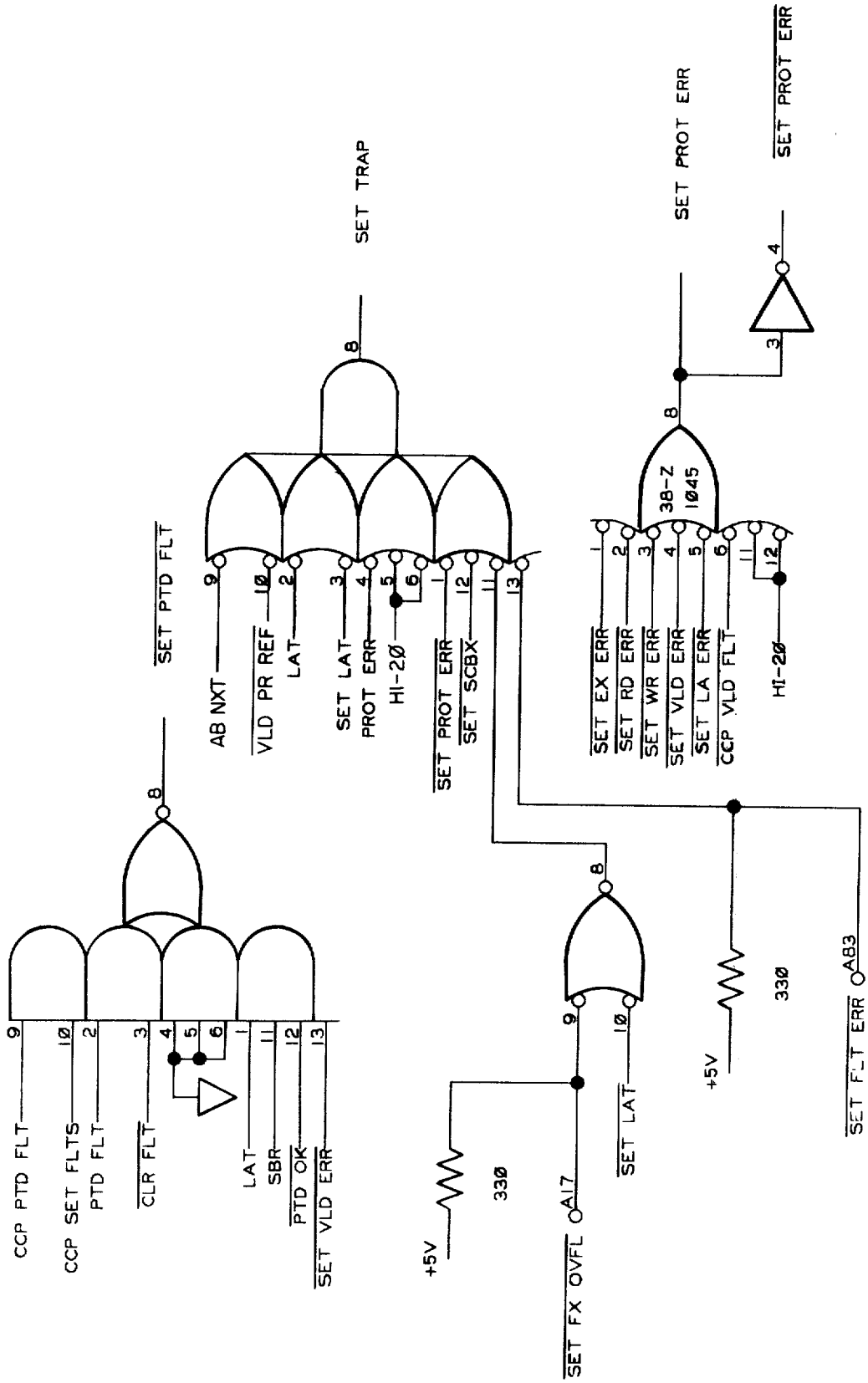

CCP PTD FLT
CCP SET FLTS
PTD FLT
CLR FLT
LAT
SBR
PTD OK
SET VLD ERR
SET PTD FLT
AB NXT
VLD PR REF
LAT
SET LAT
PROT ERR
HI-2Ø
SET PROT ERR
SET SCBX
SET TRAP
+5V
33Ø
A17
SET FX OVFL
SET LAT
SET EX ERR
SET RD ERR
SET WR ERR
SET VLD ERR
SET LA ERR
CCP VLD FLT
HI-2Ø
38-Z
1Ø45
SET PROT ERR
SET PROT ERR
+5V
33Ø
A83
SET FLT ERR
SET RD ERR

FIG. 93B

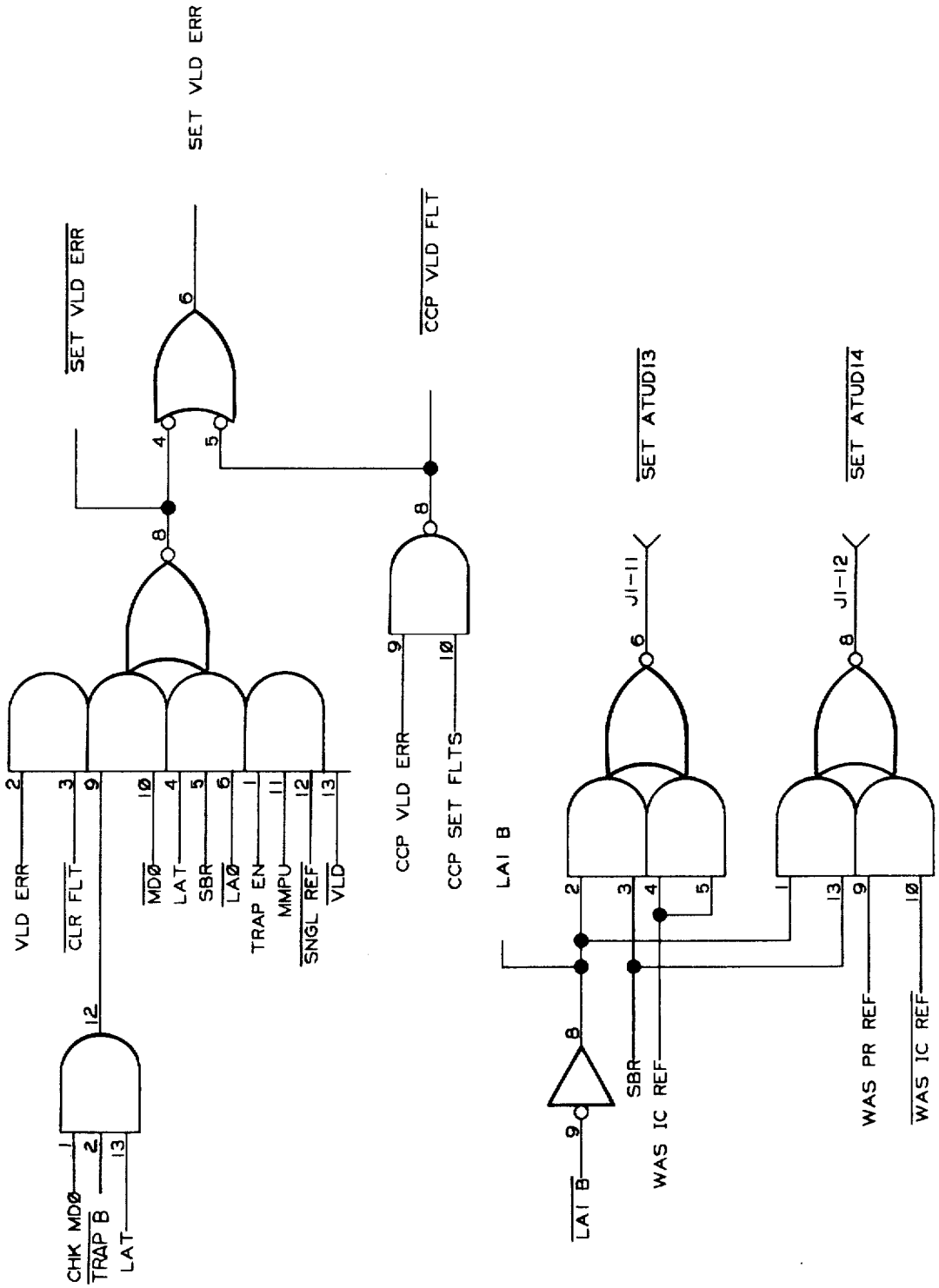

SET VLD ERR
SET VLD ERR
CCP VLD FLT
SET ATUD13
SET ATUD14
VLD ERR
CLR FLT
MDØ
LAT
SBR
LAØ
TRAP EN
MMPU
SNGL REF
VLD
CCP VLD ERR
CCP SET FLTS
LAI B
CHK MDØ
TRAP B
LAT
LAI B
SBR
WAS IC REF
J1-11
J1-12
WAS PR REF
WAS IC REF

FIG. 93C

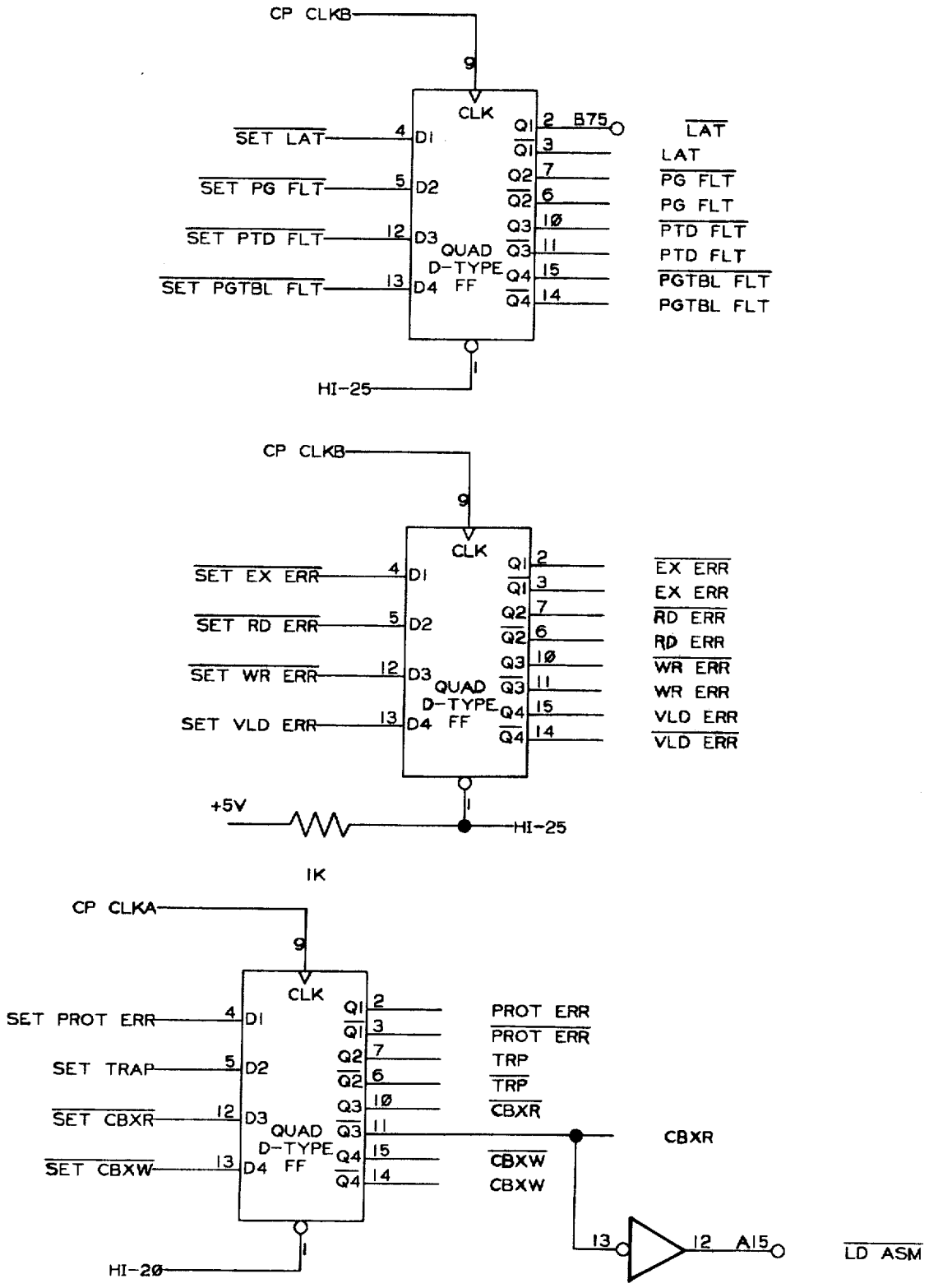

CP CLKB
9
CLK
SET LAT
4
D1
SET PG FLT
5
D2
SET PTD FLT
12
D3
SET PGTBL FLT
13
D4
QUAD
D-TYPE
FF
Q1
Q1
Q2
Q2
Q3
Q3
Q4
Q4
2
3
7
6
10
11
15
14
B75
LAT
LAT
PG FLT
PG FLT
PTD FLT
PTD FLT
PGTBL FLT
PGTBL FLT
1
HI-25
CP CLKB
9
CLK
SET EX ERR
4
D1
SET RD ERR
5
D2
SET WR ERR
12
D3
SET VLD ERR
13
D4
QUAD
D-TYPE
FF
EX ERR
EX ERR
RD ERR
RD ERR
WR ERR
WR ERR
VLD ERR
VLD ERR
+5V
1K
1
HI-25
CP CLKA
9
CLK
SET PROT ERR
4
D1
SET TRAP
5
D2
SET CBXR
12
D3
SET CBXW
13
D4
QUAD
D-TYPE
FF
PROT ERR
PROT ERR
TRP
TRP
CBXR
CBXR
CBXW
CBXW
13
12
A15
LD ASM
1
HI-20

FIG. 93D

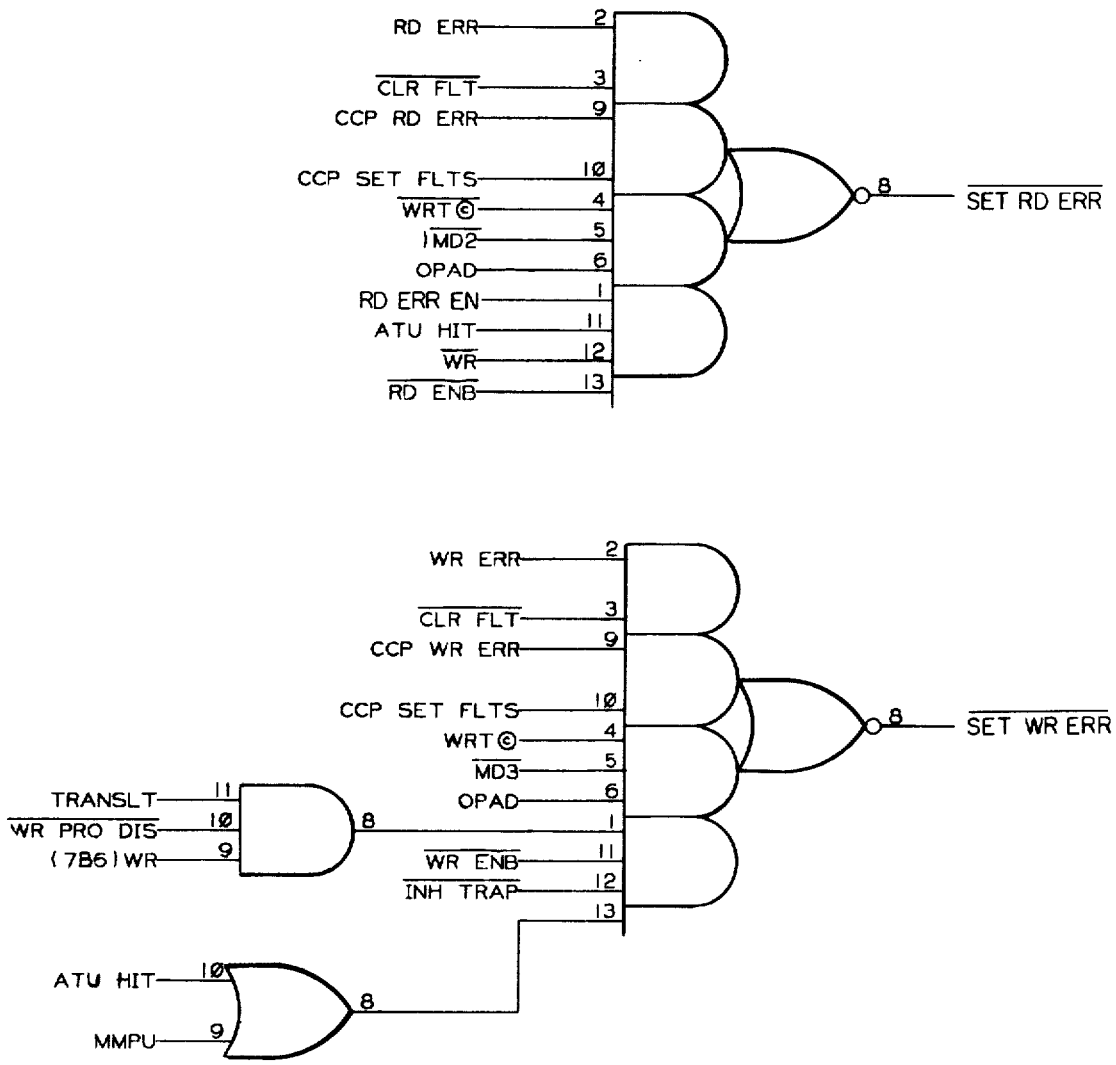
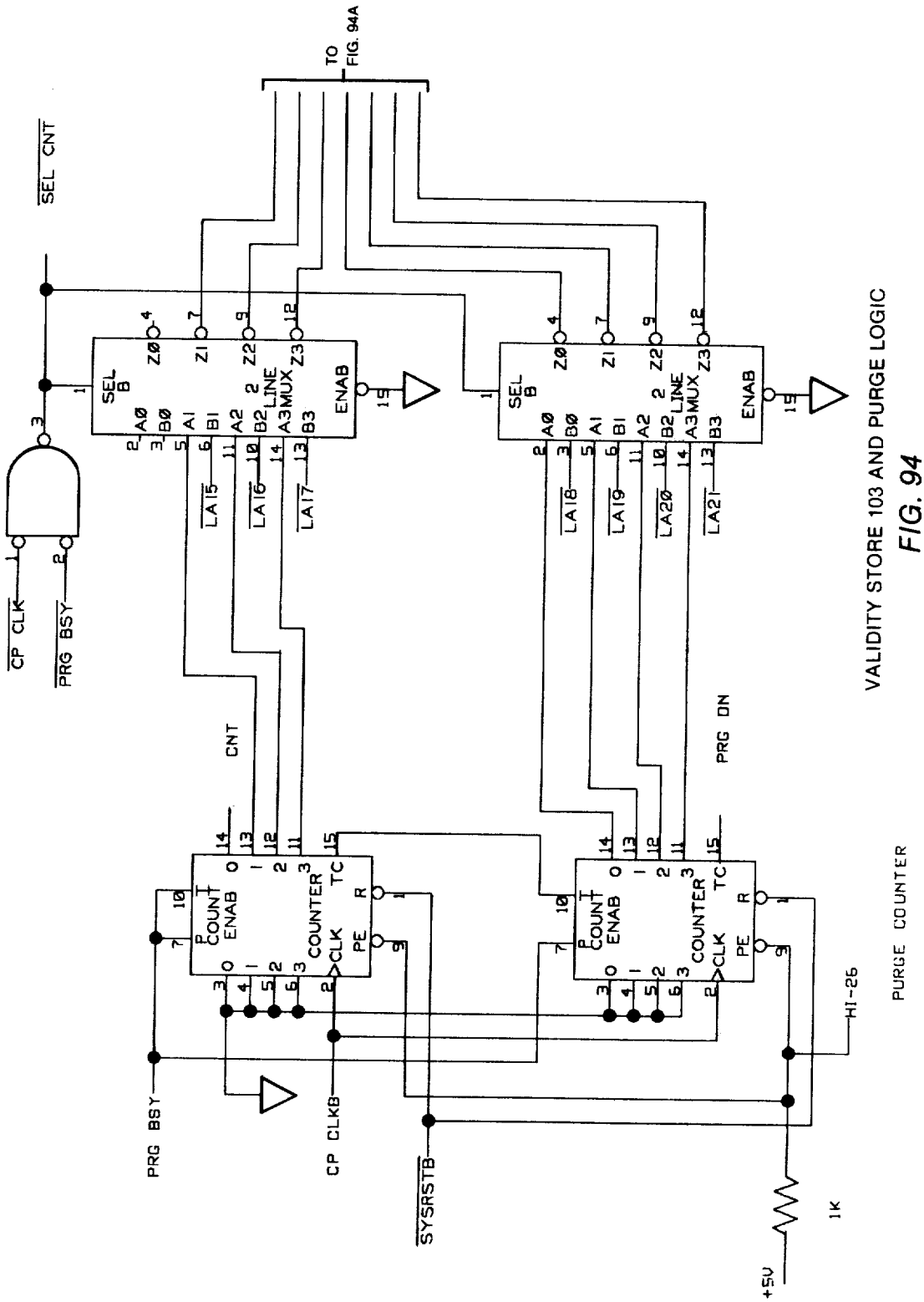

RD ERR
CLR FLT
CCP RD ERR
CCP SET FLTS
WRT©
1MD2
OPAD
RD ERR EN
ATU HIT
WR
RD ENB
SET RD ERR
WR ERR
CLR FLT
CCP WR ERR
CCP SET FLTS
WRT©
MD3
OPAD
WR ENB
INH TRAP
SET WR ERR
TRANSLT
WR PRO DIS
(7B6)WR
ATU HIT
MMPU

FIG. 93E



VALIDITY STORE 103 AND PURGE LOGIC

FIG. 94

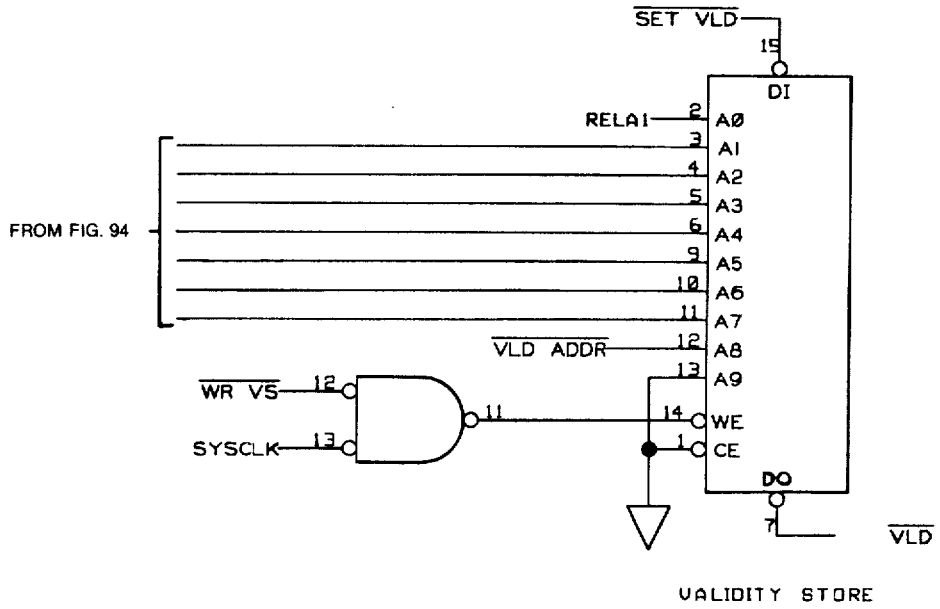
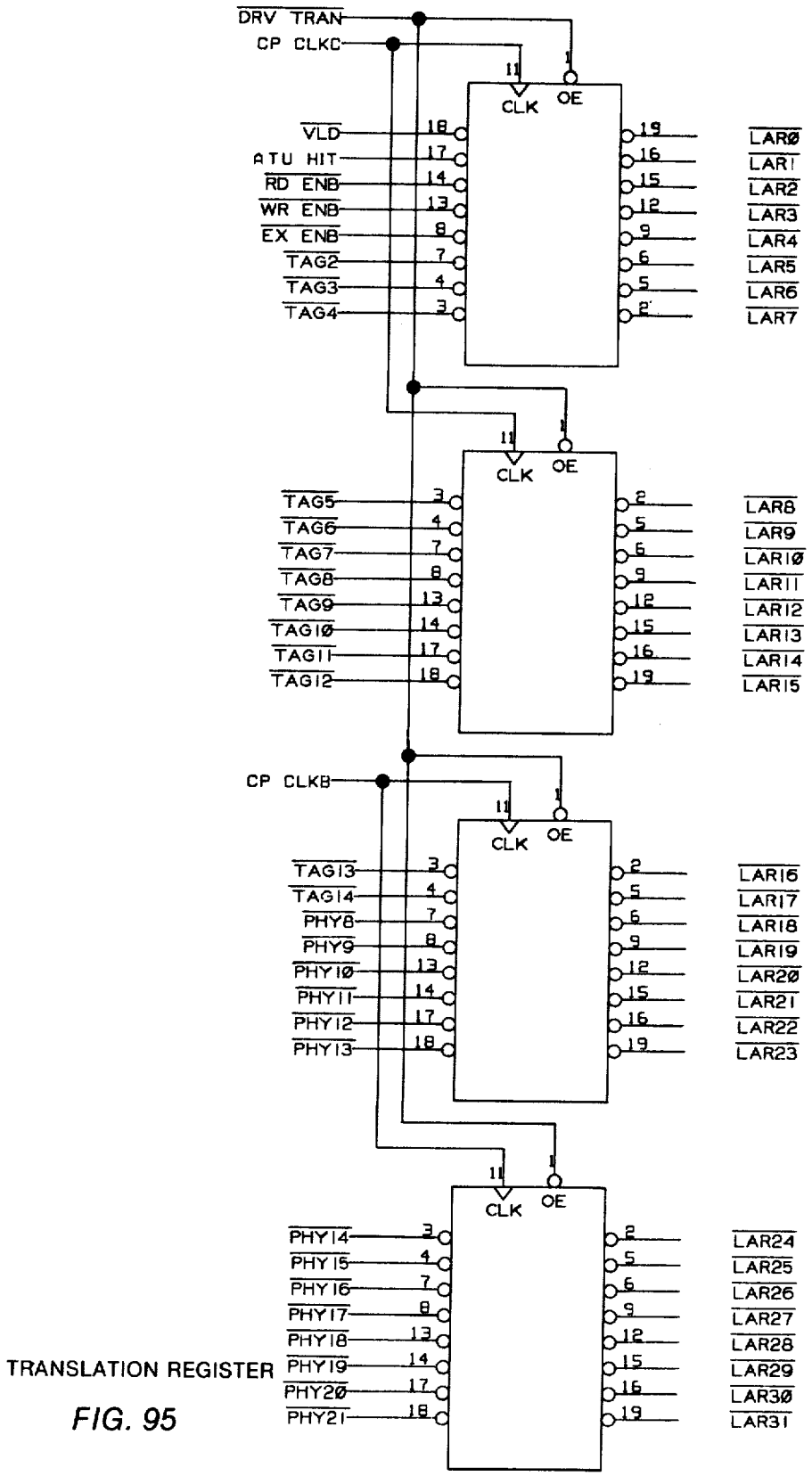
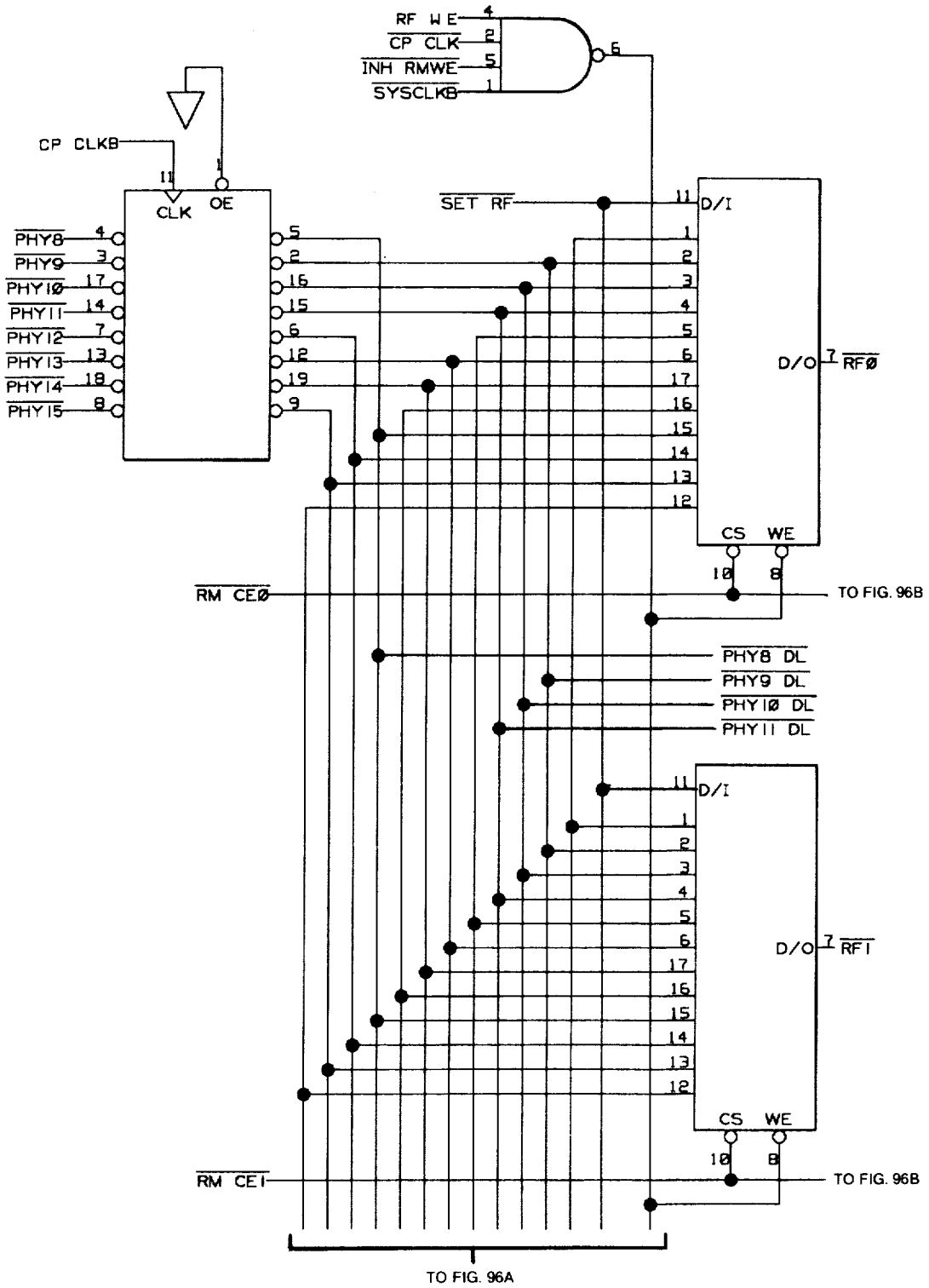

SET VLD
15
DI
RELAI
2
A0
3
A1
4
A2
5
A3
6
A4
9
A5
10
A6
11
A7
VLD ADDR
12
A8
13
A9
FROM FIG. 94
WR VS
12
11
14
WE
SYSCLK
13
1
CE
DO
7
VLD
VALIDITY STORE

FIG. 94A



TRANSLATION REGISTER

FIG. 95



REFERENCE/MODIFY STORAGE AND CONTROL LOGIC

FIG. 96

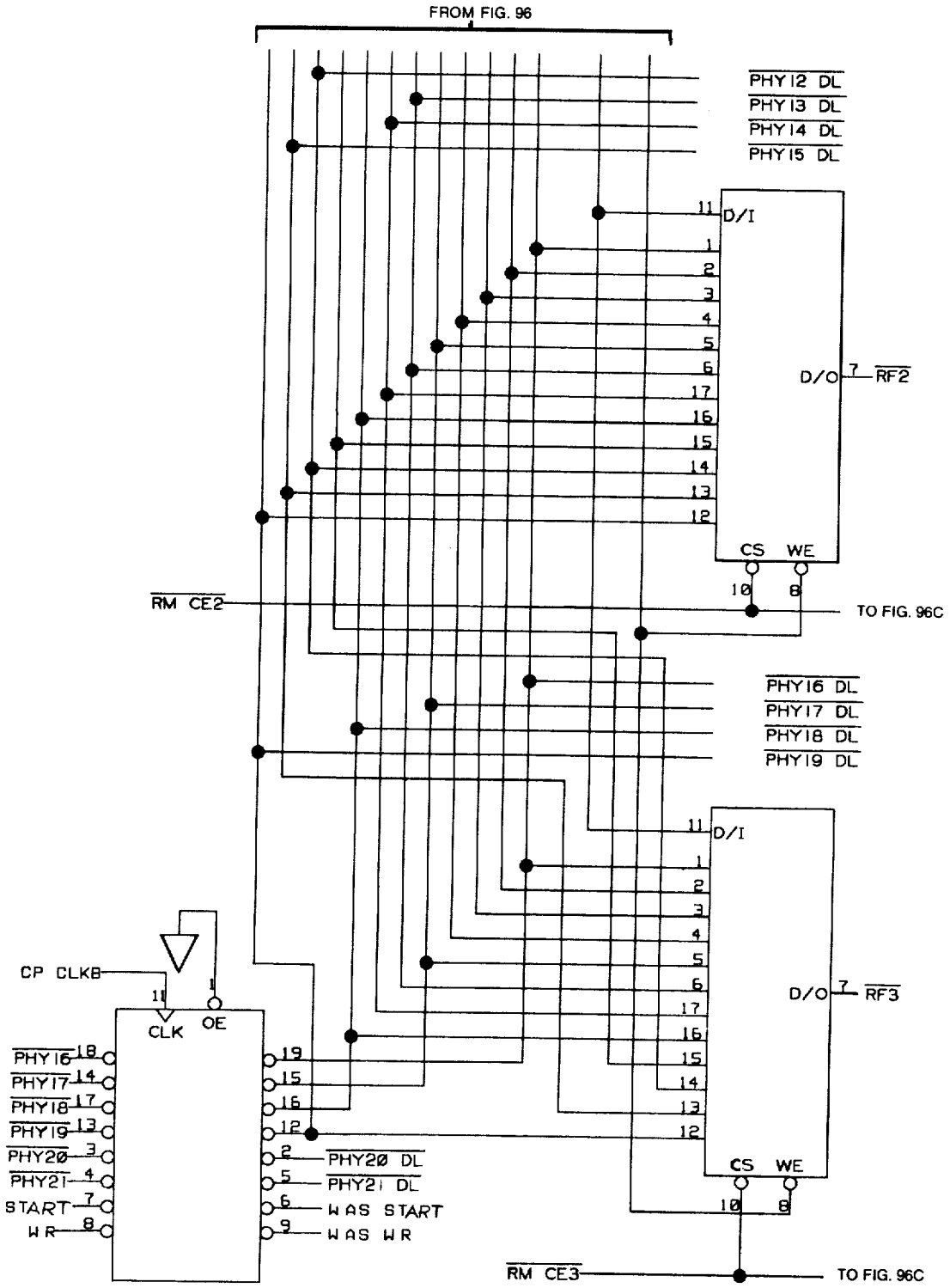

FROM FIG. 96
PHY12 DL
PHY13 DL
PHY14 DL
PHY15 DL
D/I
D/O
RF2
CS
WE
RM CE2
TO FIG. 96C
PHY16 DL
PHY17 DL
PHY18 DL
PHY19 DL
D/I
D/O
RF3
CS
WE
CP CLKB
CLK
OE
PHY16
PHY17
PHY18
PHY19
PHY20
PHY21
START
WR
PHY20 DL
PHY21 DL
WAS START
WAS WR
RM CE3
TO FIG. 96C

FIG. 96A

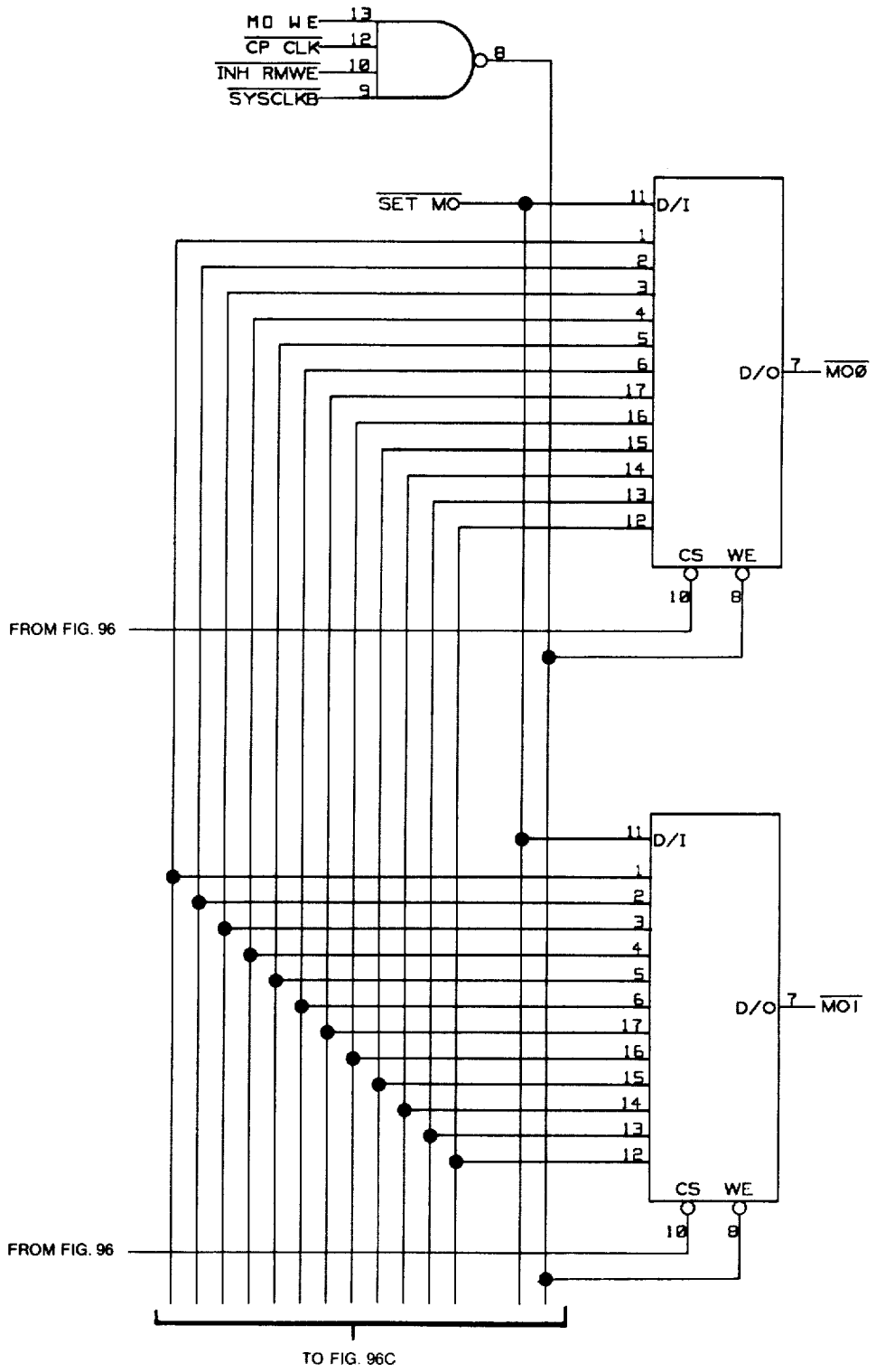

MO WE
CP CLK
INH RMWE
SYSCLKB
13
12
10
9
8
SET MO
11
D/I
1
2
3
4
5
6
17
16
15
14
13
12
D/O
7
MOØ
CS
WE
10
8
FROM FIG. 96
11
D/I
1
2
3
4
5
6
17
16
15
14
13
12
D/O
7
MO1
CS
WE
10
8
FROM FIG. 96
TO FIG. 96C

FIG. 96B

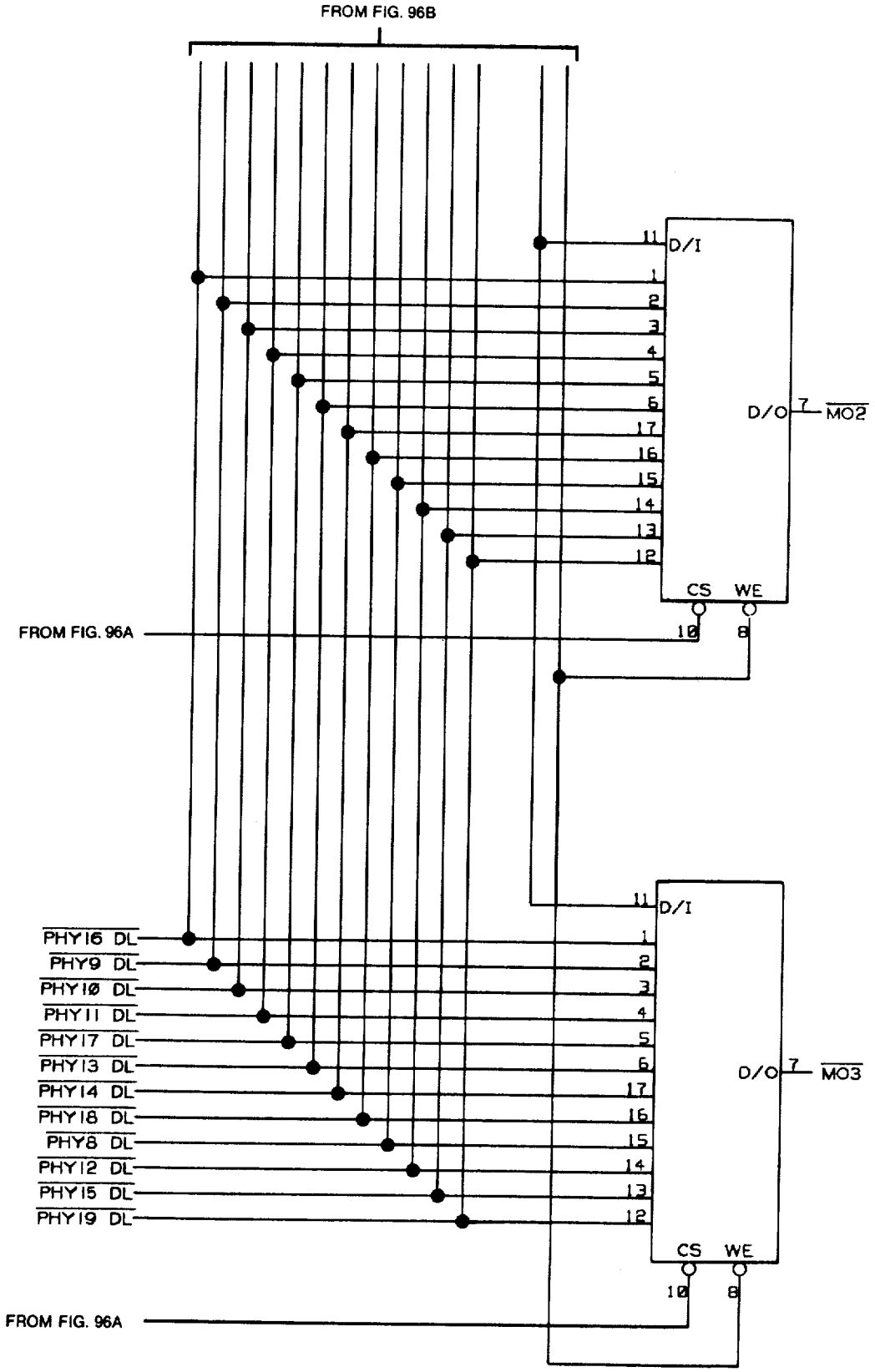

FROM FIG. 96B
D/I
D/O
MO2
CS
WE
FROM FIG. 96A
PHY16 DL
PHY9 DL
PHY10 DL
PHY11 DL
PHY17 DL
PHY13 DL
PHY14 DL
PHY18 DL
PHY8 DL
PHY12 DL
PHY15 DL
PHY19 DL
MO3

FIG. 96C

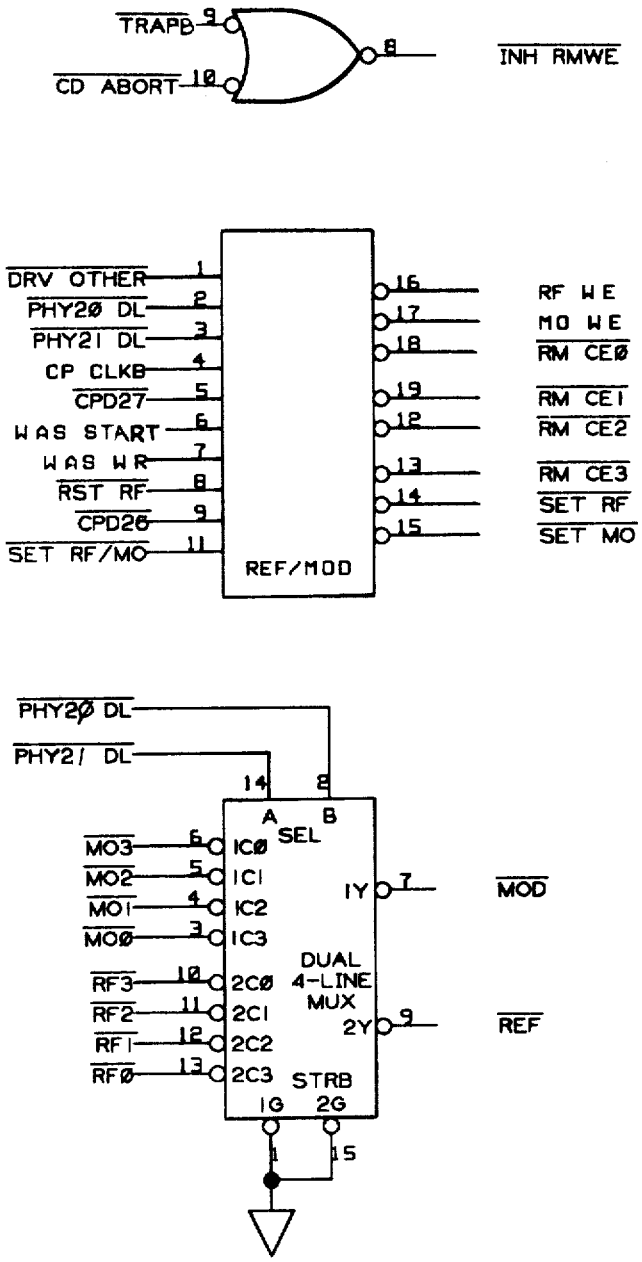
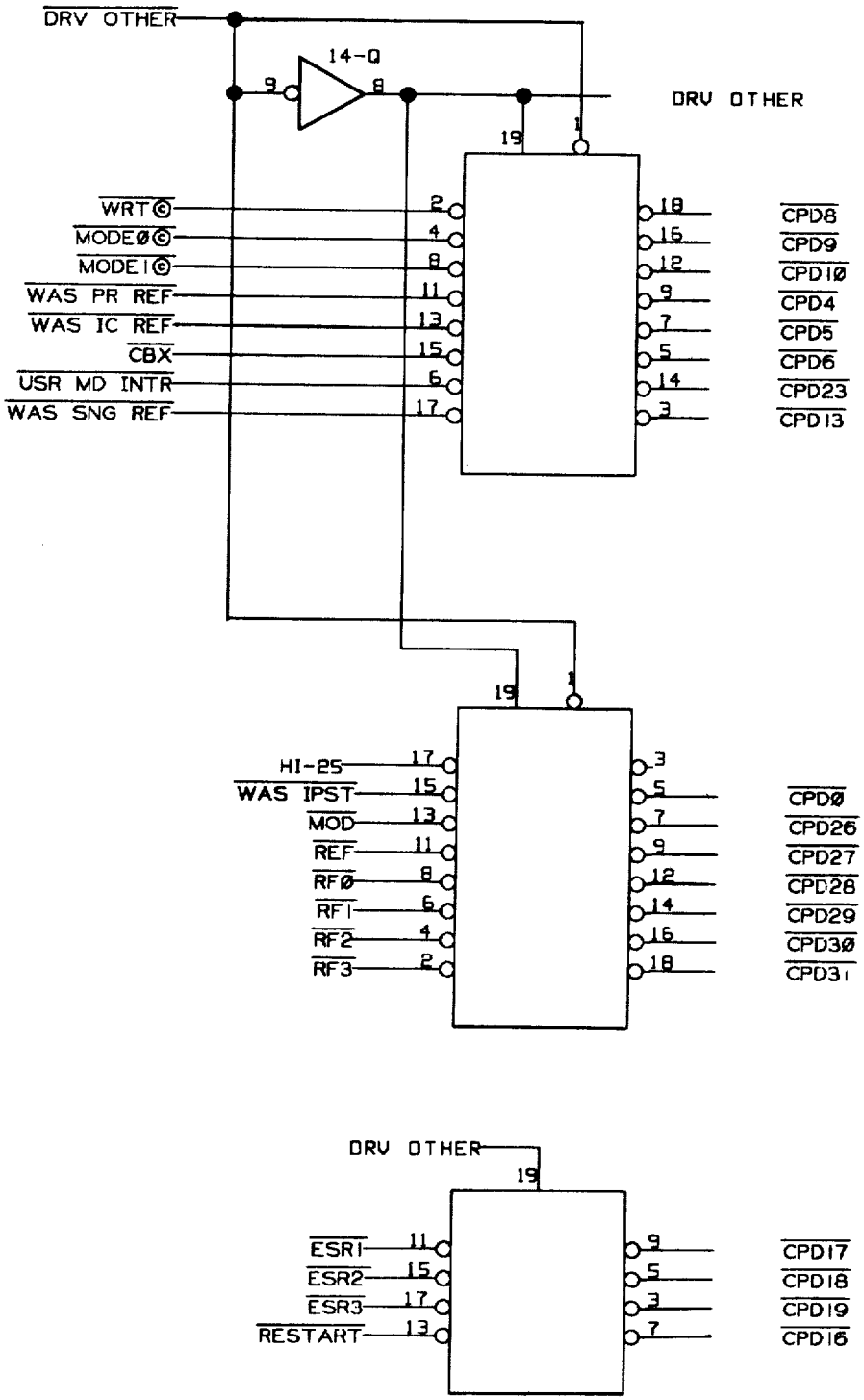
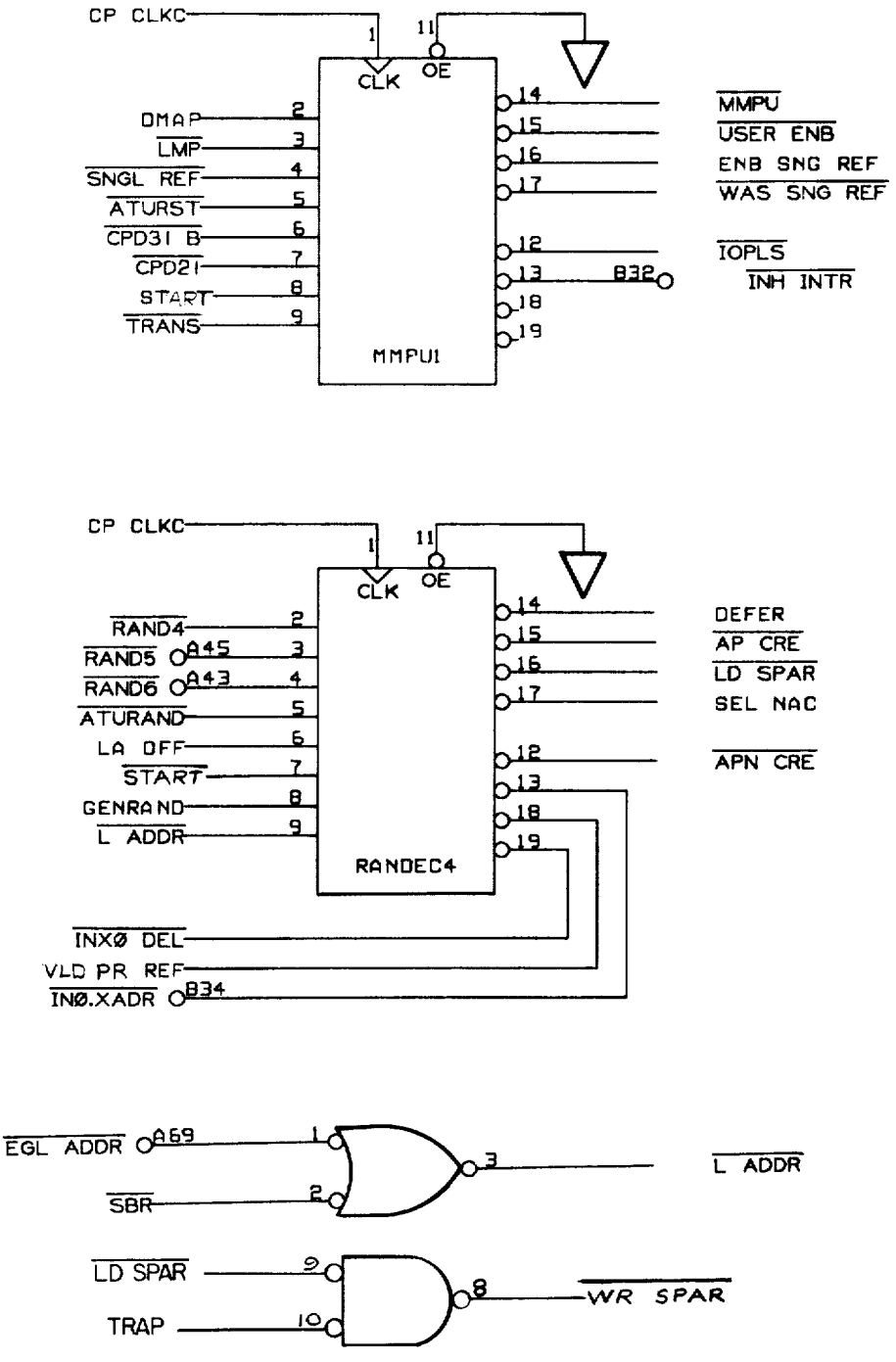

TRAPB
CD ABORT
INH RMWE
DRV OTHER
PHY2Ø DL
PHY21 DL
CP CLKB
CPD27
WAS START
WAS WR
RST RF
CPD26
SET RF/MO
REF/MOD
RF WE
MO WE
RM CEØ
RM CE1
RM CE2
RM CE3
SET RF
SET MO
PHY2Ø DL
PHY21 DL
A
B
SEL
MO3
MO2
MO1
MOØ
RF3
RF2
RF1
RFØ
1CØ
1C1
1C2
1C3
2CØ
2C1
2C2
2C3
1Y
2Y
DUAL
4-LINE
MUX
STRB
1G
2G
MOD
REF

FIG. 96D



STATE SAVE DRIVERS

FIG. 97



16 BIT MMPU EMULATION CONTROL LOGIC

FIG. 98

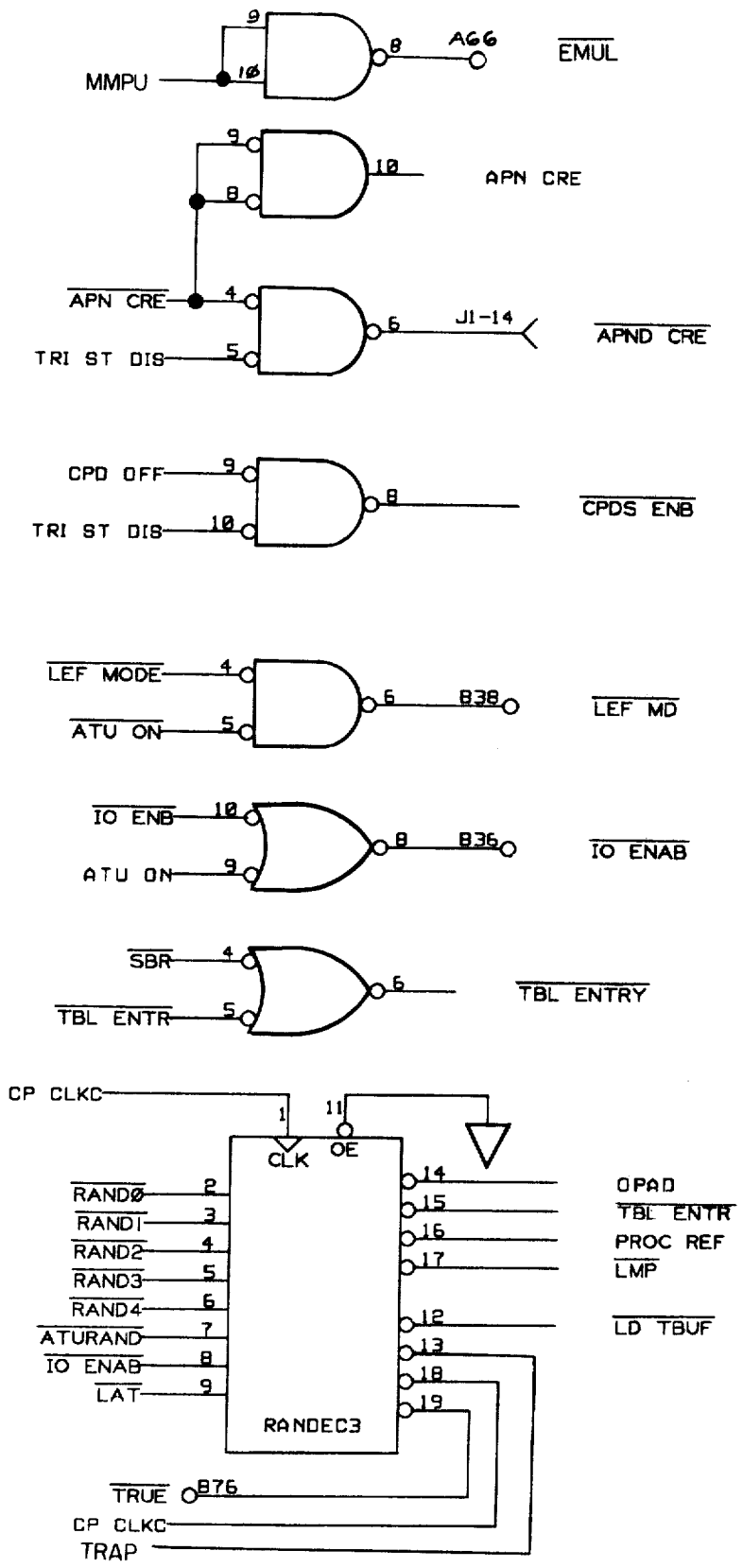

MMPU
9
10
8
A66
EMUL
9
8
10
APN CRE
APN CRE
4
TRI ST DIS
5
6
J1-14
APND CRE
CPD OFF
9
TRI ST DIS
10
8
CPDS ENB
LEF MODE
4
ATU ON
5
6
B38
LEF MD
IO ENB
10
ATU ON
9
8
B36
IO ENAB
SBR
4
TBL ENTR
5
6
TBL ENTRY
CP CLKC
1
11
CLK
OE
RAND0
2
RAND1
3
RAND2
4
RAND3
5
RAND4
6
ATURAND
7
IO ENAB
8
LAT
9
14
15
16
17
12
13
18
19
RANDEC3
OPAD
TBL ENTR
PROC REF
LMP
LD TBUF
TRUE
B76
CP CLKC
TRAP

FIG. 98A

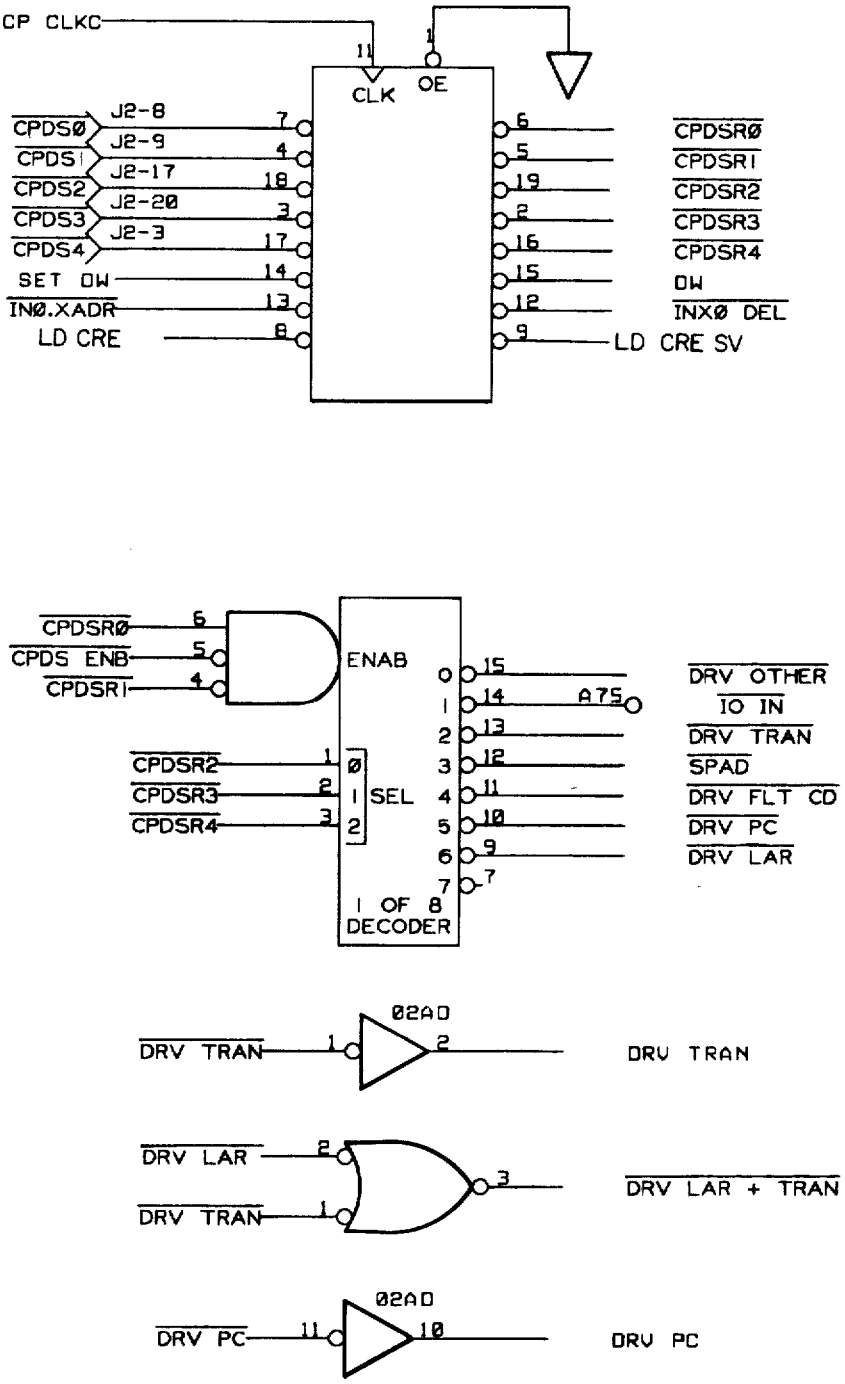

CP CLKC
CLK
OE
CPDS0
CPDS1
CPDS2
CPDS3
CPDS4
J2-8
J2-9
J2-17
J2-20
J2-3
SET DW
IN0.XADR
LD CRE
CPDSR0
CPDSR1
CPDSR2
CPDSR3
CPDSR4
DW
INX0 DEL
LD CRE SV
CPDSR0
CPDS ENB
CPDSR1
ENAB
CPDSR2
CPDSR3
CPDSR4
SEL
1 OF 8
DECODER
A75
DRV OTHER
IO IN
DRV TRAN
SPAD
DRV FLT CD
DRV PC
DRV LAR
02AD
DRV TRAN
DRV TRAN
DRV LAR
DRV TRAN
DRV LAR + TRAN
02AD
DRV PC
DRV PC

FIG. 98B

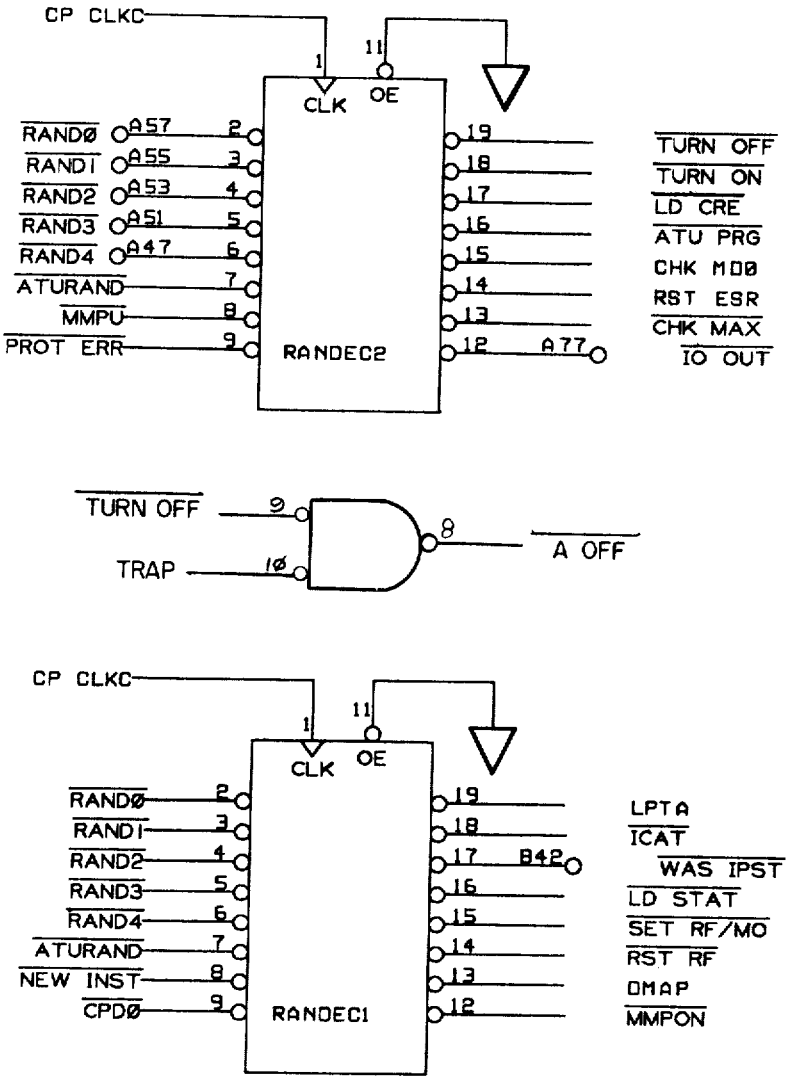

CP CLKC
CLK
OE
RANDØ
RAND1
RAND2
RAND3
RAND4
ATURAND
MMPU
PROT ERR
A57
A55
A53
A51
A47
RANDEC2
A77
TURN OFF
TURN ON
LD CRE
ATU PRG
CHK MDØ
RST ESR
CHK MAX
IO OUT
TURN OFF
TRAP
A OFF
CP CLKC
CLK
OE
RANDØ
RAND1
RAND2
RAND3
RAND4
ATURAND
NEW INST
CPDØ
RANDEC1
B42
LPTA
ICAT
WAS IPST
LD STAT
SET RF/MO
RST RF
DMAP
MMPON

FIG. 98C

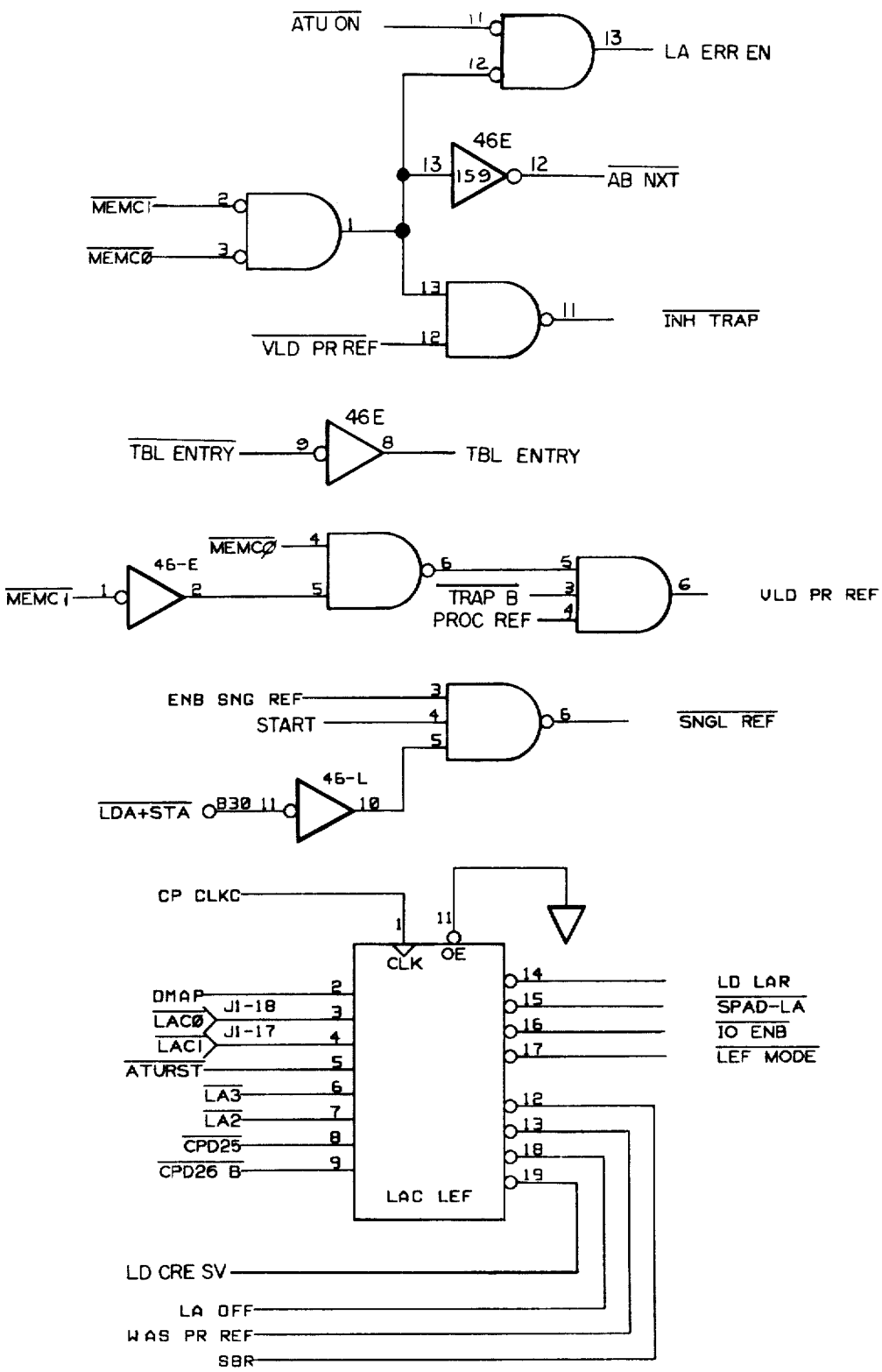

ATU ON
LA ERR EN
46E
159
AB NXT
MEMC1
MEMCØ
INH TRAP
VLD PR REF
46E
TBL ENTRY
TBL ENTRY
MEMCØ
46-E
MEMC1
TRAP B
PROC REF
VLD PR REF
ENB SNG REF
START
SNGL REF
46-L
LDA+STA
B30
CP CLKC
CLK
OE
DMAP
LACØ
LAC1
J1-18
J1-17
ATURST
LA3
LA2
CPD25
CPD26 B
LAC LEF
LD LAR
SPAD-LA
IO ENB
LEF MODE
LD CRE SV
LA OFF
WAS PR REF
SBR

FIG. 98D

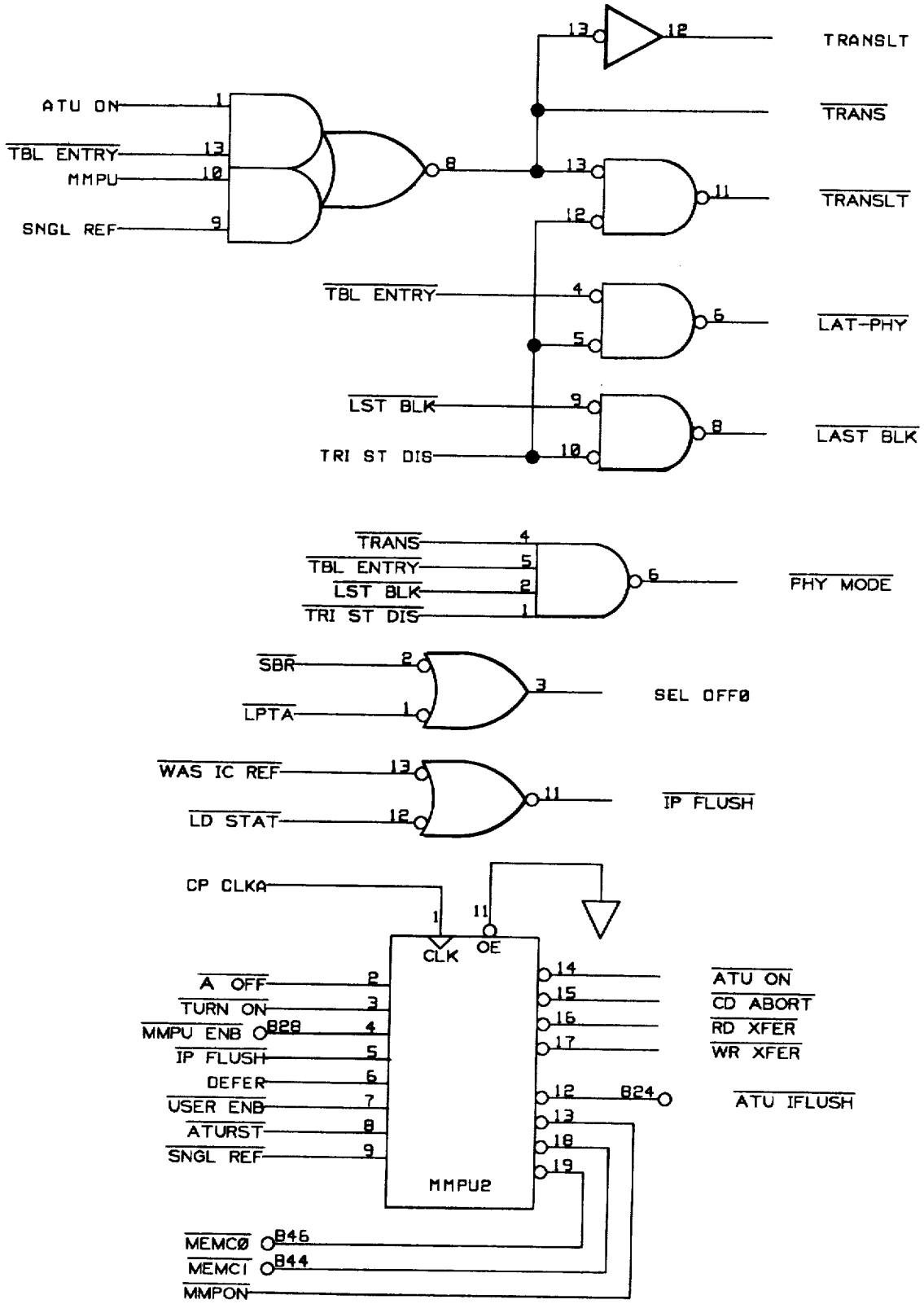

TRANSLT
TRANS
TRANSLT
LAT-PHY
LAST BLK
ATU ON
TBL ENTRY
MMPU
SNGL REF
TBL ENTRY
LST BLK
TRI ST DIS
TRANS
TBL ENTRY
LST BLK
TRI ST DIS
PHY MODE
SBR
LPTA
SEL OFF0
WAS IC REF
LD STAT
IP FLUSH
CP CLKA
CLK
OE
A OFF
TURN ON
MMPU ENB
IP FLUSH
DEFER
USER ENB
ATURST
SNGL REF
MMPU2
ATU ON
CD ABORT
RD XFER
WR XFER
ATU IFLUSH
MEMC0
MEMC1
MMPON

FIG. 98E

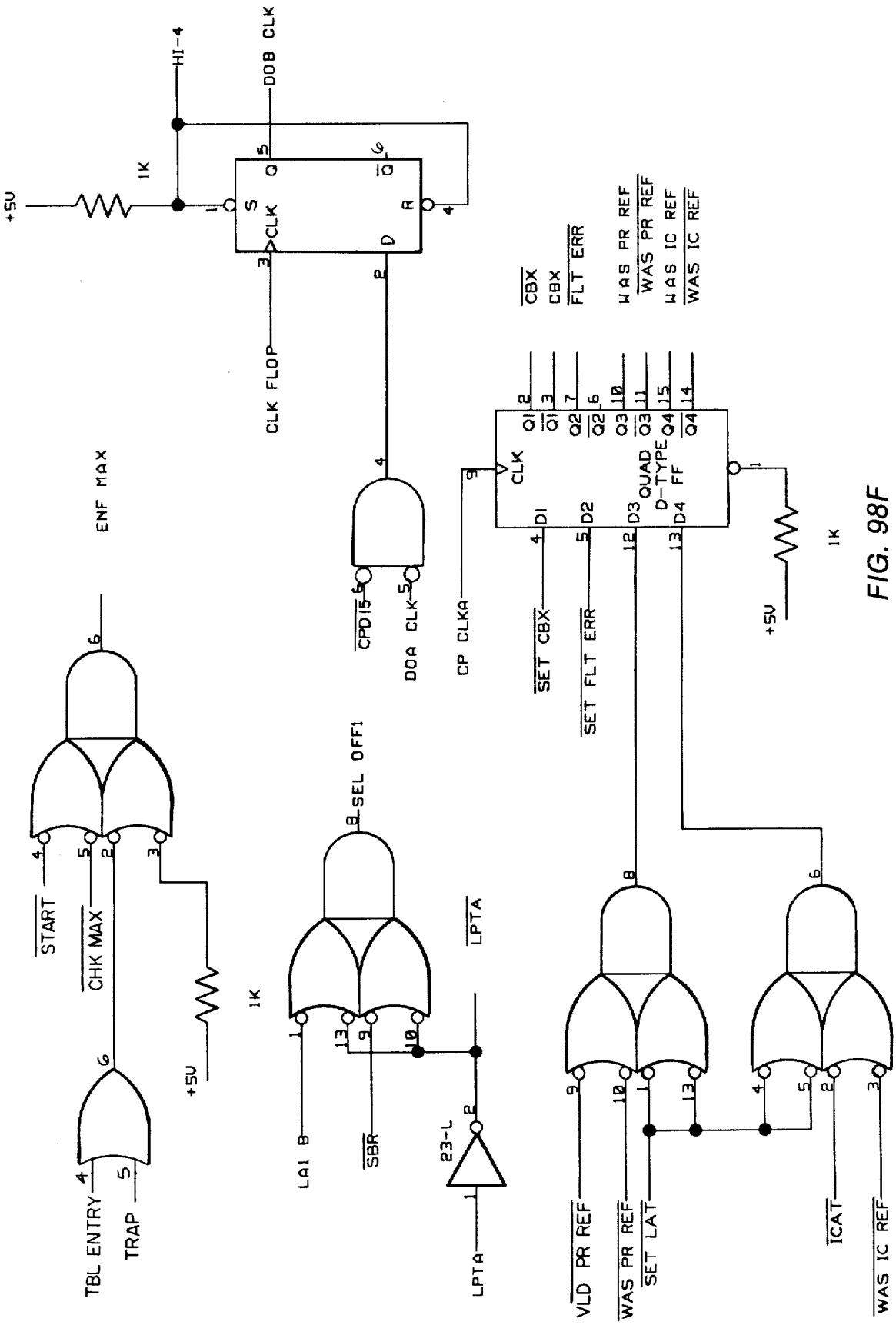

+5V
1K
HI-4
DOB CLK
S
CLK
Q
R
D
CLK FLOP
CPD15
DOA CLK
CP CLKA
CLK
D1
D2
D3
D4
QUAD
D-TYPE
FF
Q1
Q2
Q3
Q4
SET CBX
SET FLT ERR
CBX
CBX
FLT ERR
WAS PR REF
WAS PR REF
WAS IC REF
WAS IC REF
ENF MAX
START
CHK MAX
TBL ENTRY
TRAP
SEL OFF1
LA1 B
SBR
23-L
LPTA
LPTA
VLD PR REF
WAS PR REF
SET LAT
ICAT
WAS IC REF

FIG. 98F

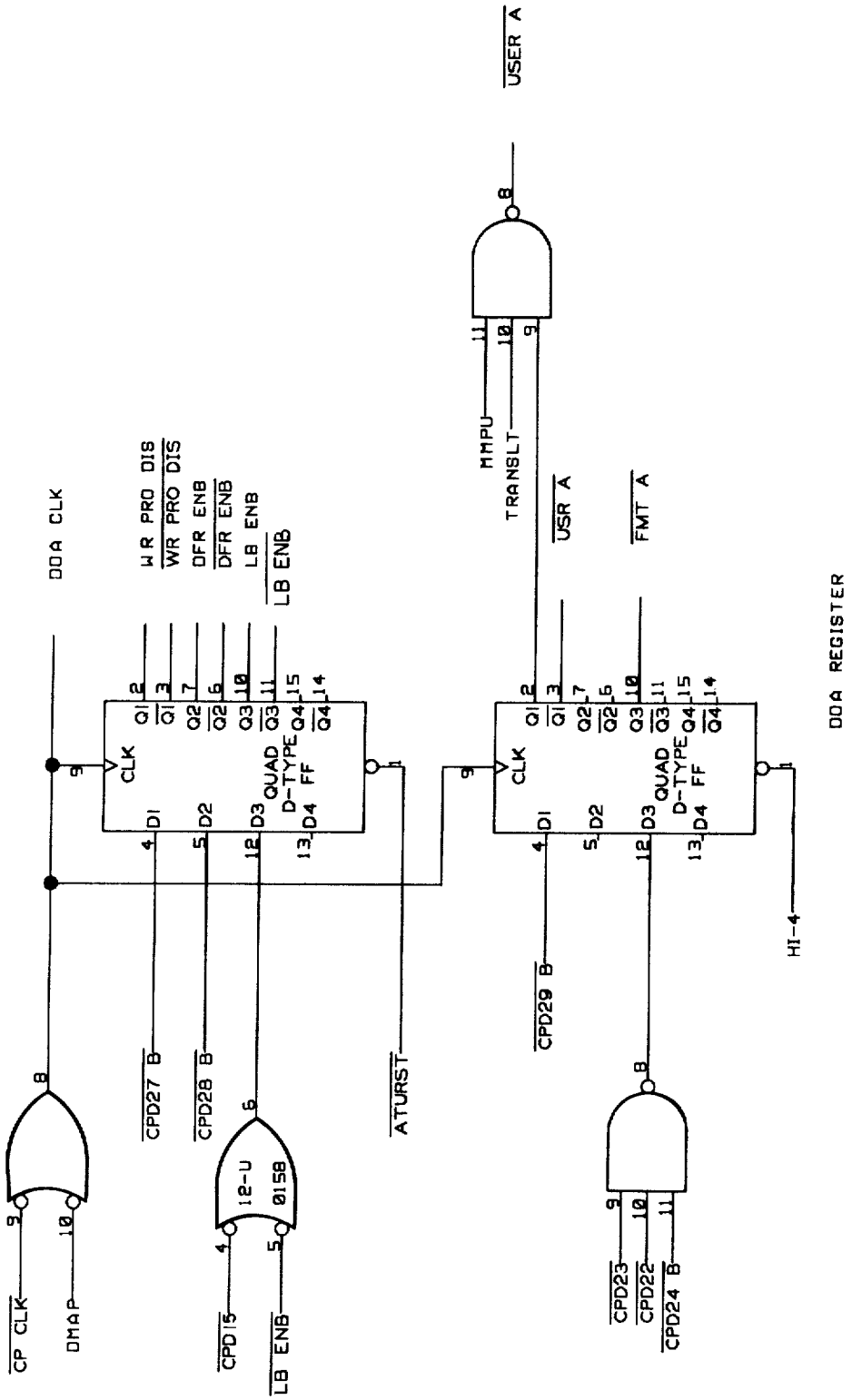
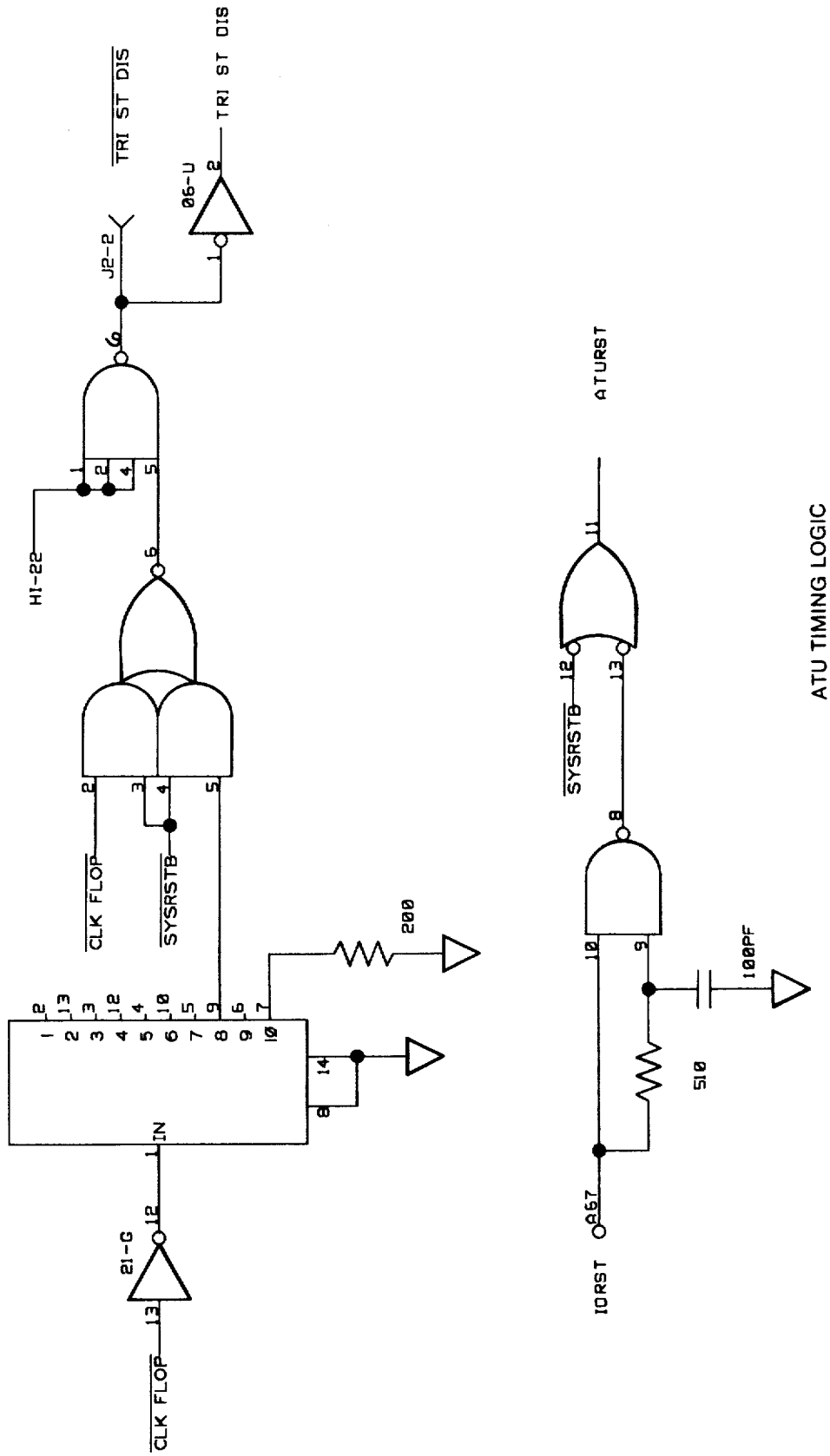

DDA CLK
WR PRO DIS
WR PRO DIS
DFR ENB
DFR ENB
LB ENB
LB ENB
CLK
Q1
Q1
Q2
Q2
Q3
Q3
Q4
Q4
QUAD
D-TYPE
FF
D1
D2
D3
D4
CP CLK
DMAP
CPD27 B
CPD28 B
CPD15
LB ENB
12-U
0158
ATURST
MMPU
TRANSLT
USER A
USR A
FMT A
CPD29 B
CPD23
CPD22
CPD24 B
HI-4
DDA REGISTER

FIG. 98G



ATU TIMING LOGIC

FIG. 99

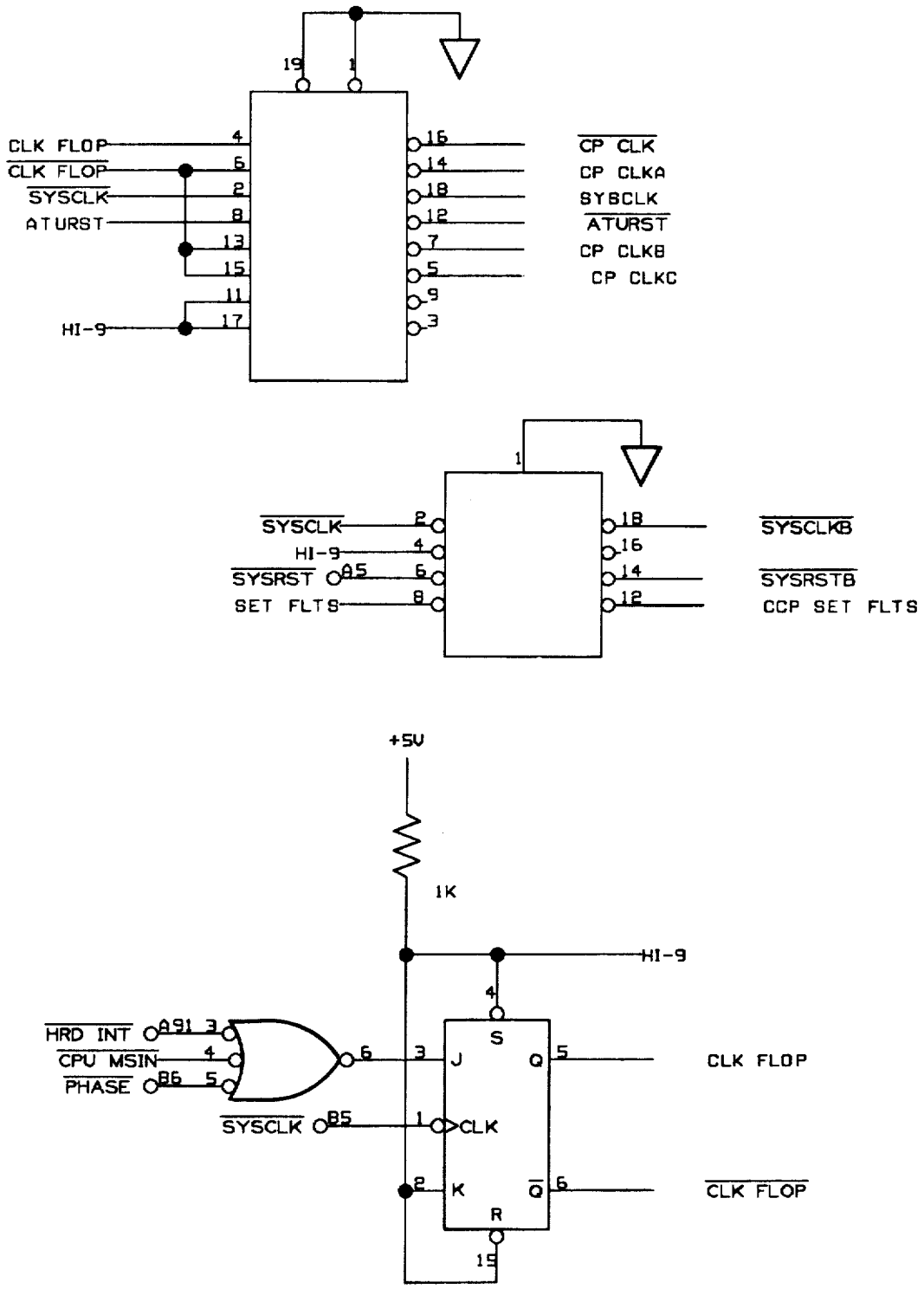
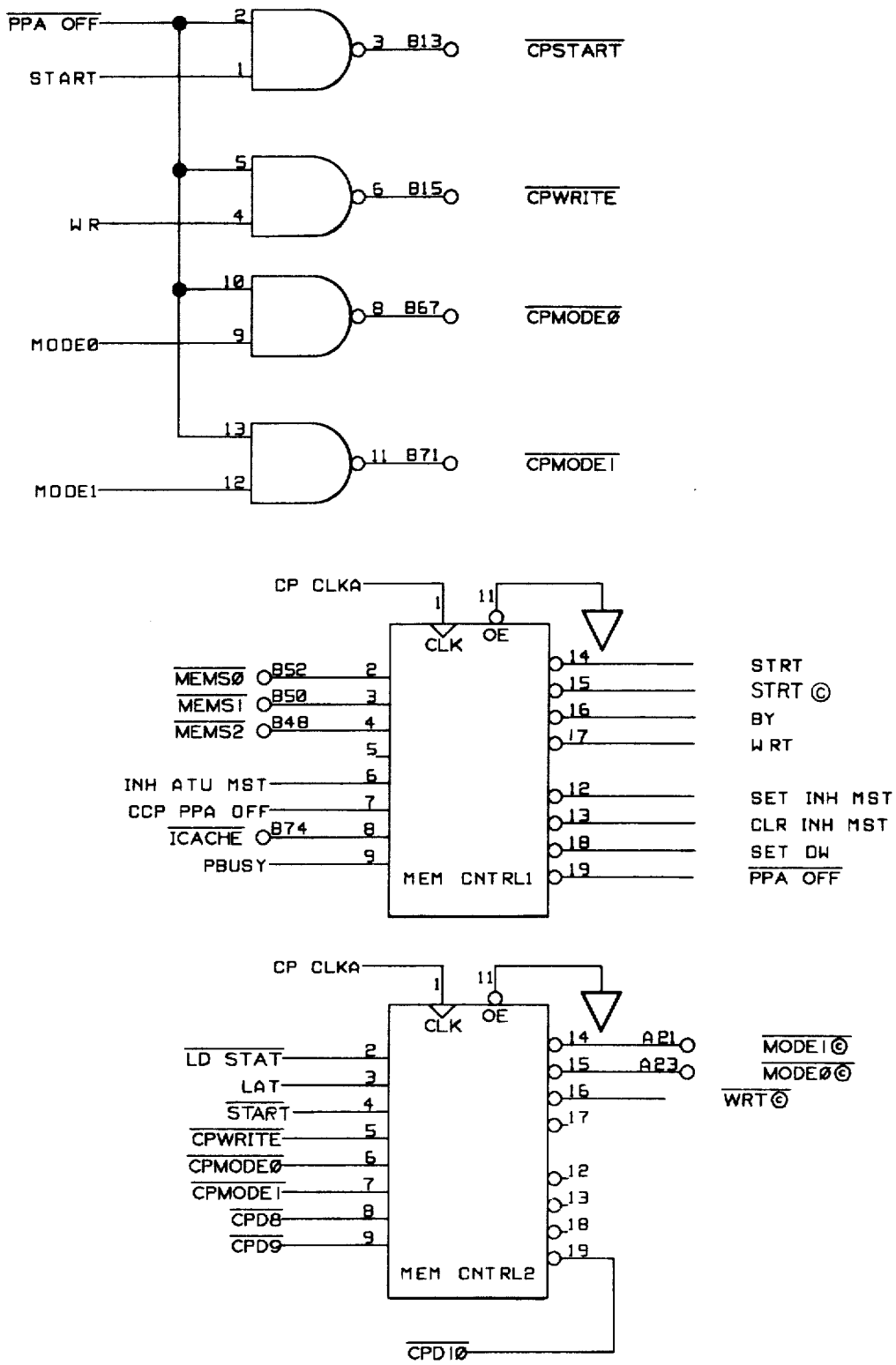

19
1
CLK FLOP
CLK FLOP
SYSCLK
ATURST
HI-9
4
6
2
8
13
15
11
17
16
14
18
12
7
5
9
3
CP CLK
CP CLKA
SYBCLK
ATURST
CP CLKB
CP CLKC
1
SYSCLK
HI-9
SYSRST
SET FLTS
A5
2
4
6
8
18
16
14
12
SYSCLKB
SYSRSTB
CCP SET FLTS
+5V
1K
HI-9
HRD INT
CPU MSIN
PHASE
A91
B6
3
4
5
6
3
J
S
Q
5
CLK FLOP
SYSCLK
B5
1
CLK
2
K
R
Q
6
CLK FLOP
15

FIG. 99A



SYSTEM CACHE INTERFACE LOGIC

FIG. 100

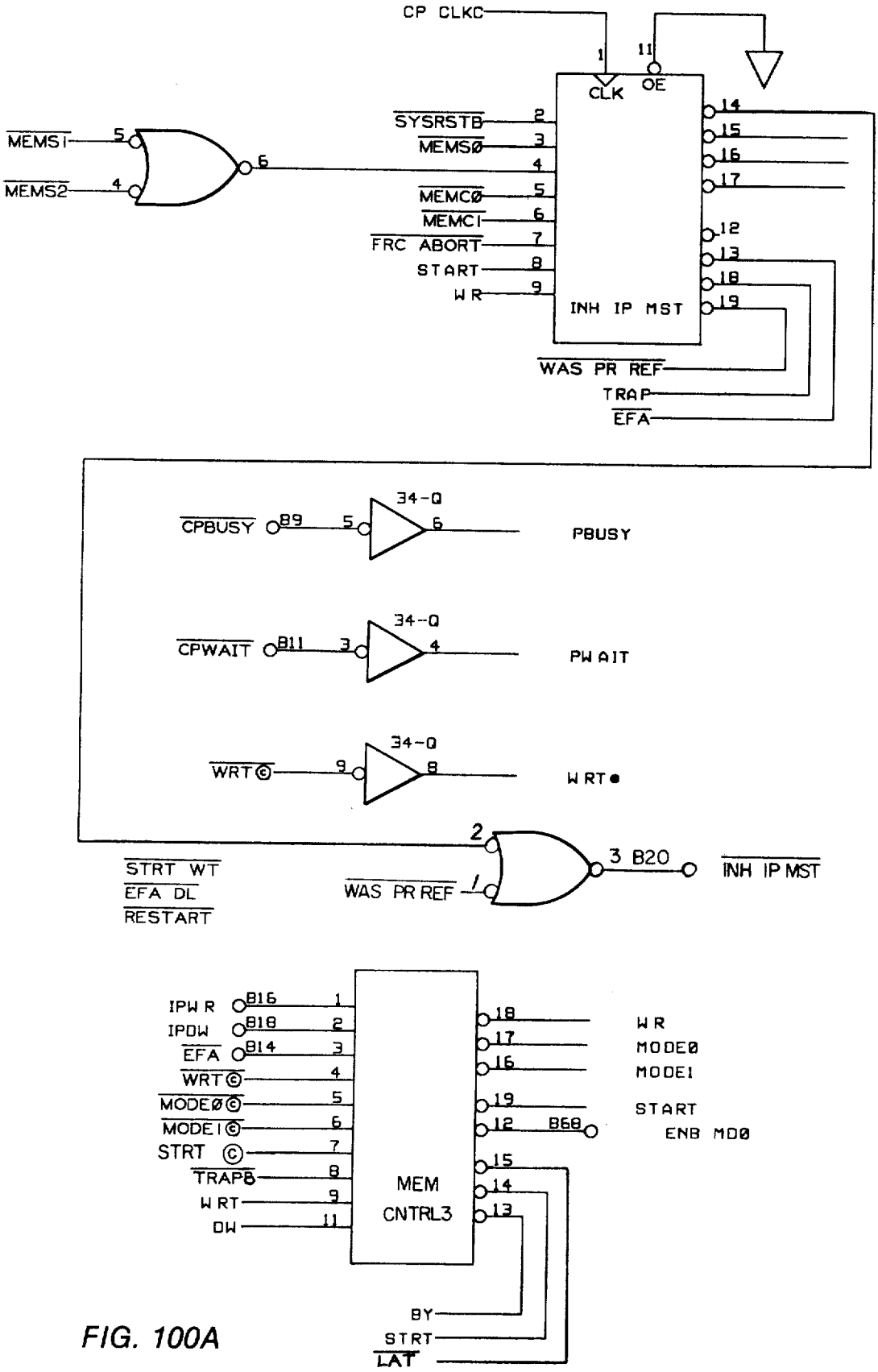

CP CLKC
CLK
OE
MEMS1
MEMS2
SYSRSTB
MEMSØ
MEMCØ
MEMC1
FRC ABORT
START
WR
INH IP MST
WAS PR REF
TRAP
EFA
CPBUSY
34-Q
PBUSY
CPWAIT
PWAIT
WRT©
WRT●
STRT WT
EFA DL
RESTART
WAS PR REF
B20
INH IP MST
IPWR
IPDW
EFA
WRT©
MODEØ©
MODE1©
STRT ©
TRAPB
WRT
DW
MEM
CNTRL3
WR
MODEØ
MODE1
START
ENB MDØ
BY
STRT
LAT

FIG. 100A

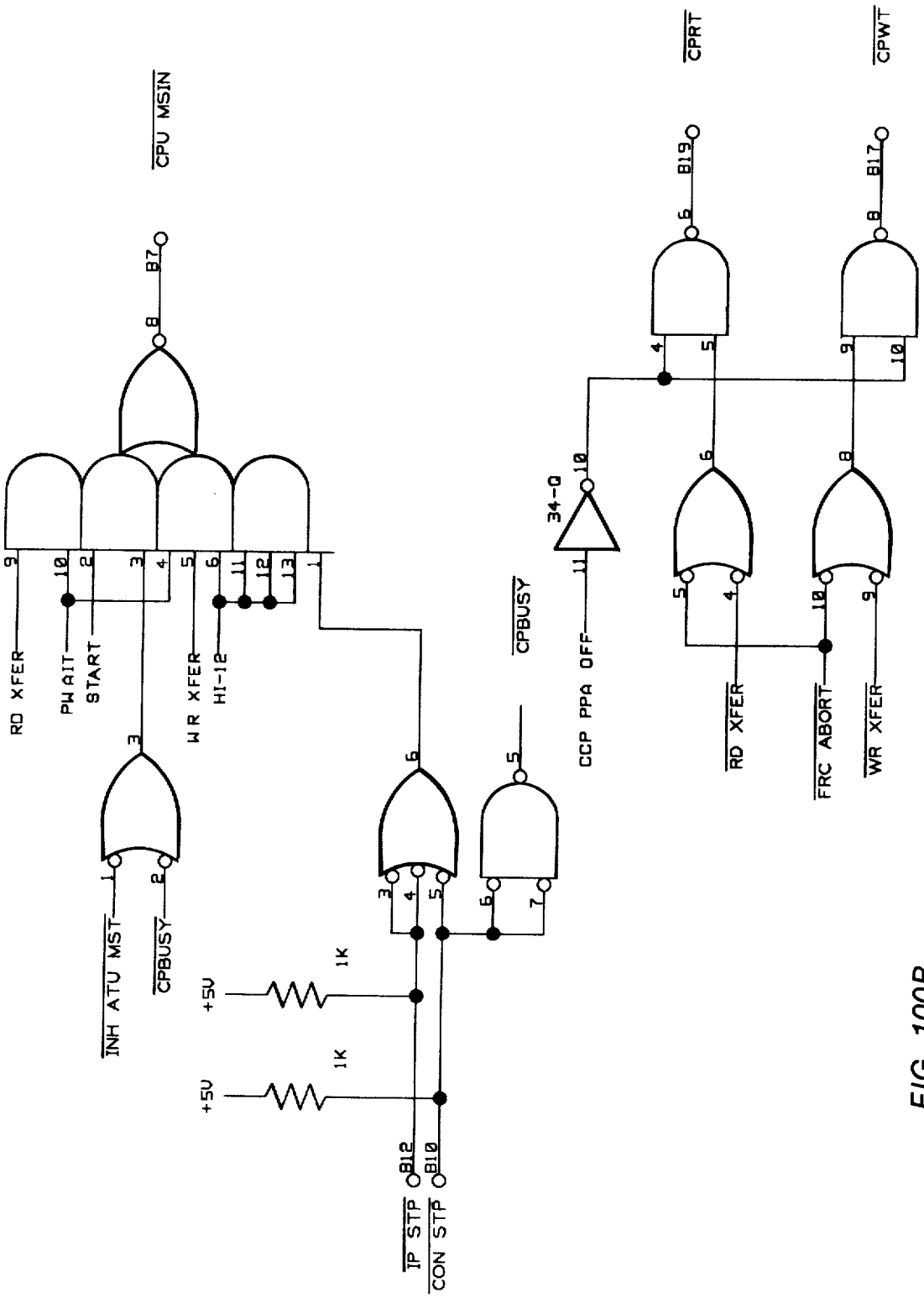

CPU MSIN
B7
RD XFER
PW AIT
START
WR XFER
HI-12
INH ATU MST
CPBUSY
+5V
1K
+5V
1K
B12
B10
IP STP
CON STP
CPBUSY
34-Q
CCP PPA OFF
CPRT
B19
CPWT
B17
RD XFER
FRC ABORT
WR XFER

FIG. 100B

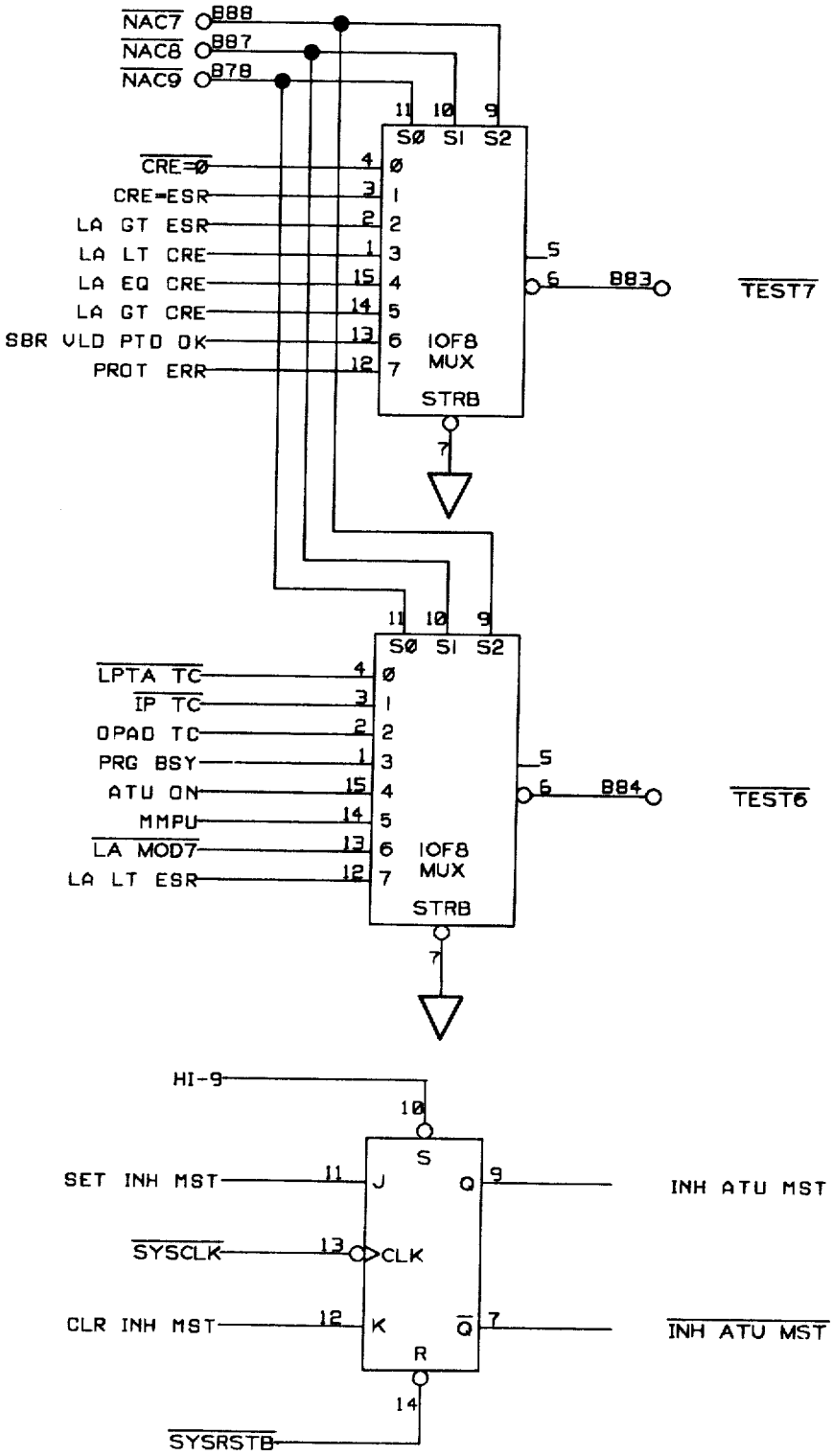

NAC7
NAC8
NAC9
B88
B87
B78
S0 S1 S2
CRE=0
CRE=ESR
LA GT ESR
LA LT CRE
LA EQ CRE
LA GT CRE
SBR VLD PTD OK
PROT ERR
1OF8
MUX
STRB
B83
TEST7
LPTA TC
IP TC
OPAD TC
PRG BSY
ATU ON
MMPU
LA MOD7
LA LT ESR
B84
TEST6
HI-9
SET INH MST
SYSCLK
CLR INH MST
S
J
Q
CLK
K
R
INH ATU MST
SYSRSTB

FIG. 100C

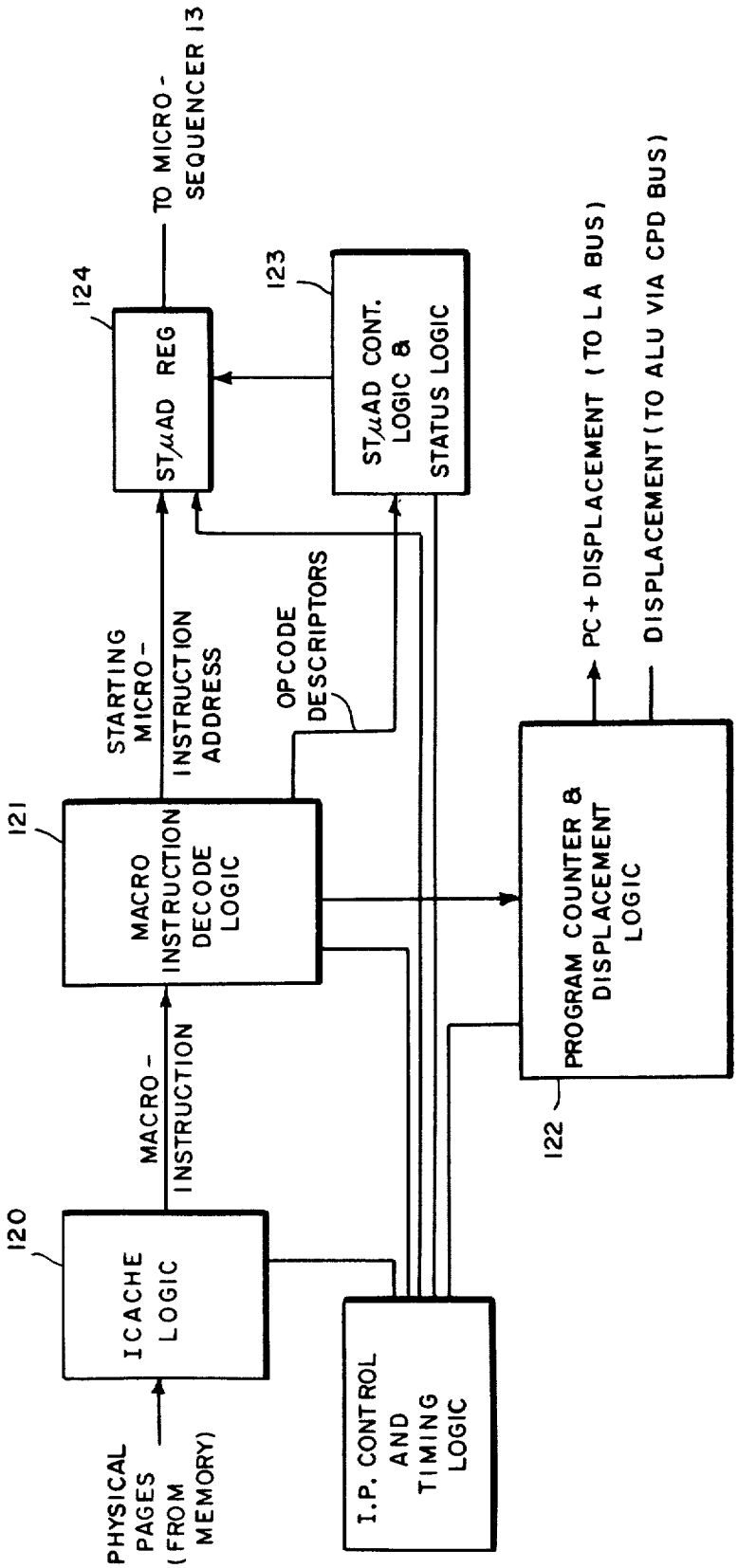

PHYSICAL PAGES (FROM MEMORY)
120
ICACHE LOGIC
MACRO-INSTRUCTION
121
MACRO INSTRUCTION DECODE LOGIC
STARTING MICRO-INSTRUCTION ADDRESS
124
ST/μAD REG
TO MICRO-SEQUENCER 13
OPCODE DESCRIPTORS
123
ST/μAD CONT. LOGIC & STATUS LOGIC
I.P. CONTROL AND TIMING LOGIC
122
PROGRAM COUNTER & DISPLACEMENT LOGIC
PC+DISPLACEMENT (TO LA BUS)
DISPLACEMENT (TO ALU VIA CPD BUS)

FIG.101

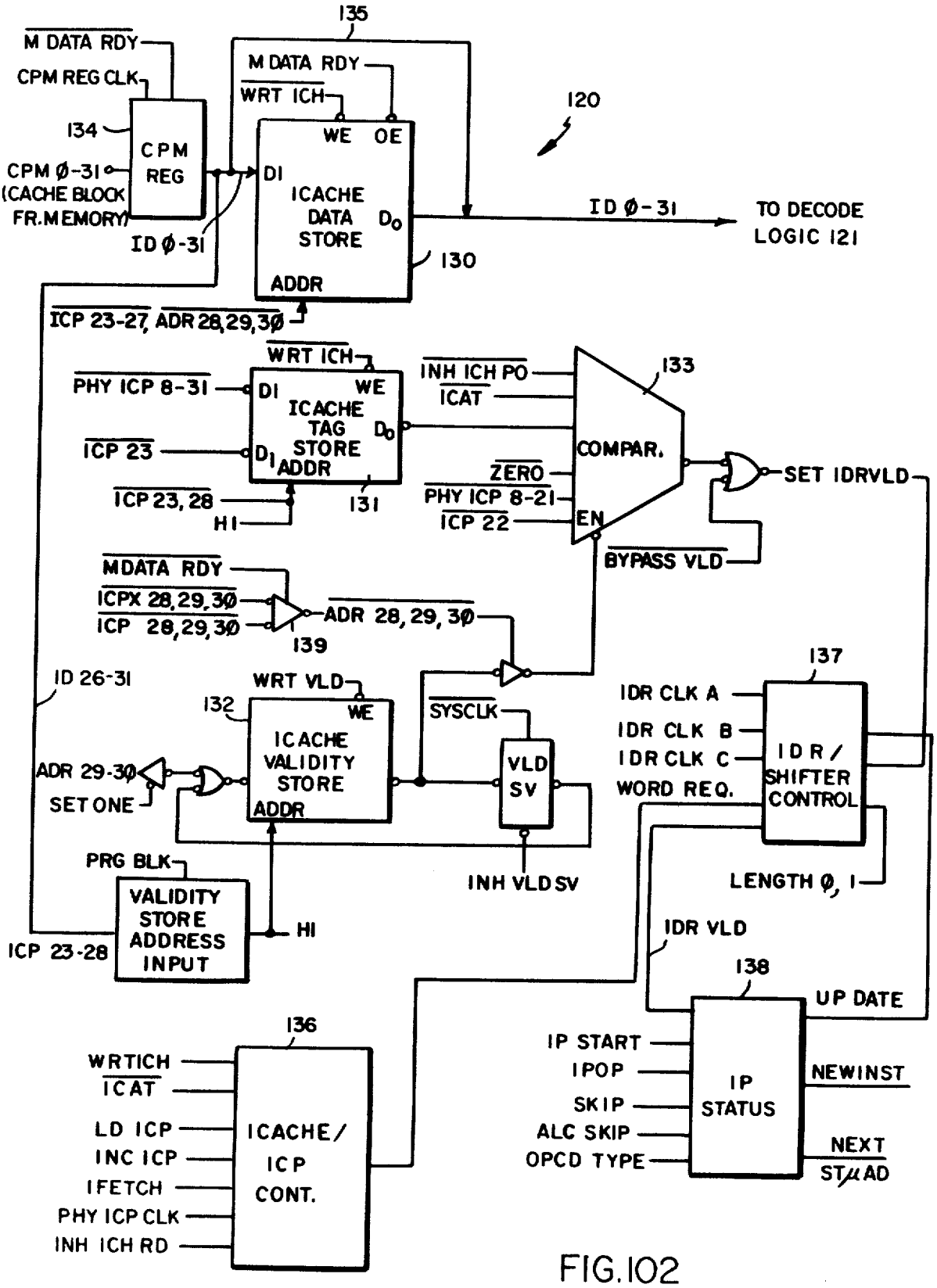

135
M DATA RDY
CPM REG CLK
134
CPM REG
CPM Ø-31
(CACHE BLOCK FR.MEMORY)
ID Ø-31
M DATA RDY
WRT ICH
WE
OE
DI
ICACHE DATA STORE
Do
ADDR
130
120
ID Ø-31
TO DECODE LOGIC 121
ICP 23-27, ADR 28,29,3Ø
WRT ICH
PHY ICP 8-31
DI
WE
ICACHE TAG STORE
Do
ICP 23
D1
ADDR
131
ICP 23, 28
HI
INH ICH PO
ICAT
133
COMPAR.
ZERO
PHY ICP 8-21
ICP 22
EN
SET IDRVLD
BYPASS VLD
MDATA RDY
ICPX 28,29,3Ø
ICP 28,29,3Ø
139
ADR 28, 29, 3Ø
ID 26-31
WRT VLD
132
WE
ICACHE VALIDITY STORE
ADDR
SYSCLK
VLD SV
ADR 29-3Ø
SET ONE
INH VLD SV
137
IDR CLK A
IDR CLK B
IDR CLK C
WORD REQ.
IDR / SHIFTER CONTROL
LENGTH Ø, 1
PRG BLK
VALIDITY STORE ADDRESS INPUT
HI
ICP 23-28
IDR VLD
138
UP DATE
136
WRTICH
ICAT
LD ICP
INC ICP
IFETCH
PHY ICP CLK
INH ICH RD
ICACHE/ ICP CONT.
IP START
IPOP
SKIP
ALC SKIP
OPCD TYPE
IP STATUS
NEWINST
NEXT
ST/μAD

FIG.102

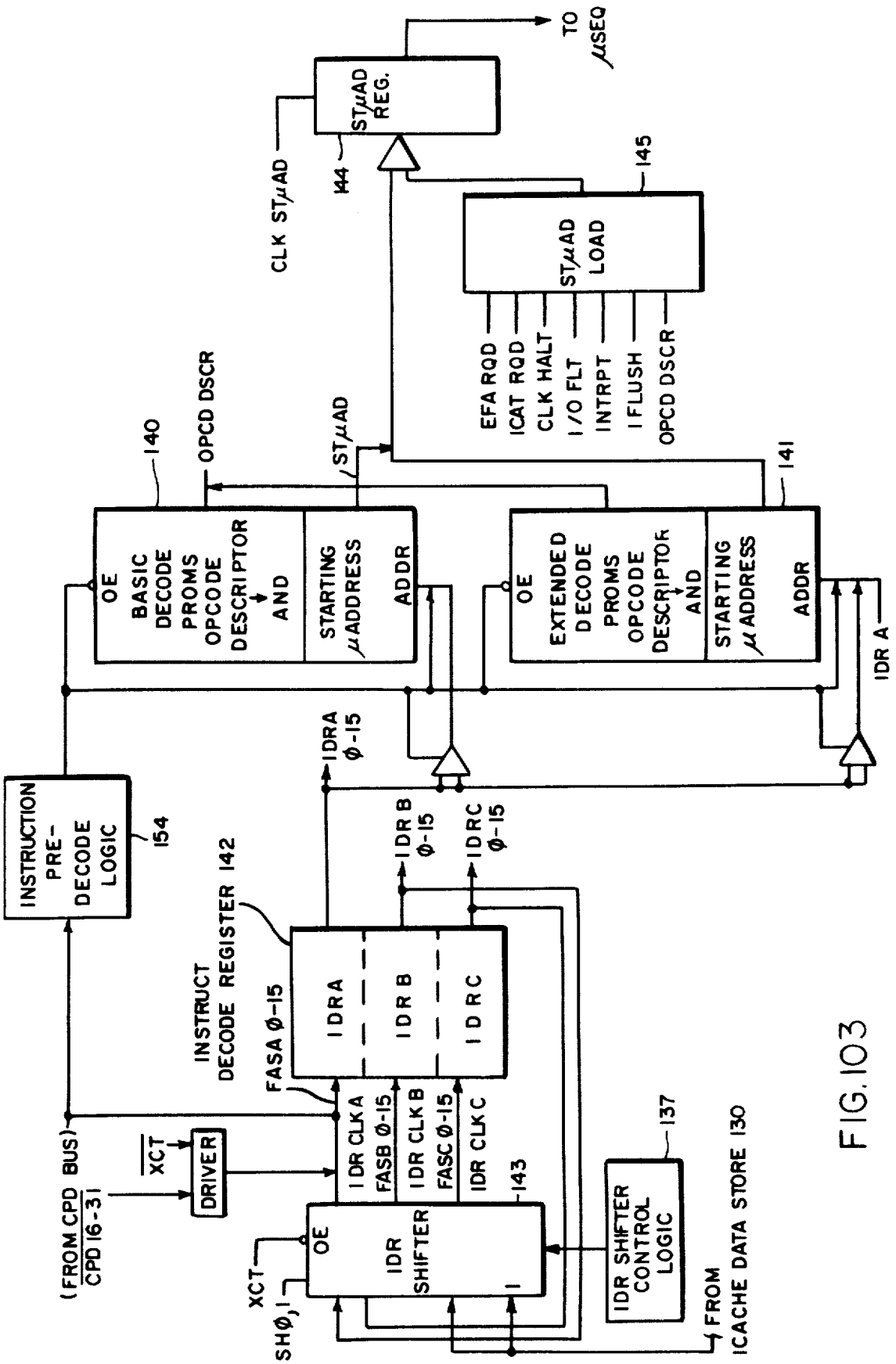

TO μSEQ
ST μAD REG.
144
CLK ST μAD
145
ST μAD LOAD
EFA RQD
ICAT RQD
CLK HALT
I/O FLT
INTRPT
I FLUSH
OPCD DSCR
140
OPCD DSCR
ST μAD
141
OE
BASIC DECODE PROMS OPCODE DESCRIPTOR AND
STARTING μADDRESS
ADDR
OE
EXTENDED DECODE PROMS OPCODE DESCRIPTOR AND
STARTING μADDRESS
ADDR
IDR A
IDRA Φ-15
INSTRUCTION PRE-DECODE LOGIC
154
IDR B Φ-15
IDR C Φ-15
INSTRUCT DECODE REGISTER 142
FASA Φ-15
IDR A
IDR B
IDR C
(FROM CPD BUS) CPD 16-31
XCT
DRIVER
IDR CLK A
FASB Φ-15
IDR CLK B
FASC Φ-15
IDR CLK C
143
137
XCT
SHΦ,I
OE
IDR SHIFTER
IDR SHIFTER CONTROL LOGIC
FROM ICACHE DATA STORE 130

FIG. 103

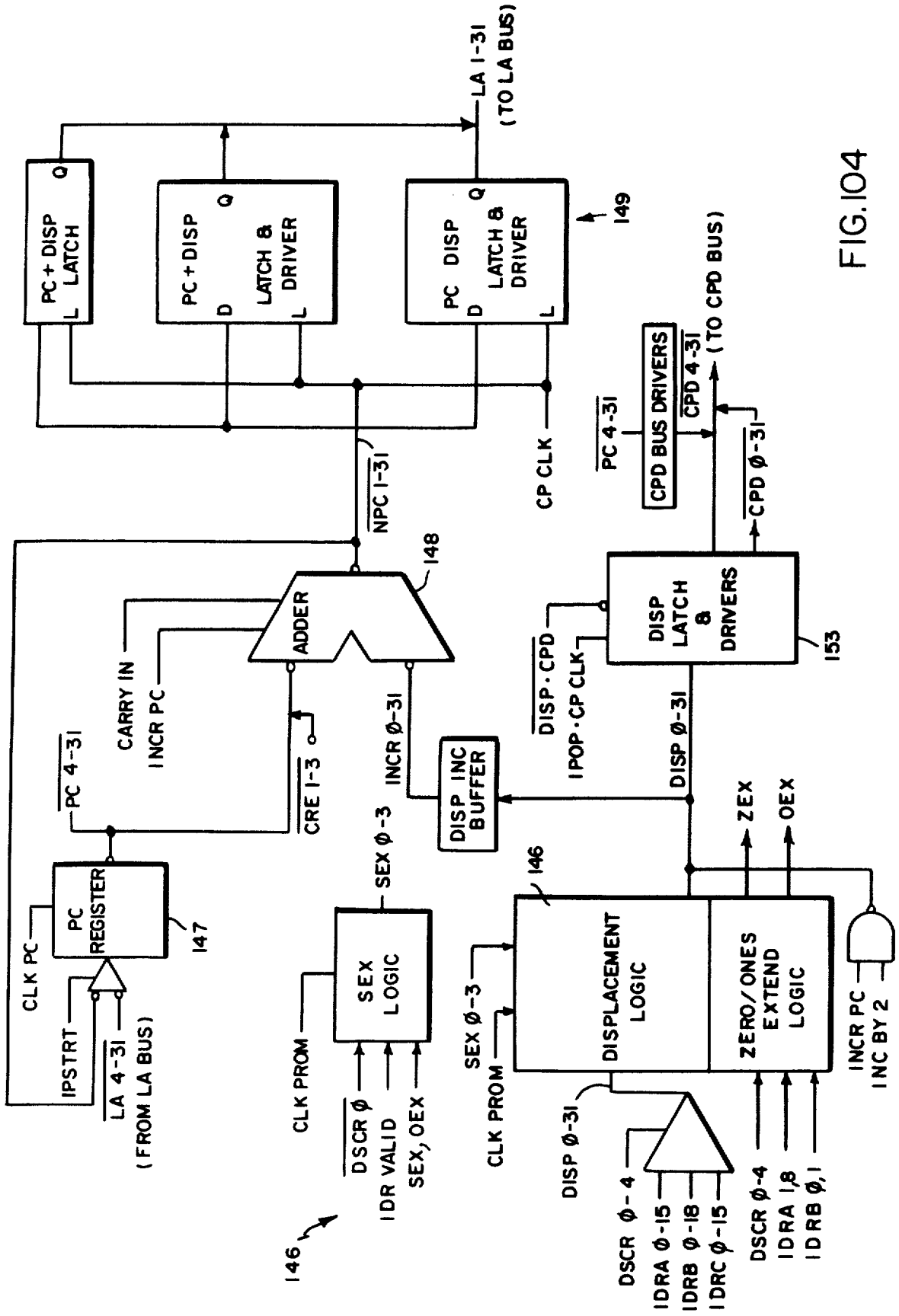

PC + DISP LATCH
PC + DISP LATCH & DRIVER
PC DISP LATCH & DRIVER
LA 1-31 (TO LA BUS)
149
CPD BUS DRIVERS
CPD 4-31 (TO CPD BUS)
PC 4-31
CPD Ø-31
CP CLK
NPC 1-31
148
ADDER
CARRY IN
INCR PC
CRE 1-3
INCR Ø-31
DISP LATCH & DRIVERS
153
DISP·CPD
IPOP·CP CLK
DISP Ø-31
DISP INC BUFFER
PC 4-31
PC REGISTER
147
CLK PC
IPSTRT
LA 4-31 (FROM LA BUS)
SEX Ø-3
SEX LOGIC
CLK PROM
DSCR Ø
IDR VALID
SEX, OEX
146
DISPLACEMENT LOGIC
ZERO/ONES EXTEND LOGIC
ZEX
OEX
INCR PC INC BY 2
DISP Ø-31
DSCR Ø-4
IDRA Ø-15
IDRB Ø-18
IDRC Ø-15
DSCR Ø-4
IDRA 1,8
IDRB Ø,1

FIG. 104

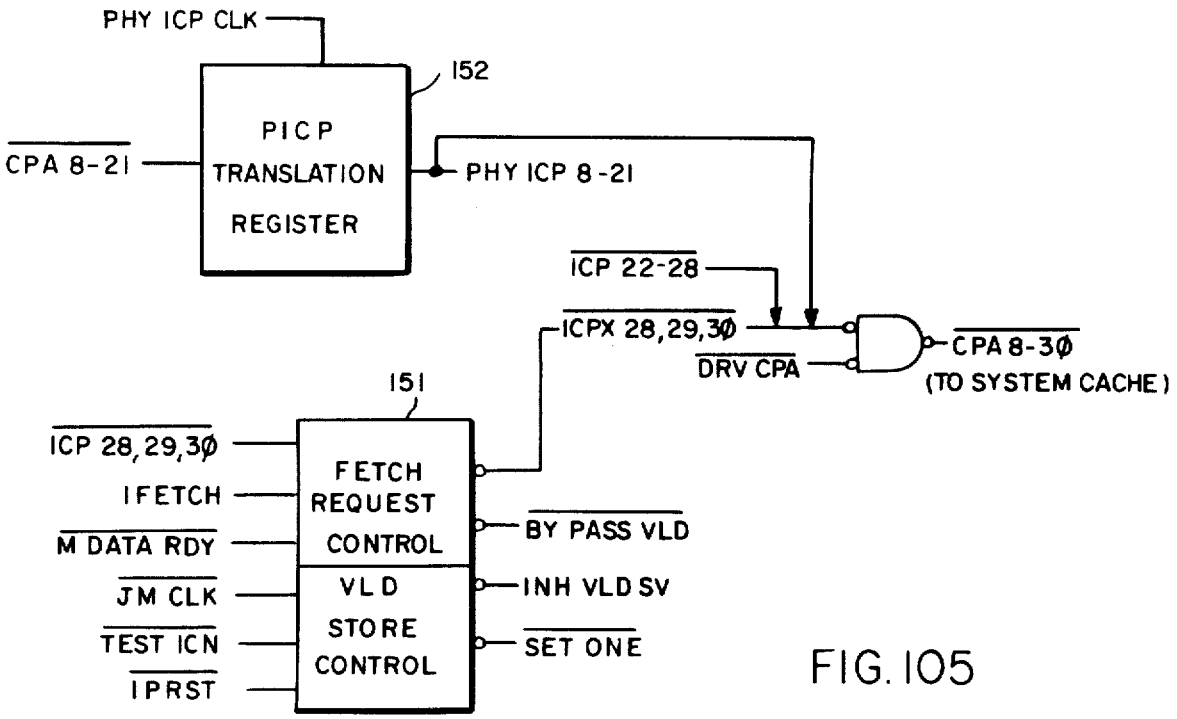
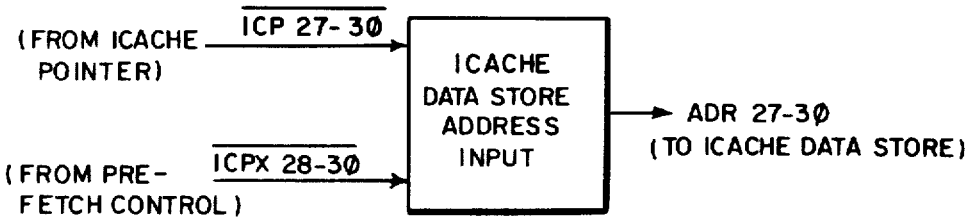


FIG. 105



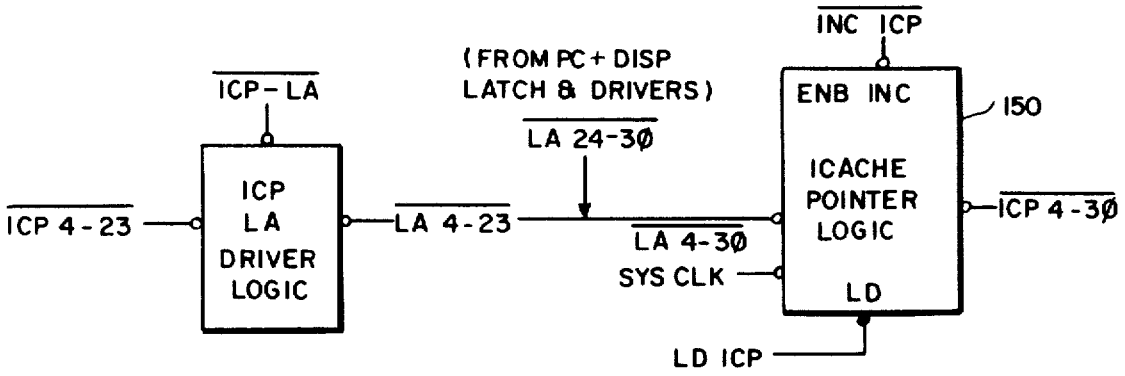
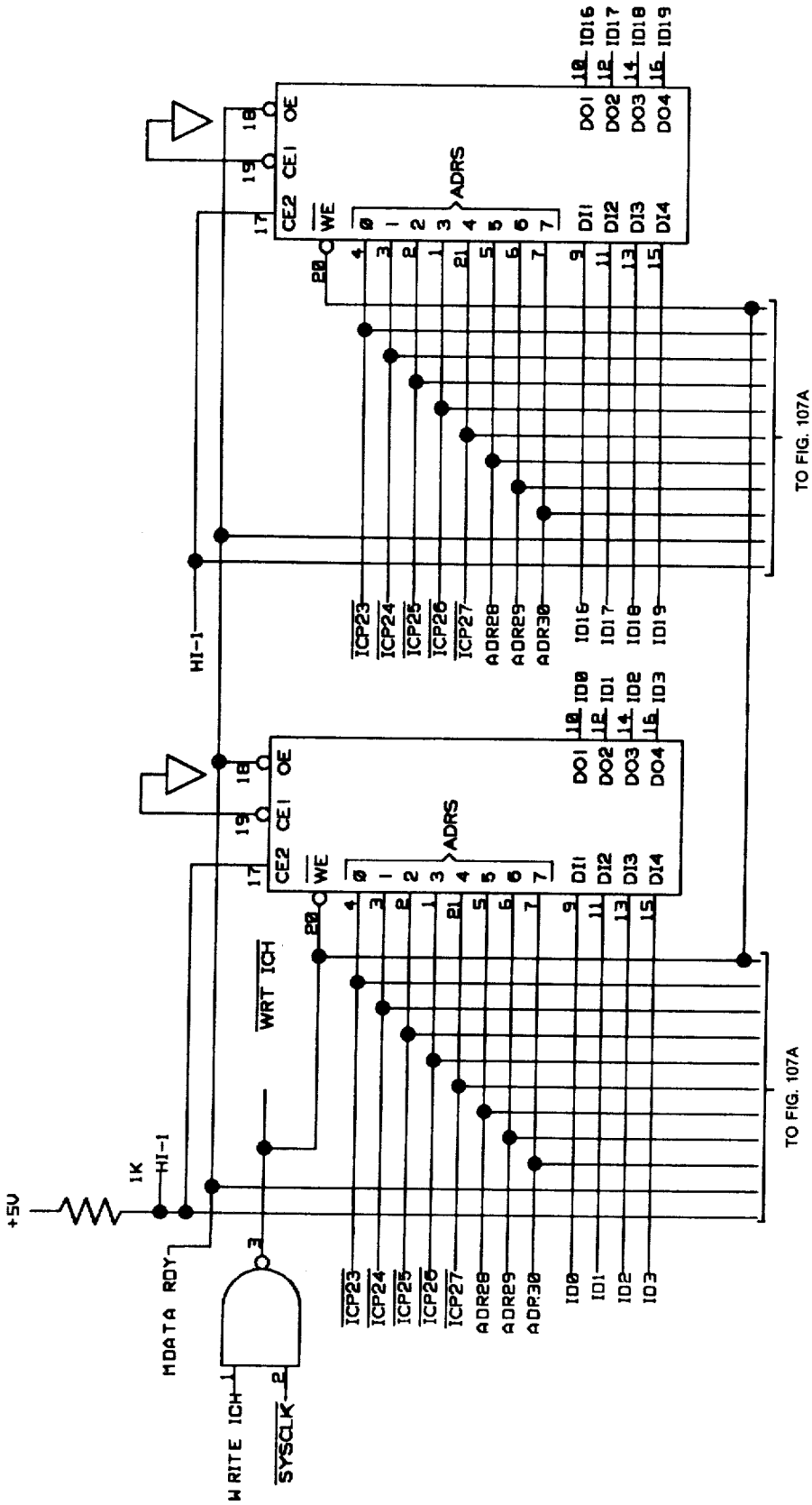


FIG. 106



ICACHE DATA STORE 130

FIG. 107

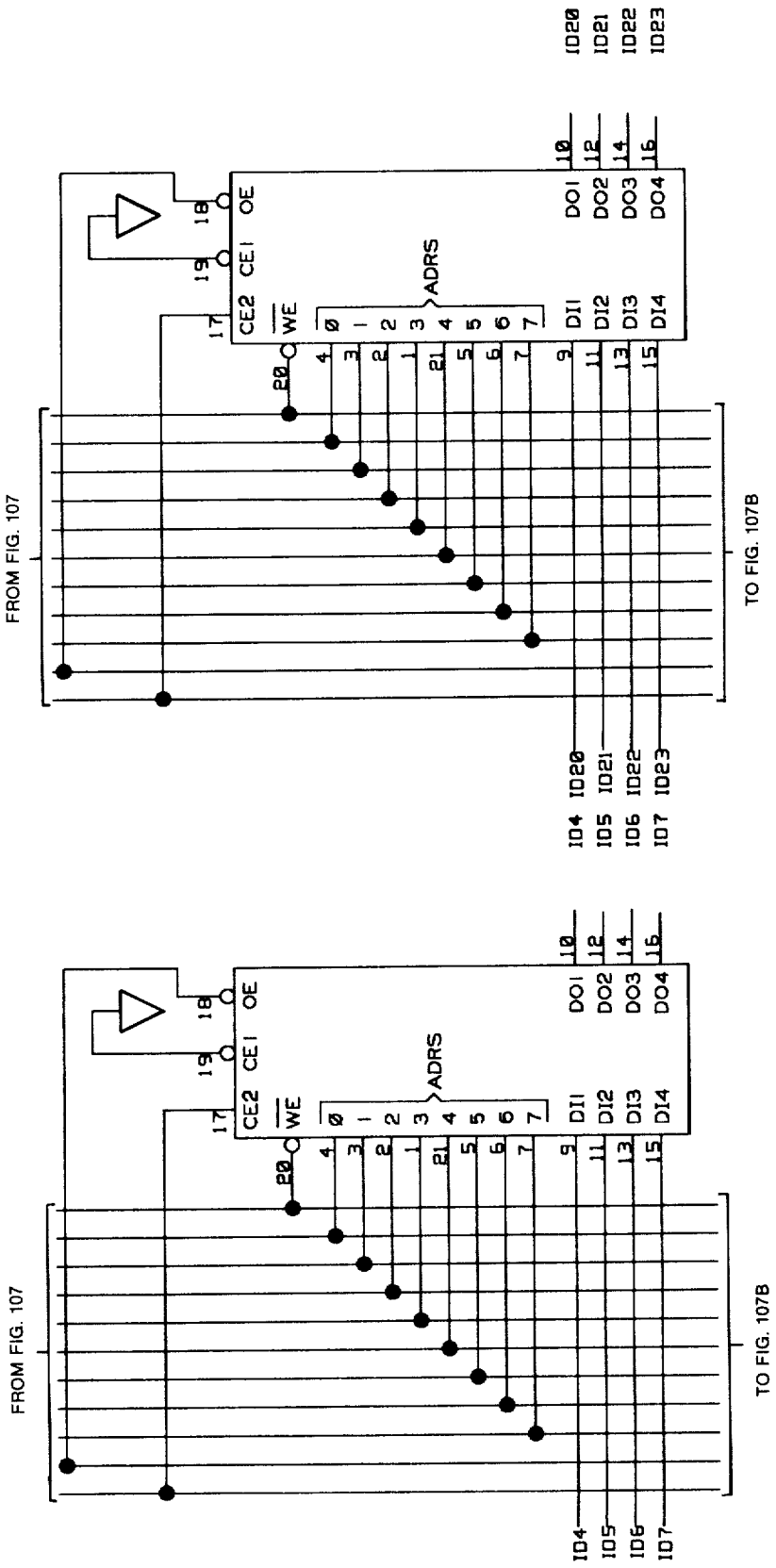

FROM FIG. 107
TO FIG. 107B
CE2
CE1
OE
WE
ADRS
DI1
DI2
DI3
DI4
DO1
DO2
DO3
DO4
ID4
ID5
ID6
ID7
ID20
ID21
ID22
ID23

FIG. 107A

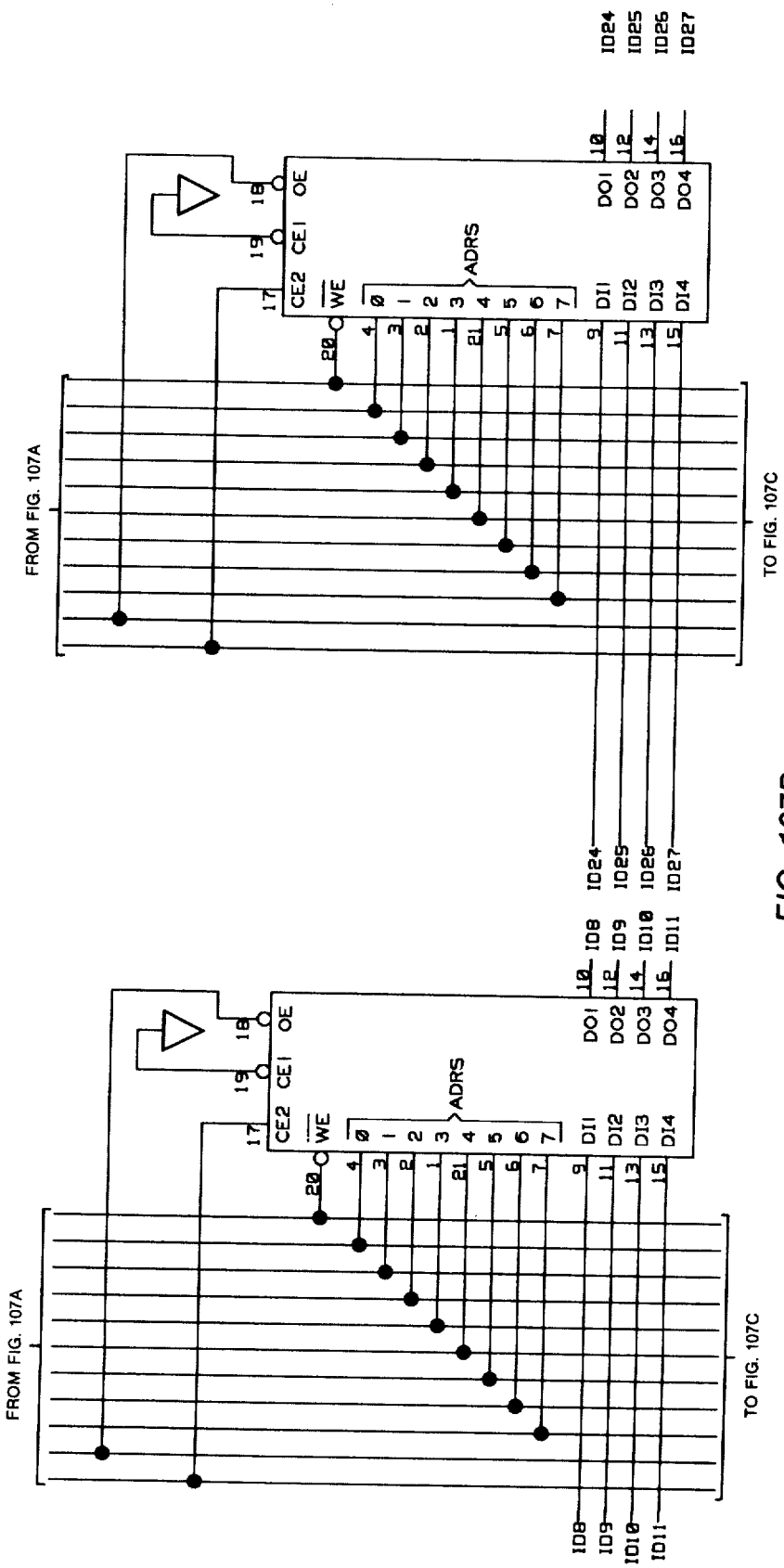

FROM FIG. 107A
TO FIG. 107C
CE2
CE1
OE
WE
ADRS
DO1
DO2
DO3
DO4
DI1
DI2
DI3
DI4
ID8
ID9
ID10
ID11
ID24
ID25
ID26
ID27

FIG. 107B

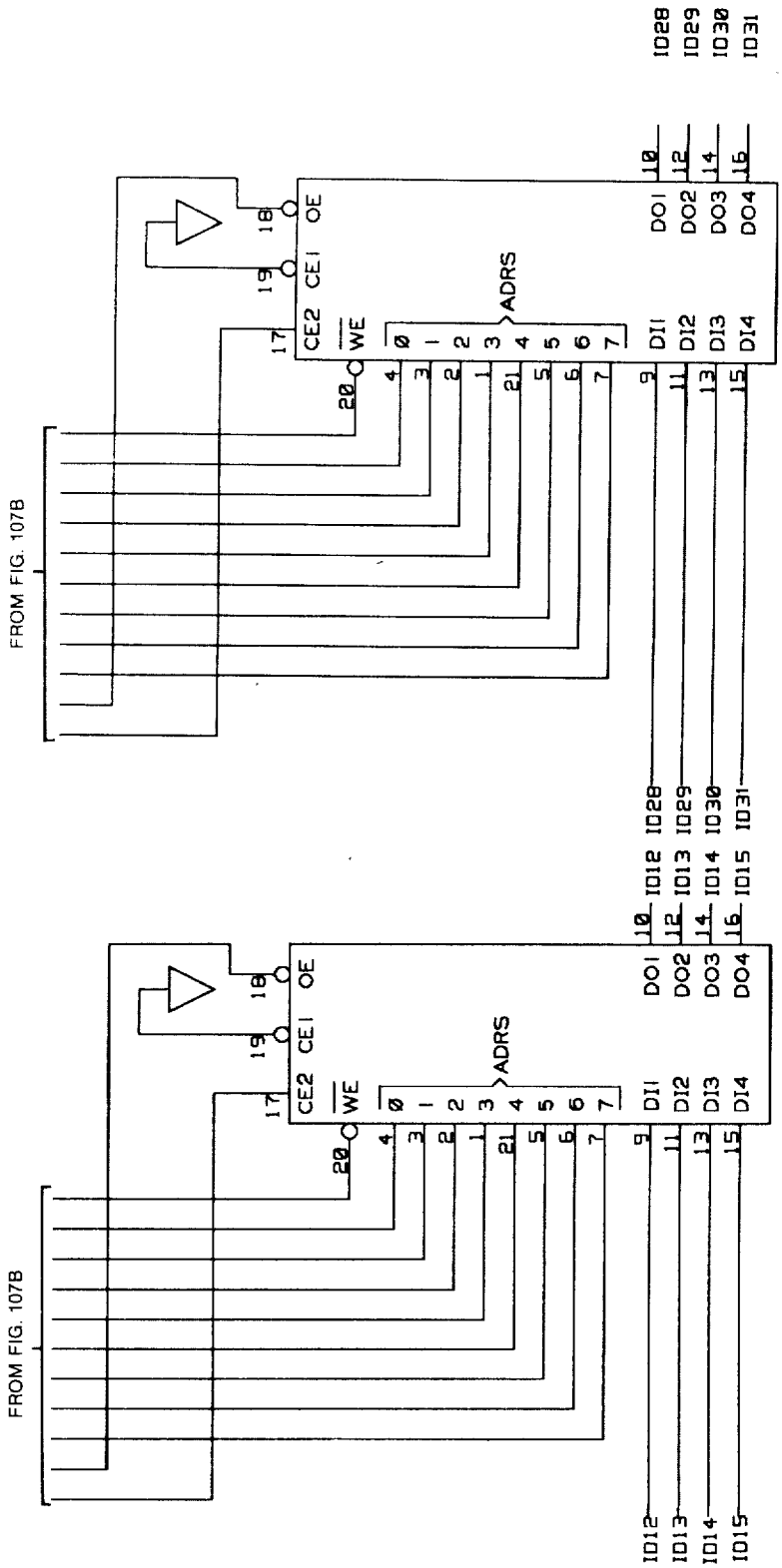
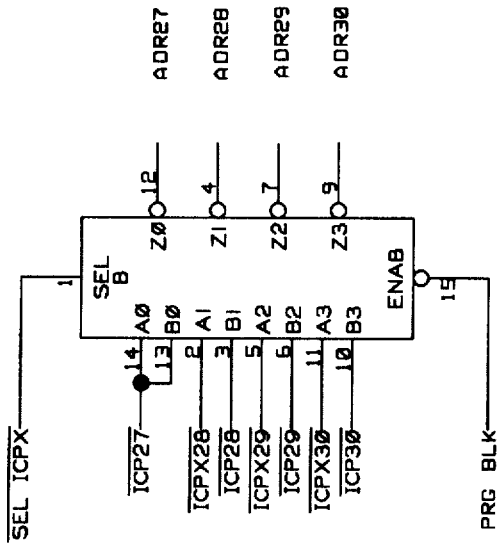
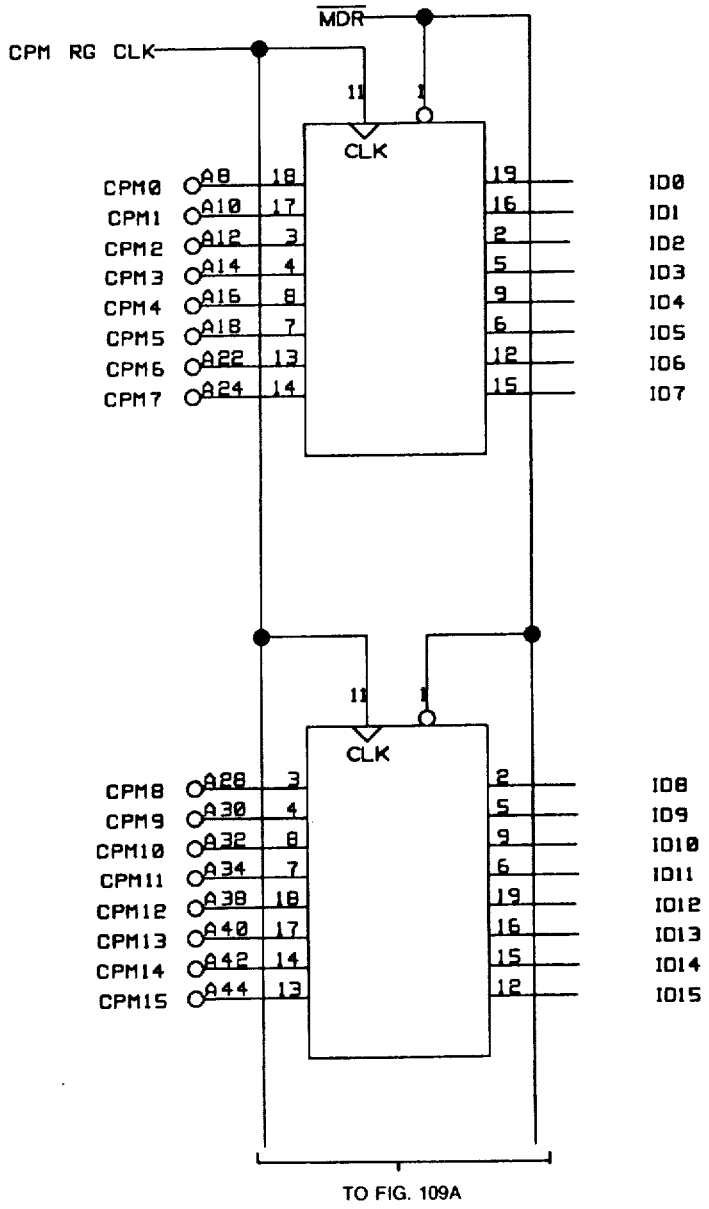

FROM FIG. 107B
CE2
CE1
OE
WE
ADRS
DI1
DI2
DI3
DI4
DO1
DO2
DO3
DO4
ID12
ID13
ID14
ID15
ID28
ID29
ID30
ID31
FROM FIG. 107B
CE2
CE1
OE
WE
ADRS
DI1
DI2
DI3
DI4
DO1
DO2
DO3
DO4
ID12
ID13
ID14
ID15
ID28
ID29
ID30
ID31

FIG. 107C



ICACHE DATA STORE ADDRESS INPUT

FIG. 108



CPM REGISTER 134

FIG. 109

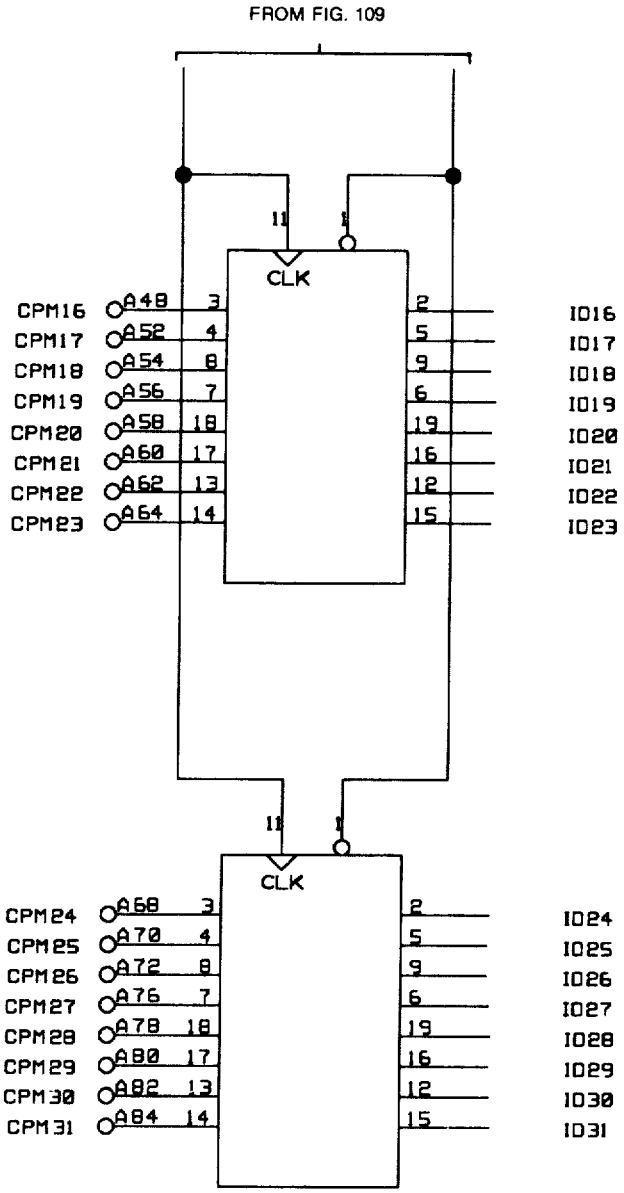

FROM FIG. 109
CLK
CPM16
CPM17
CPM18
CPM19
CPM20
CPM21
CPM22
CPM23
A48
A52
A54
A56
A58
A60
A62
A64
ID16
ID17
ID18
ID19
ID20
ID21
ID22
ID23
CLK
CPM24
CPM25
CPM26
CPM27
CPM28
CPM29
CPM30
CPM31
A68
A70
A72
A76
A78
A80
A82
A84
ID24
ID25
ID26
ID27
ID28
ID29
ID30
ID31

FIG. 109A

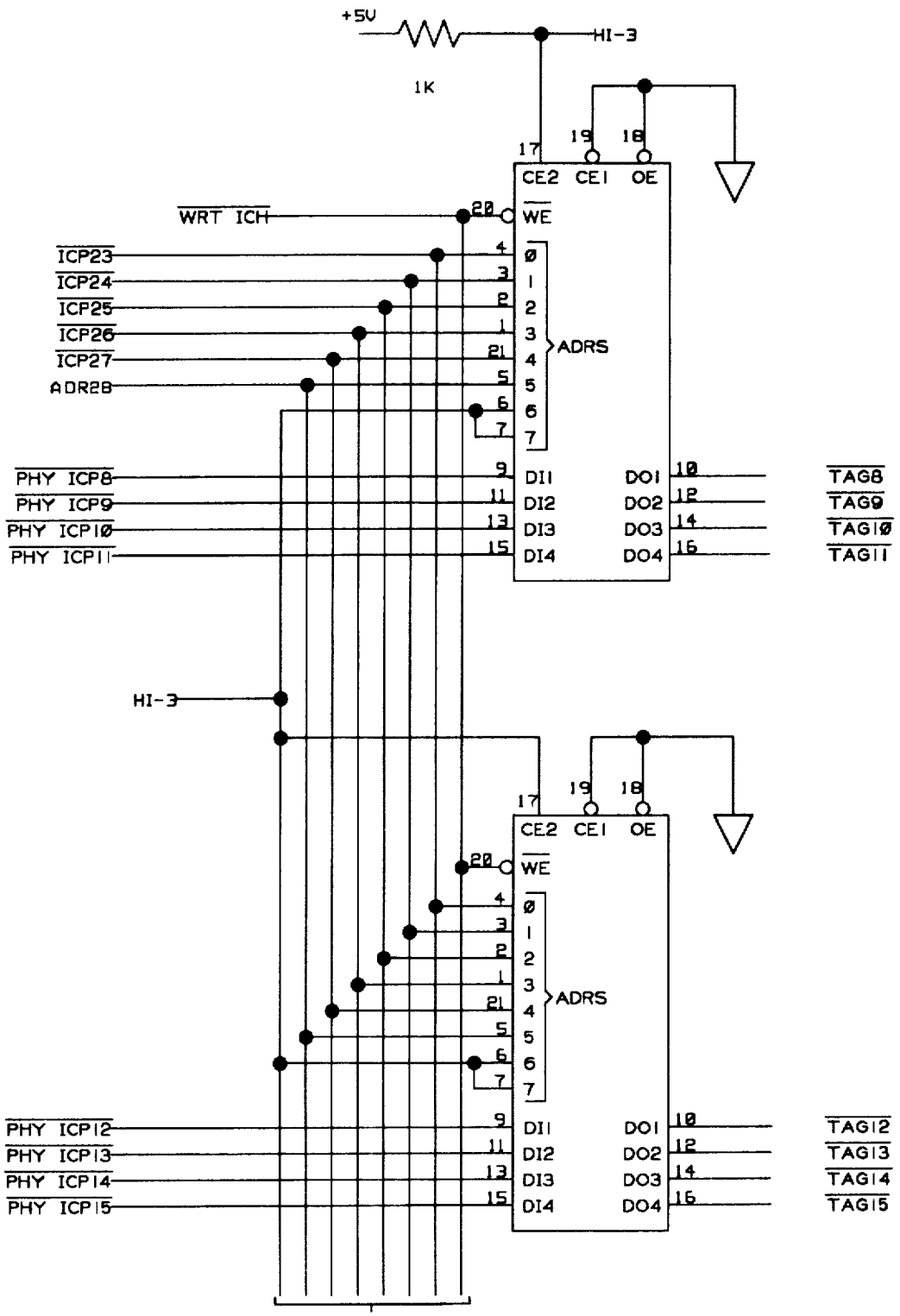

+5V
1K
HI-3
17
19
18
CE2
CE1
OE
WRT ICH
20
WE
ICP23
ICP24
ICP25
ICP26
ICP27
ADR28
ADRS
PHY ICP8
PHY ICP9
PHY ICP10
PHY ICP11
DI1
DI2
DI3
DI4
DO1
DO2
DO3
DO4
TAG8
TAG9
TAG10
TAG11
HI-3
PHY ICP12
PHY ICP13
PHY ICP14
PHY ICP15
TAG12
TAG13
TAG14
TAG15
TO FIG. 109C

FIG. 109B

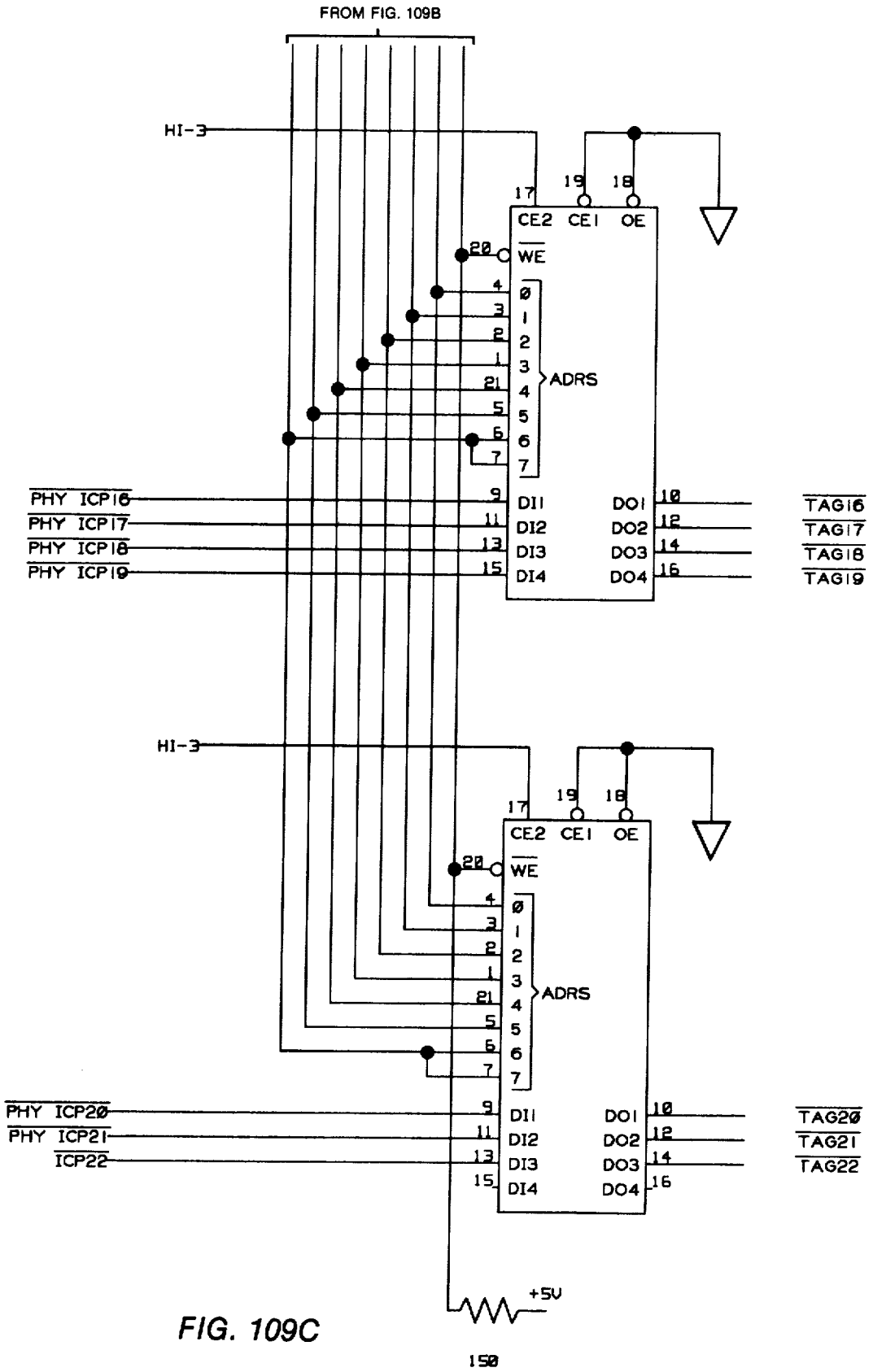
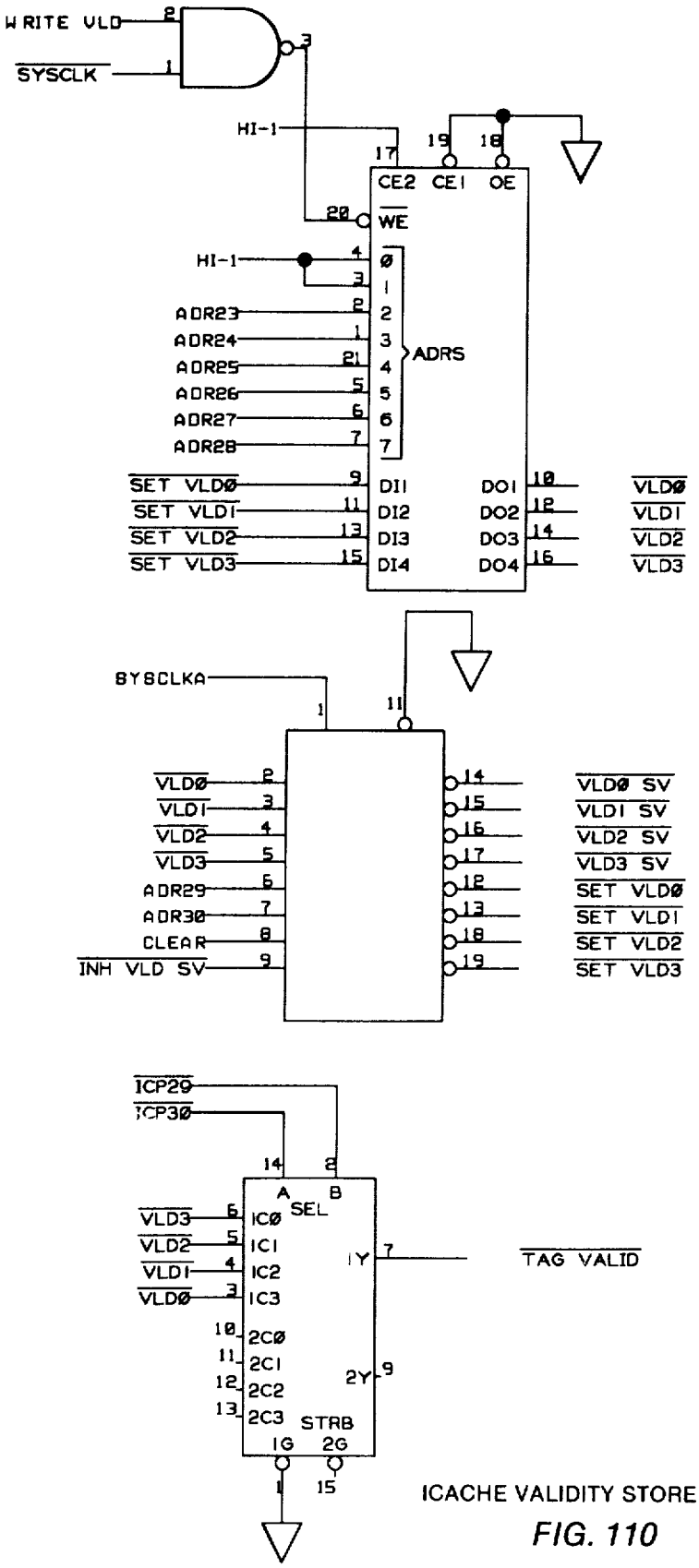
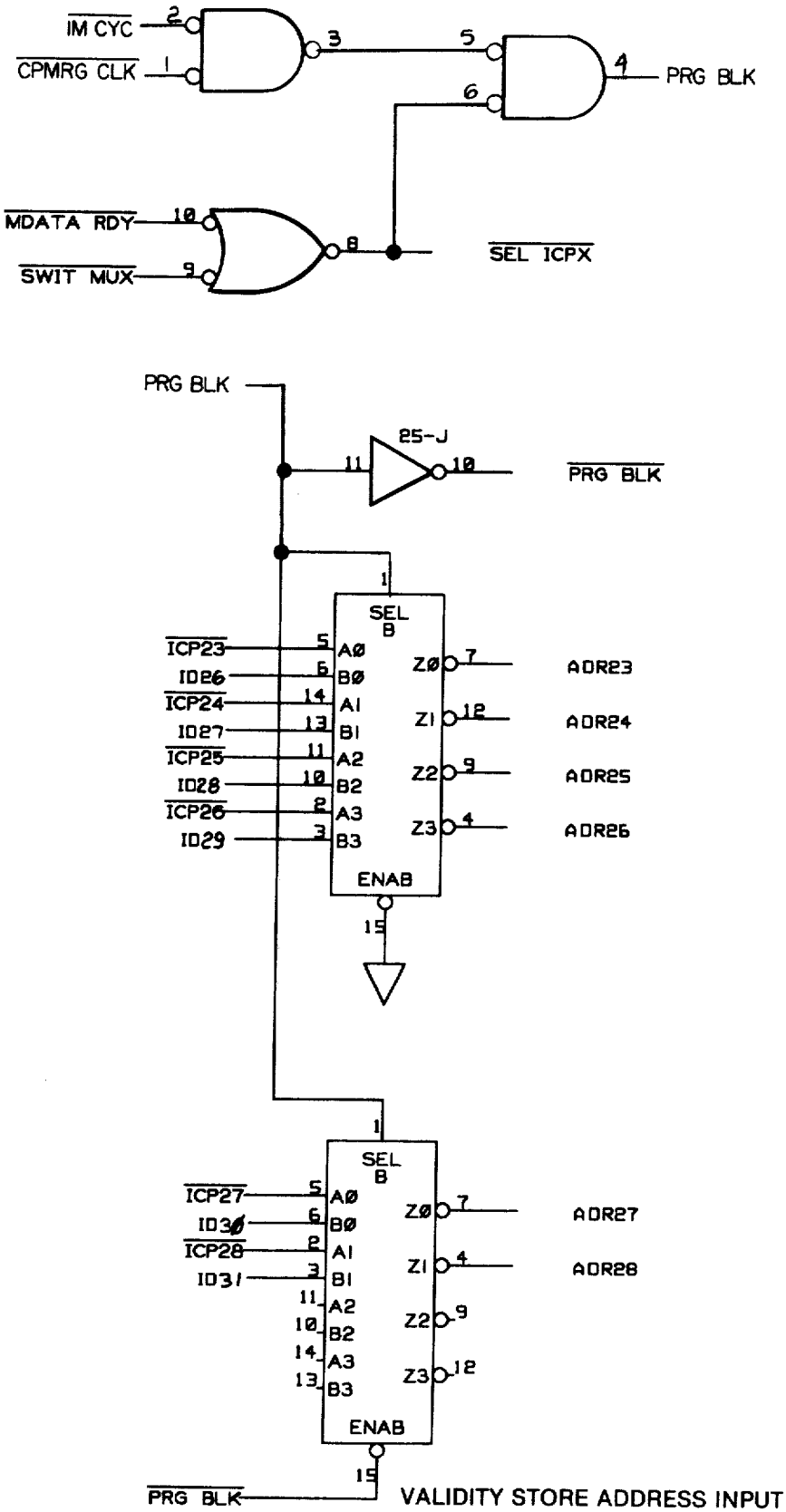

FROM FIG. 109B
HI-3
CE2
CE1
OE
WE
ADRS
DI1
DI2
DI3
DI4
DO1
DO2
DO3
DO4
PHY ICP16
PHY ICP17
PHY ICP18
PHY ICP19
TAG16
TAG17
TAG18
TAG19
HI-3
PHY ICP20
PHY ICP21
ICP22
TAG20
TAG21
TAG22
+5V

FIG. 109C



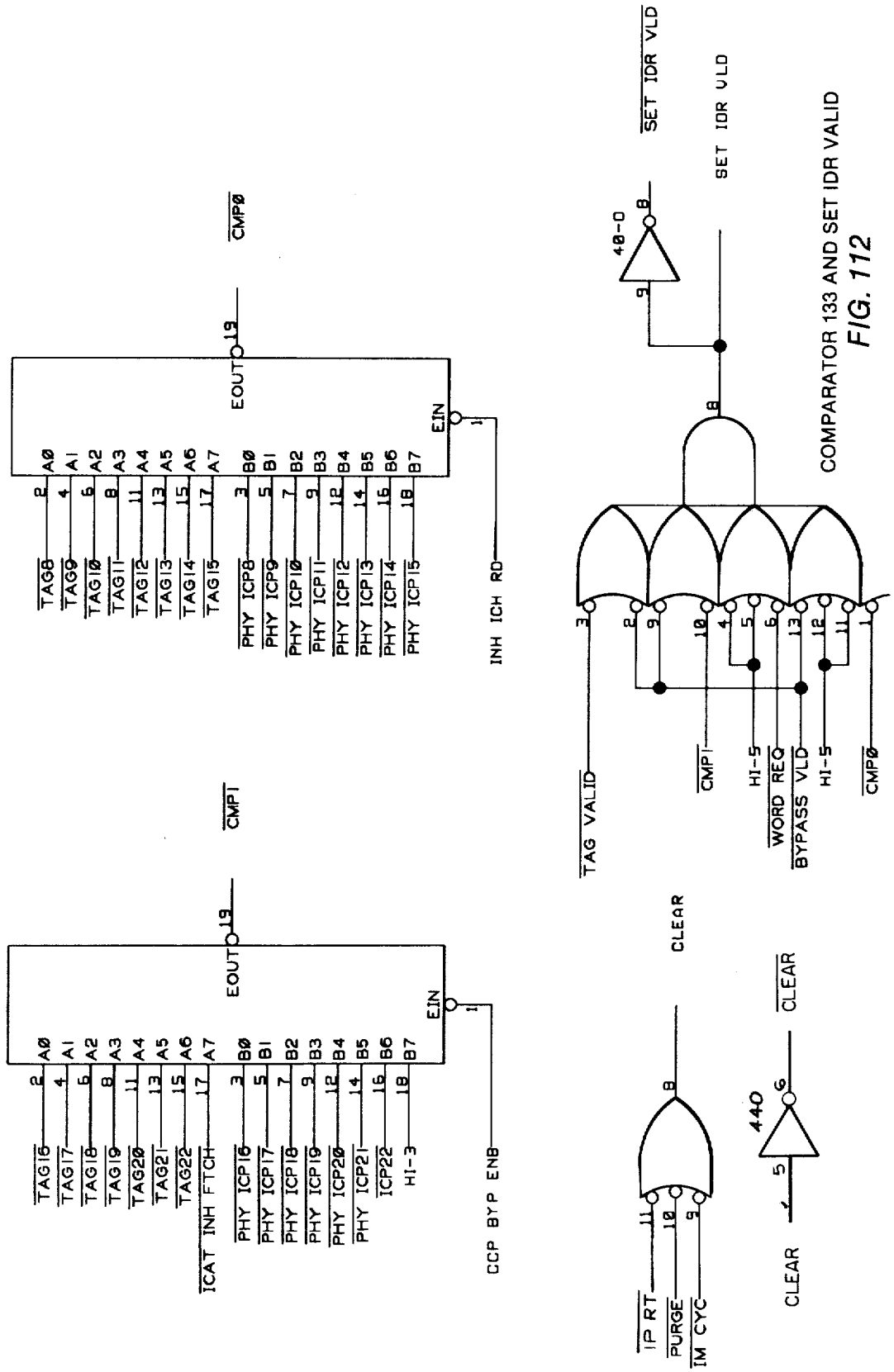
ICACHE VALIDITY STORE 132

FIG. 110



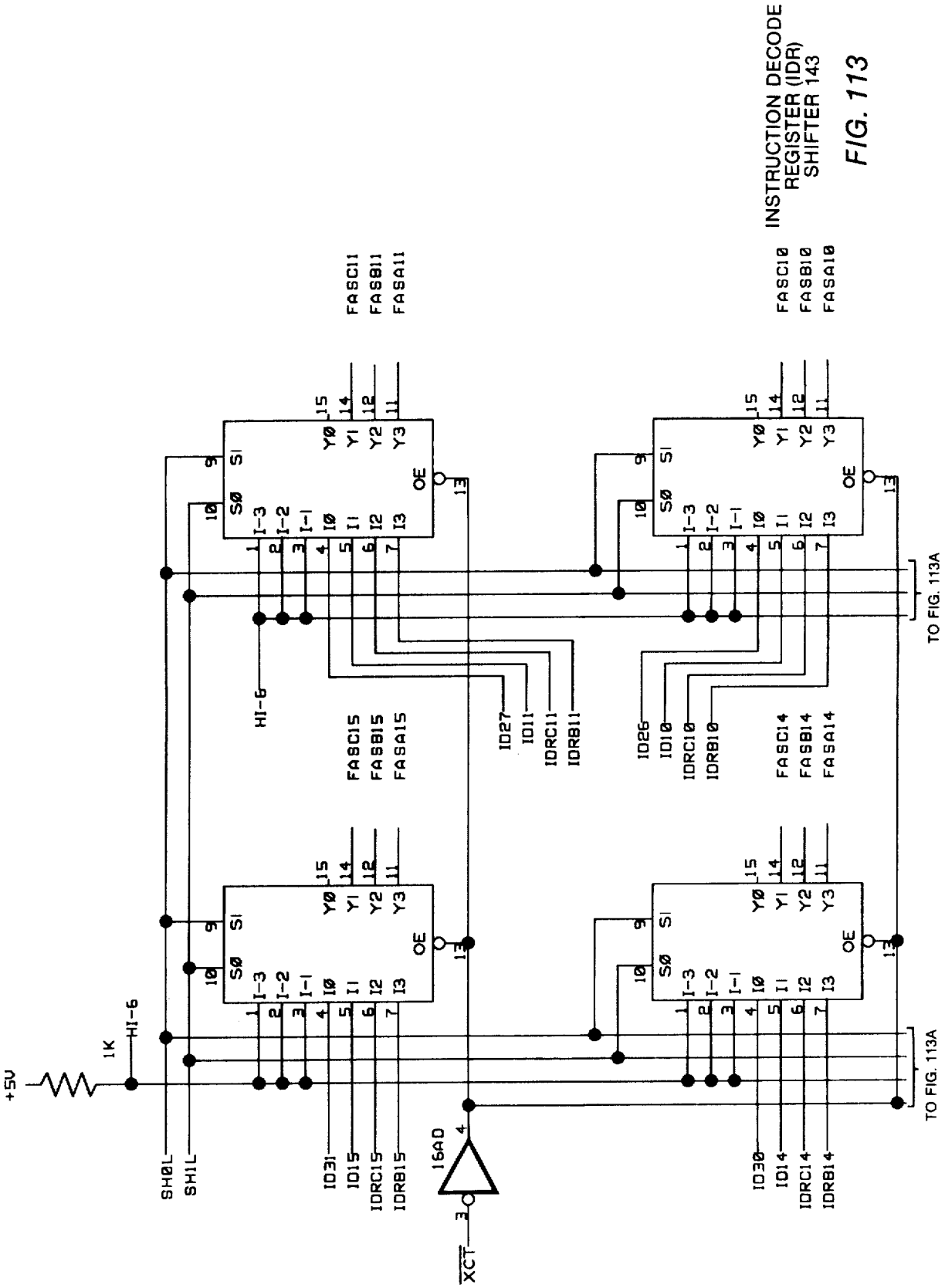
VALIDITY STORE ADDRESS INPUT

FIG. 111



COMPARATOR 133 AND SET IDR VALID

FIG. 112


INSTRUCTION DECODE REGISTER (IDR) SHIFTER 143
FIG. 113
FASC11
FASB11
FASA11
FASC10
FASB10
FASA10
FASC15
FASB15
FASA15
FASC14
FASB14
FASA14
ID27
ID11
IDRC11
IDRB11
ID26
ID10
IDRC10
IDRB10
HI-6
+5V
1K
SH0L
SH1L
ID31
ID15
IDRC15
IDRB15
16AD
XCT
ID30
ID14
IDRC14
IDRB14
TO FIG. 113A
TO FIG. 113A

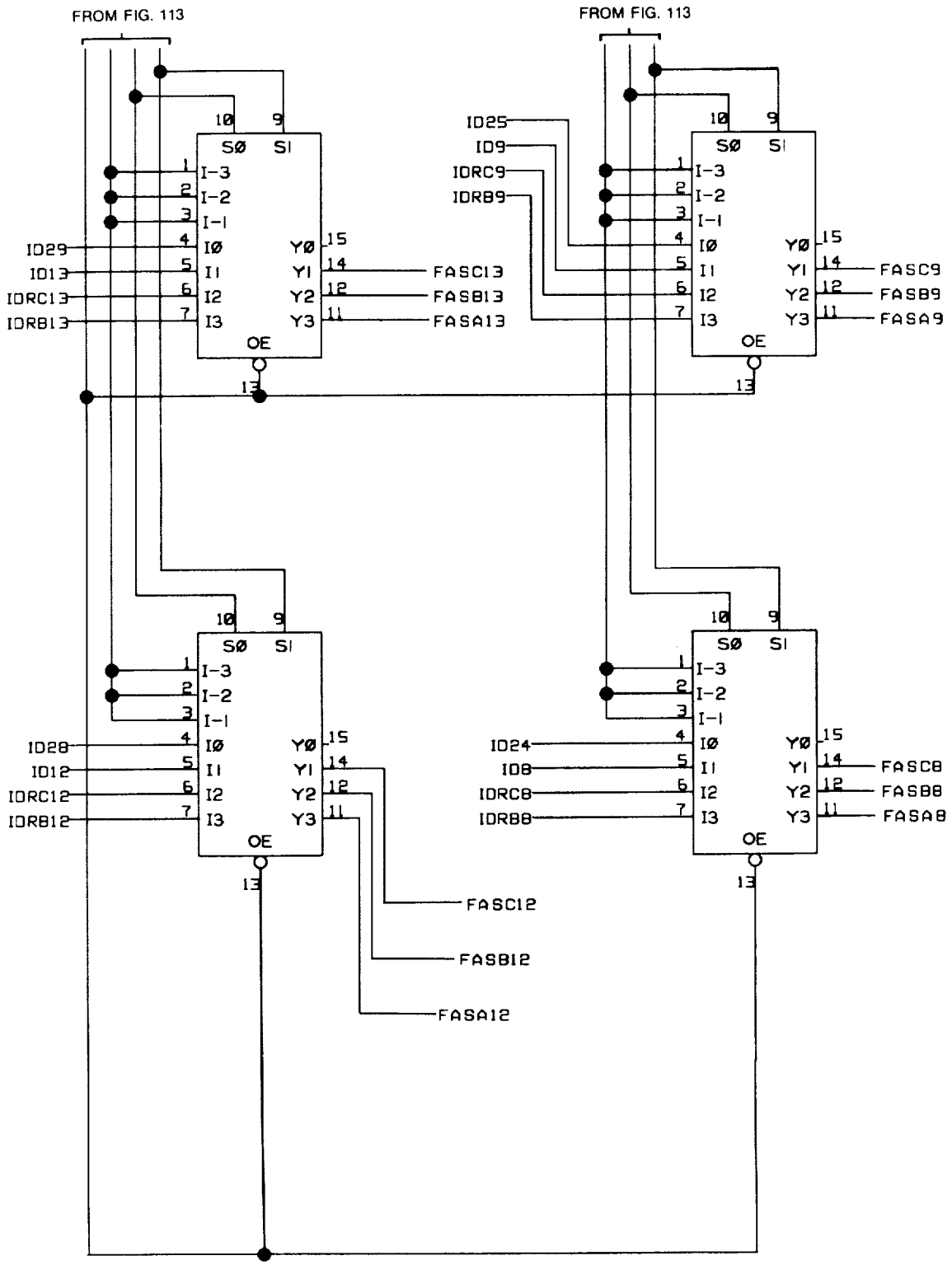

FROM FIG. 113
FROM FIG. 113
S0
S1
I-3
I-2
I-1
I0
I1
I2
I3
Y0
Y1
Y2
Y3
OE
ID29
ID13
IDRC13
IDRB13
FASC13
FASB13
FASA13
ID25
ID9
IDRC9
IDRB9
FASC9
FASB9
FASA9
ID28
ID12
IDRC12
IDRB12
FASC12
FASB12
FASA12
ID24
ID8
IDRC8
IDRB8
FASC8
FASB8
FASA8

FIG. 113A

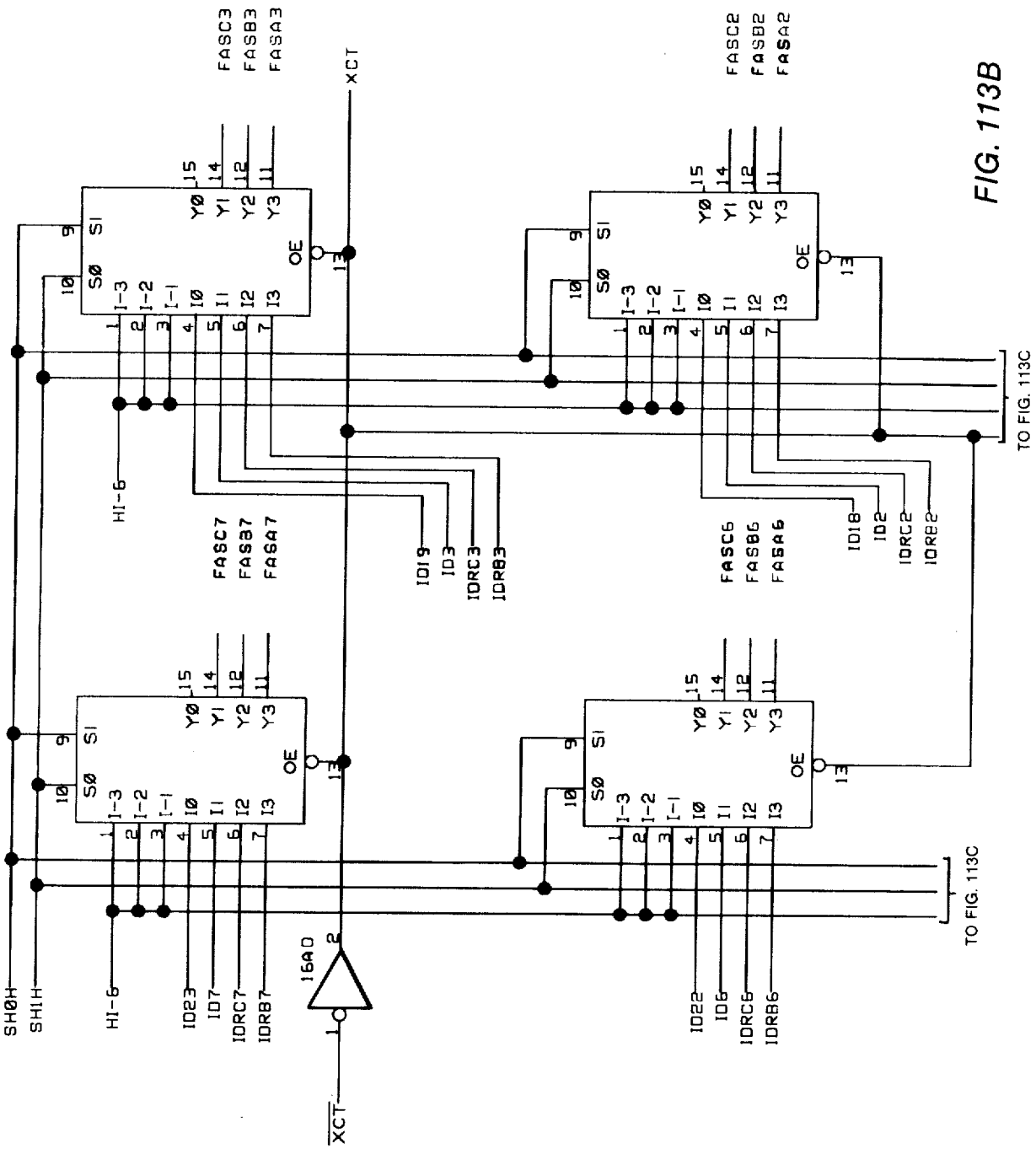

FASC3
FASB3
FASA3
XCT
FASC2
FASB2
FASA2
SI
SØ
OE
Y0
Y1
Y2
Y3
I-3
I-2
I-1
IØ
I1
I2
I3
HI-6
FASC7
FASB7
FASA7
ID19
ID3
IDRC3
IDRB3
FASC6
FASB6
FASA6
ID18
ID2
IDRC2
IDRB2
TO FIG. 113C
SHØH
SH1H
ID23
ID7
IDRC7
IDRB7
16AD
XCT
ID22
ID6
IDRC6
IDRB6
TO FIG. 113C

FIG. 113B

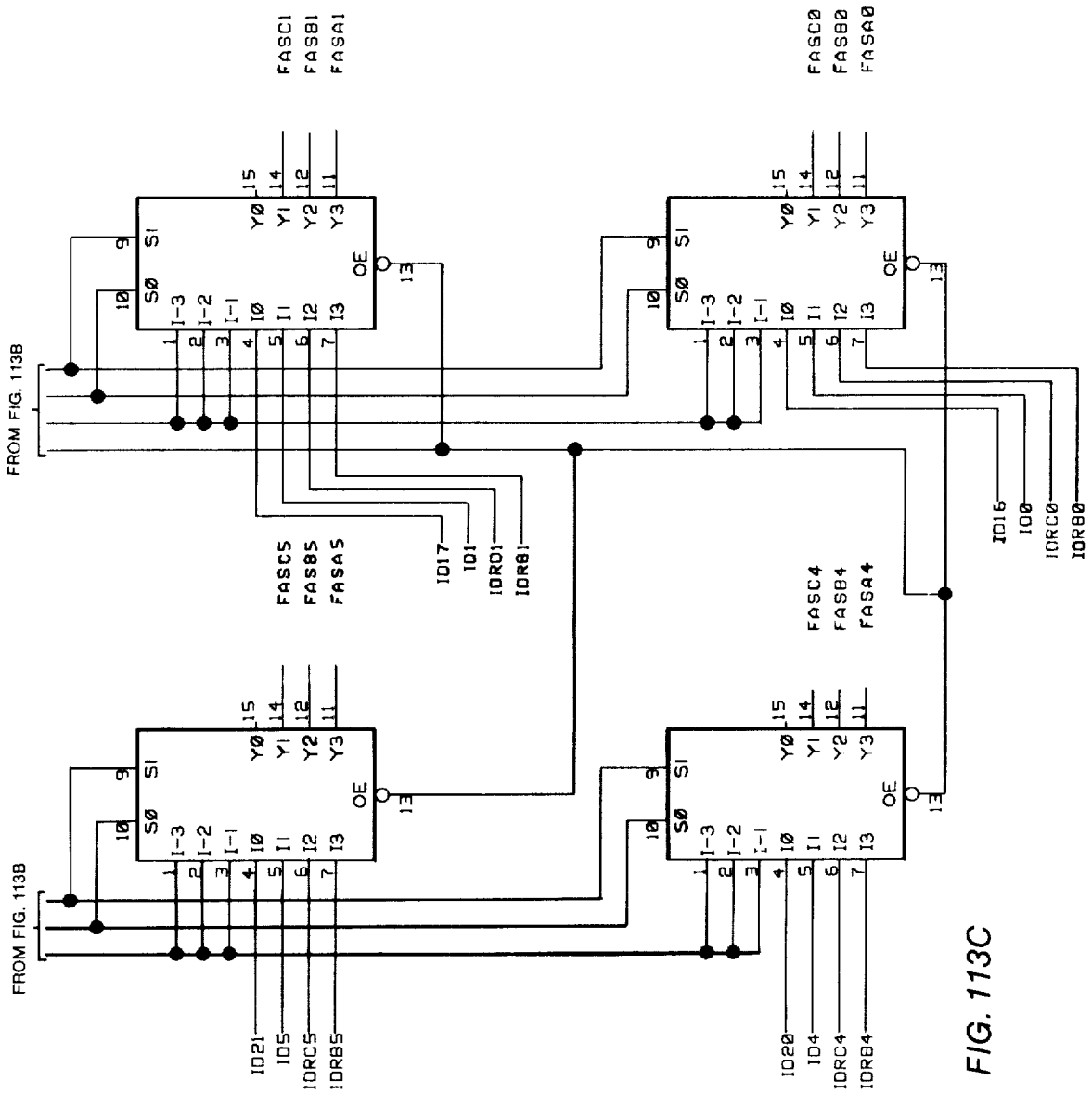

FROM FIG. 113B
FROM FIG. 113B
FASC1
FASB1
FASA1
FASC0
FASB0
FASA0
FASC5
FASB5
FASA5
FASC4
FASB4
FASA4
ID17
ID1
IDRD1
IDRB1
ID16
ID0
IDRC0
IDRB0
ID21
ID5
IDRC5
IDRB5
ID20
ID4
IDRC4
IDRB4
S0
S1
OE
Y0
Y1
Y2
Y3
I-3
I-2
I-1
I0
I1
I2
I3

FIG. 113C

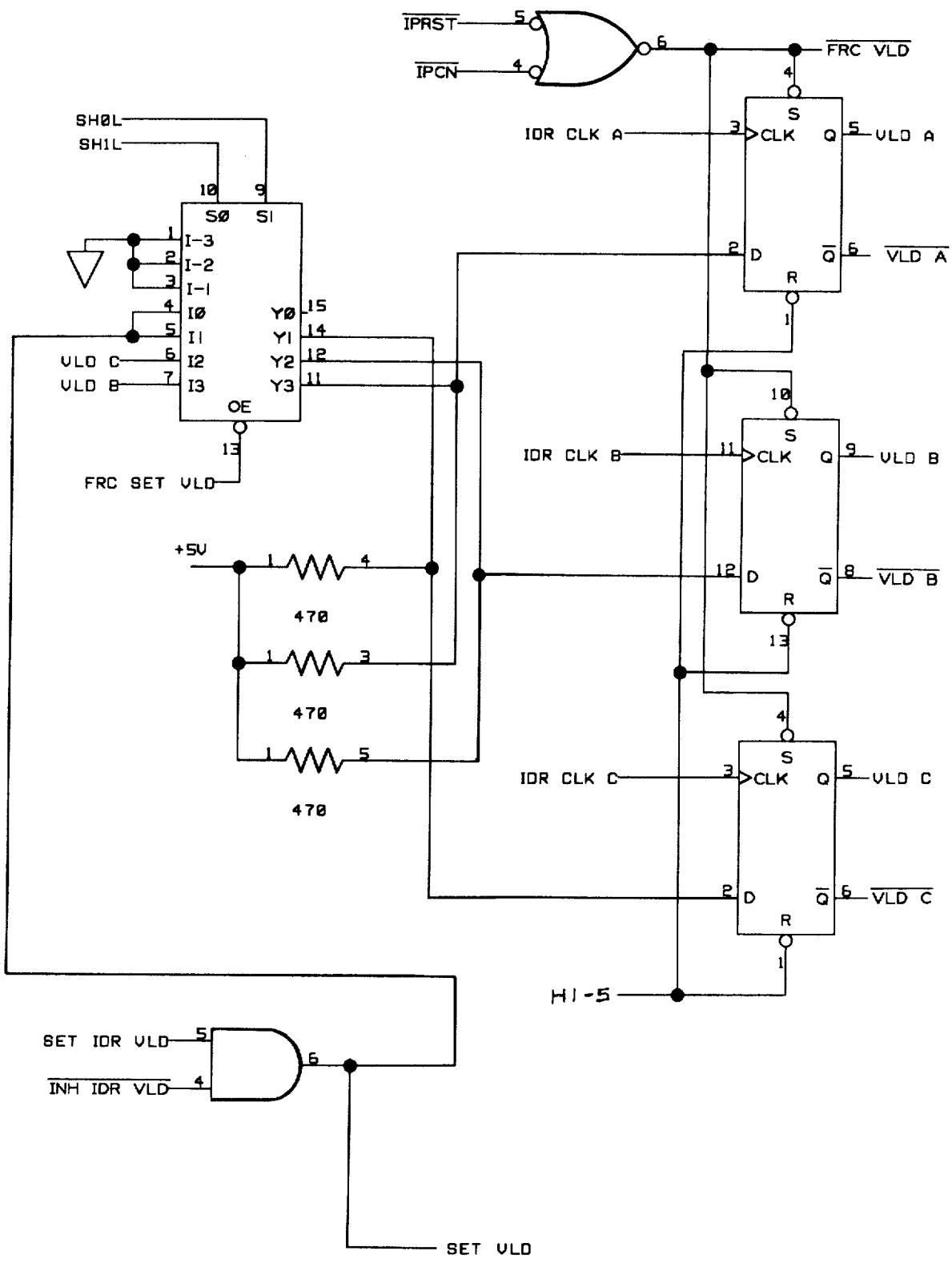

IPRST
IPCN
FRC VLD
SH0L
SH1L
IDR CLK A
VLD A
VLD A
S0
S1
I-3
I-2
I-1
I0
I1
I2
I3
Y0
Y1
Y2
Y3
OE
VLD C
VLD B
FRC SET VLD
IDR CLK B
VLD B
VLD B
+5V
470
470
470
IDR CLK C
VLD C
VLD C
HI-5
SET IDR VLD
INH IDR VLD
SET VLD

FIG. 114

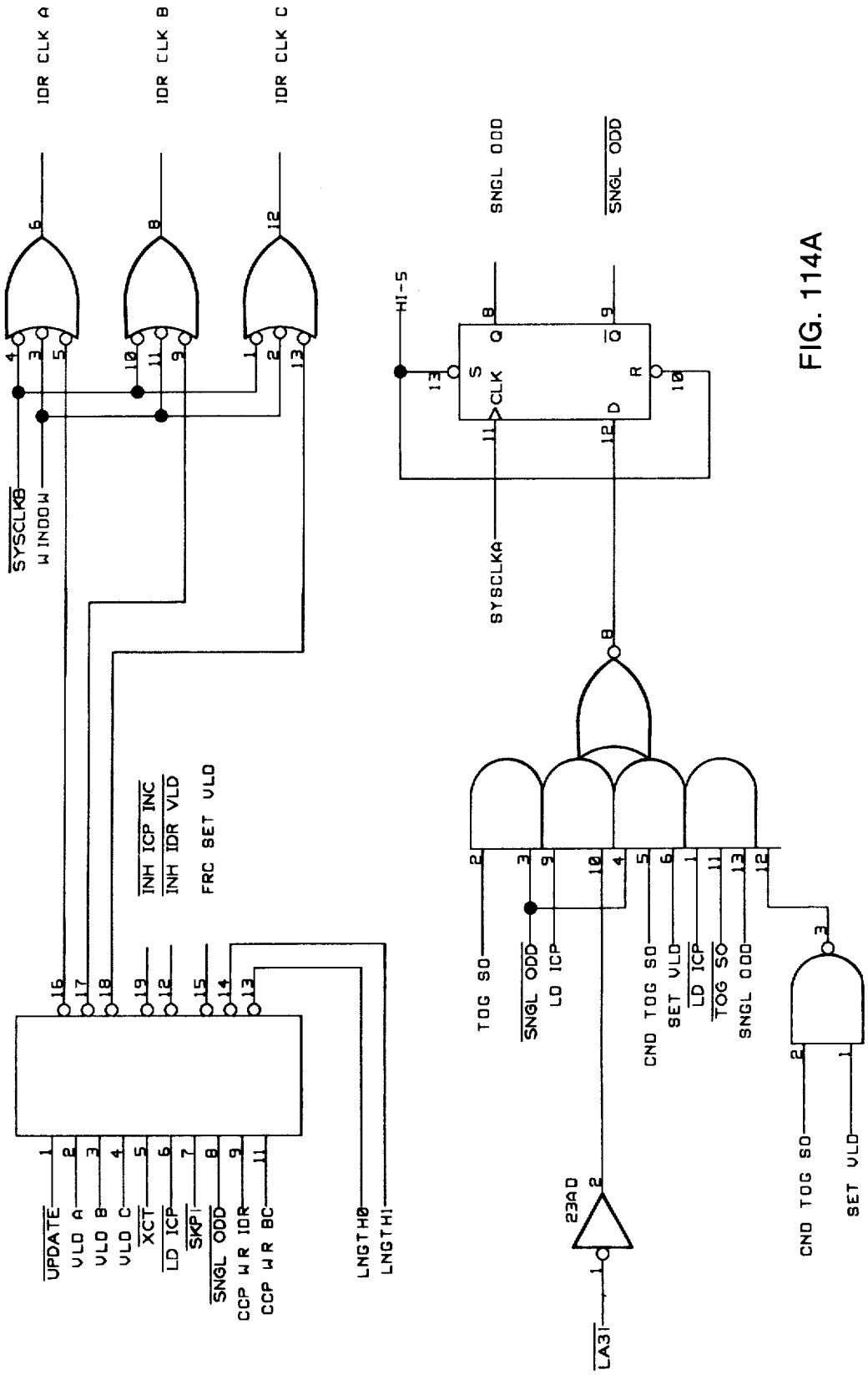

IDR CLK A
IDR CLK B
IDR CLK C
SYSCLKB
WINDOW
INH ICP INC
INH IDR VLD
FRC SET VLD
UPDATE
VLD A
VLD B
VLD C
XCT
LD ICP
SKP1
SNGL ODD
CCP WR IDR
CCP WR BC
LNGTH0
LNGTH1
HI-5
SNGL ODD
SNGL ODD
SYSCLKA
TOG SO
SNGL ODD
LD ICP
CND TOG SO
SET VLD
LD ICP
TOG SO
SNGL ODD
23AD
LA31
CND TOG SO
SET VLD

FIG. 114A

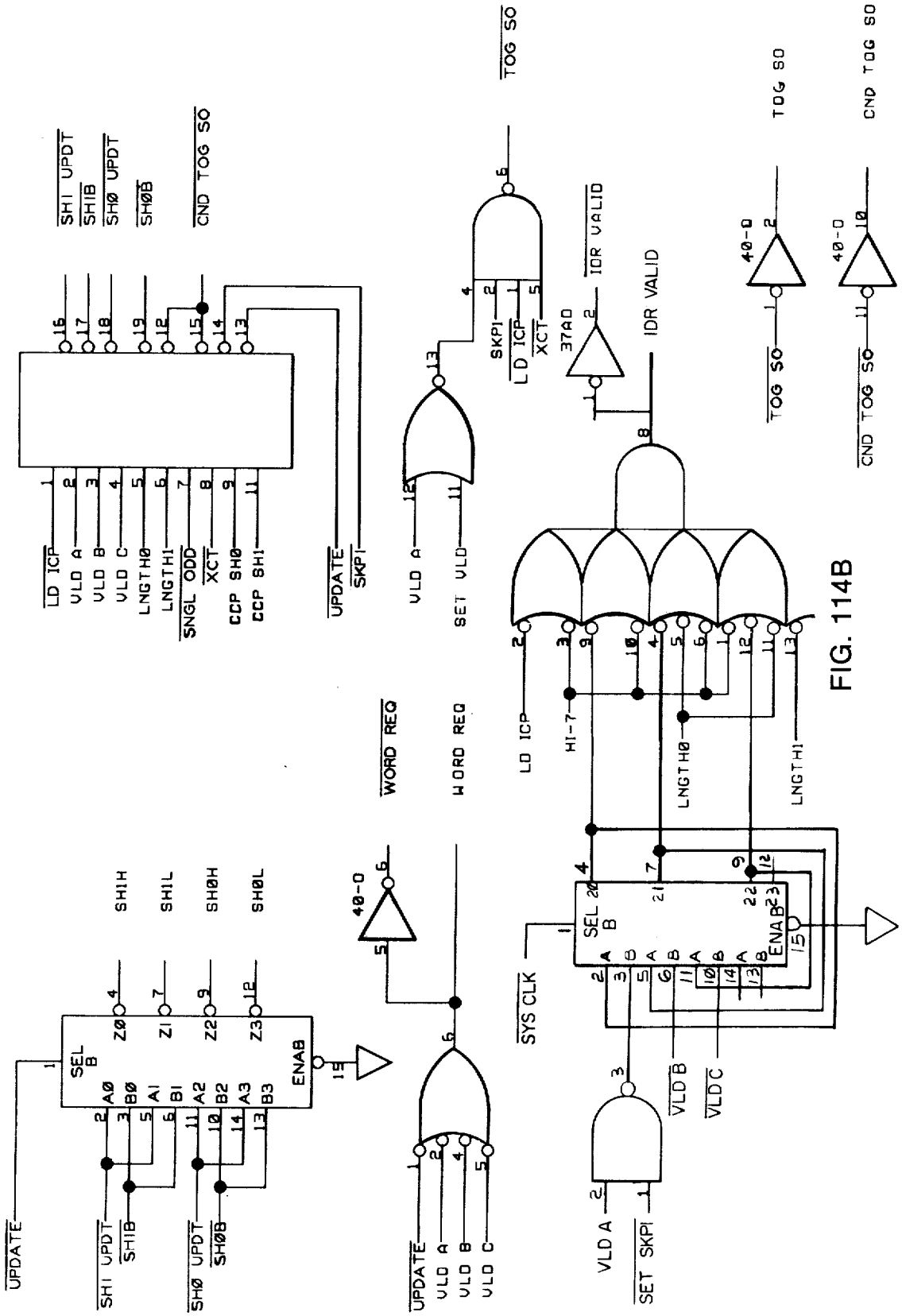
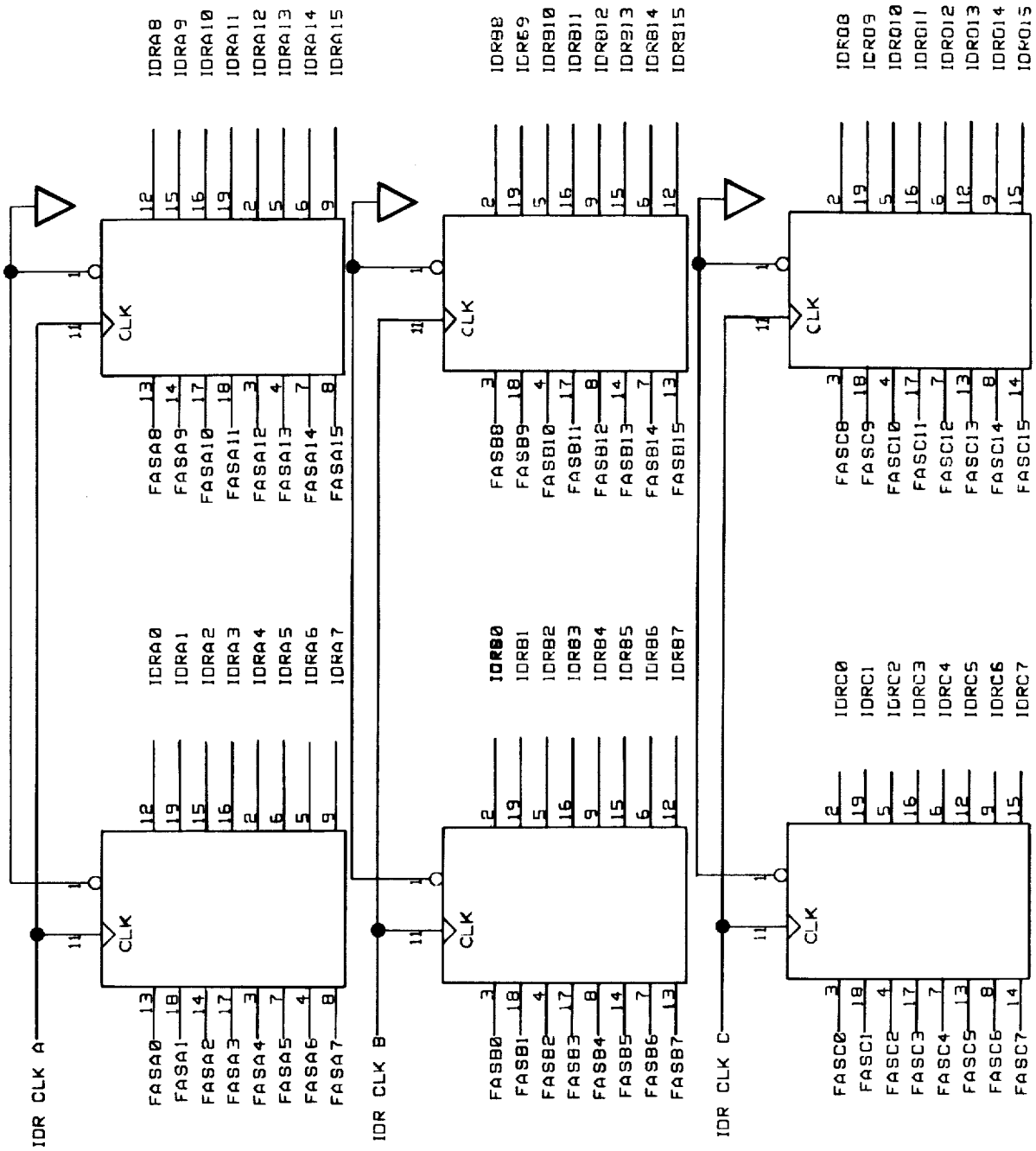
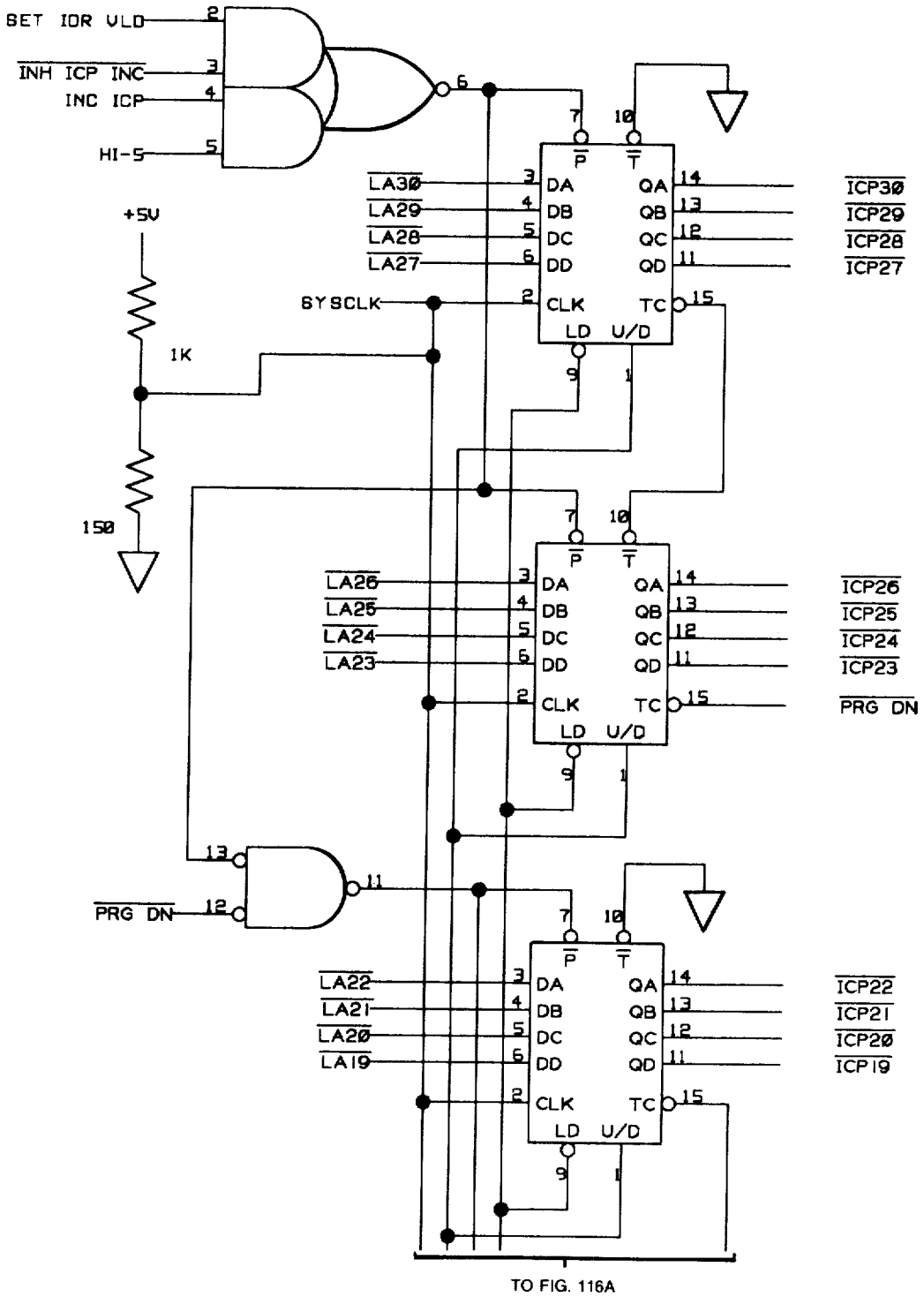

LD ICP
VLD A
VLD B
VLD C
LNGTH0
LNGTH1
SNGL ODD
XCT
CCP SH0
CCP SH1
SH1 UPDT
SH1B
SH0 UPDT
SH0B
CND TOG SO
UPDATE
SKPI
TOG SO
VLD A
SET VLD
SKPI
LD ICP
XCT
37A0
IDR VALID
40-0
WORD REQ
SH1H
SH1L
SH0H
SH0L
SEL B
Z0
Z1
Z2
Z3
ENAB
A0
B0
A1
B1
A2
B2
A3
B3
SYS CLK
VLDB
VLDC
SET SKPI
HI-7
LNGTH0
LNGTH1

FIG. 114B



INSTRUCTION DECODE REGISTER 142

FIG. 115



ICACHE POINTER LOGIC

FIG. 116

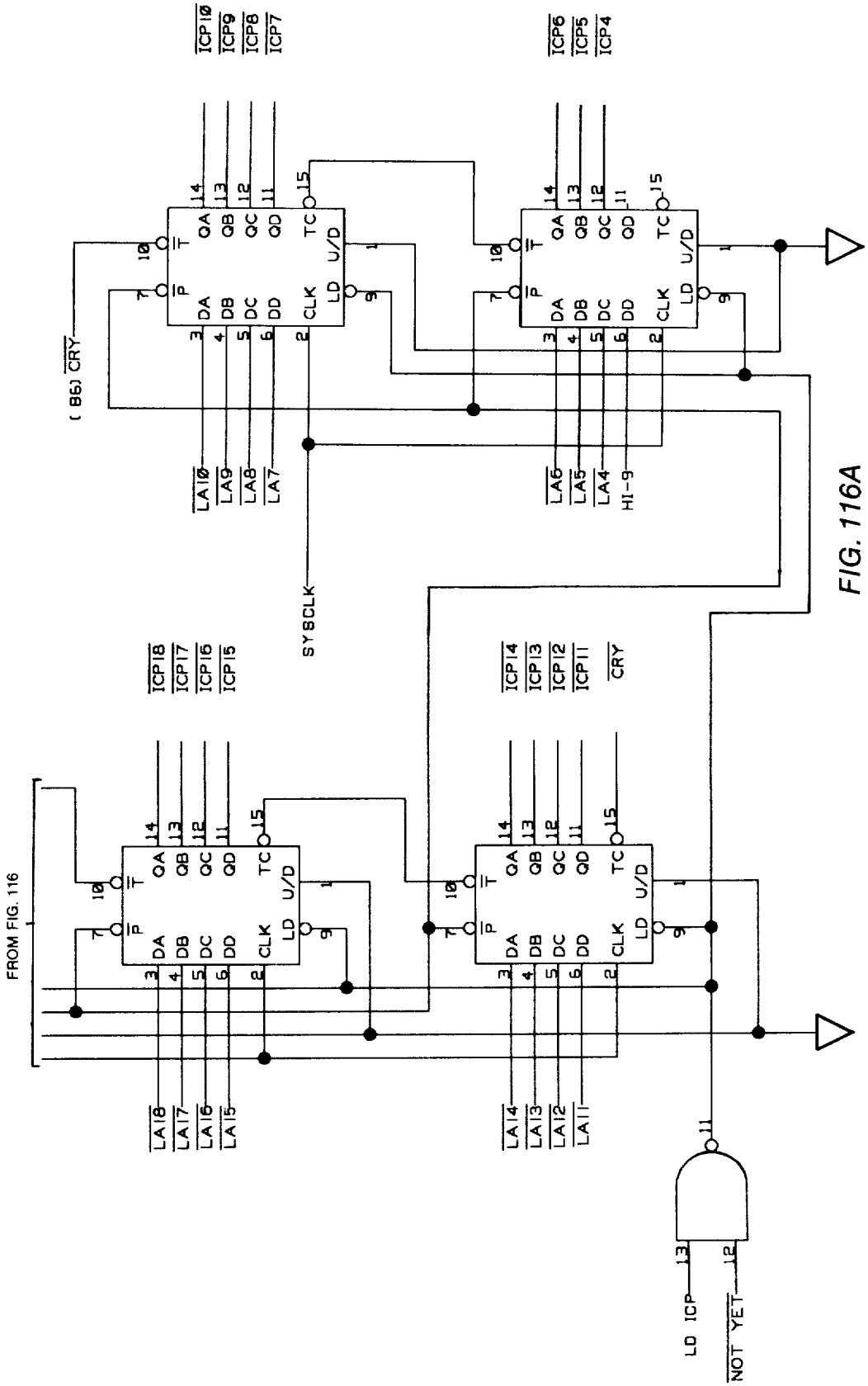
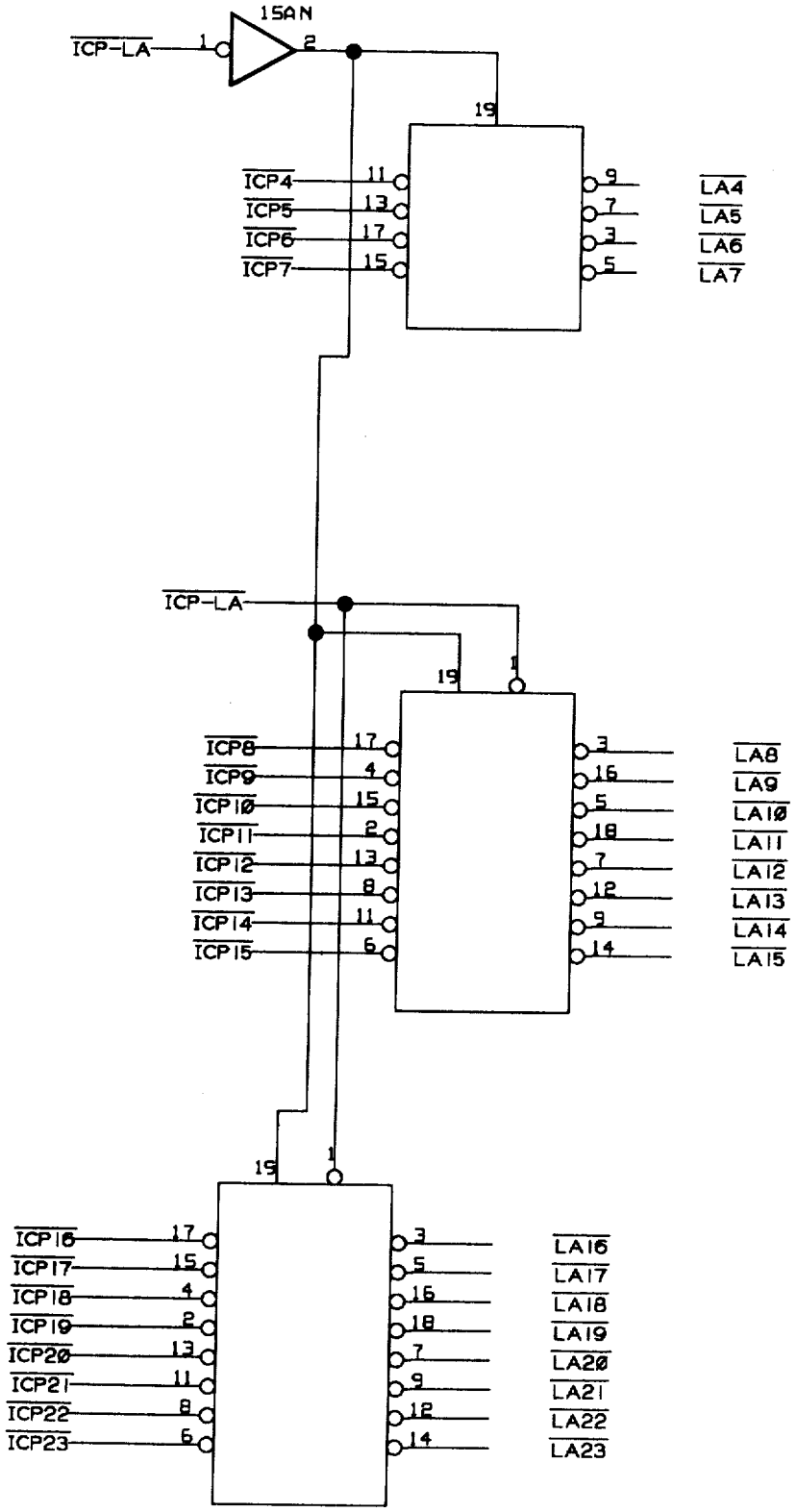
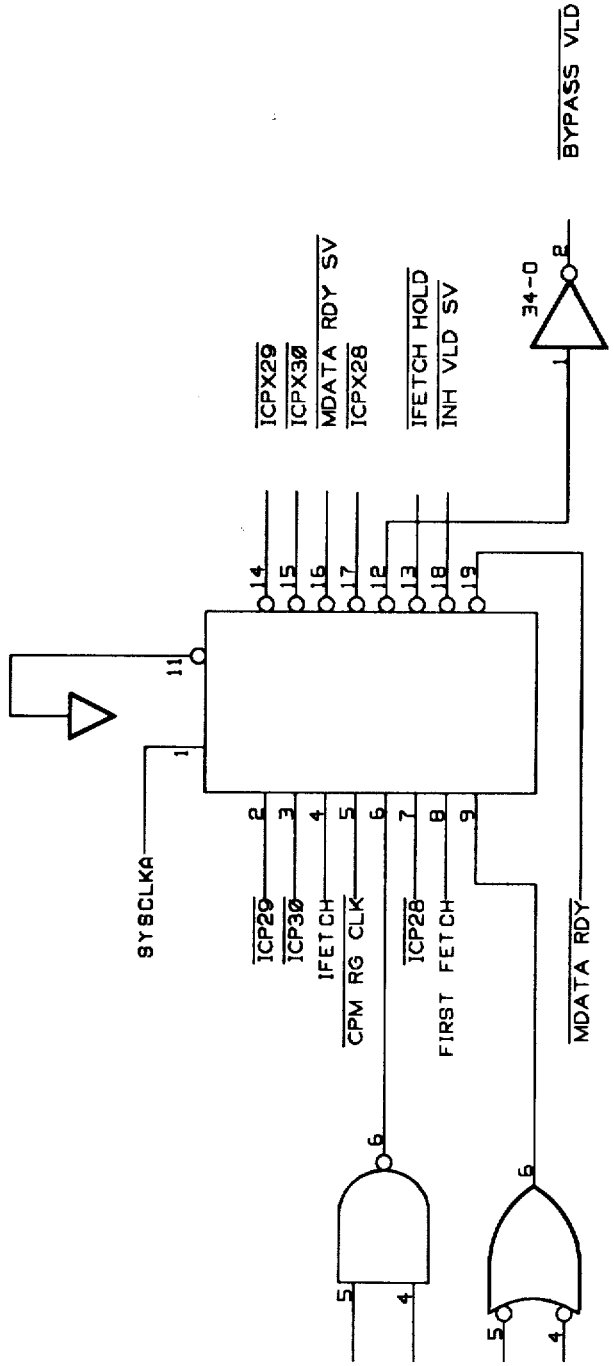

FROM FIG. 116
ICP18
ICP17
ICP16
ICP15
ICP14
ICP13
ICP12
ICP11
CRY
ICP10
ICP9
ICP8
ICP7
ICP6
ICP5
ICP4
(B6) CRY
SYSCLK
LA18
LA17
LA16
LA15
LA14
LA13
LA12
LA11
LA10
LA9
LA8
LA7
LA6
LA5
LA4
HI-9
LD ICP
NOT YET
QA
QB
QC
QD
TC
U/D
DA
DB
DC
DD
CLK
LD
T
P

FIG. 116A



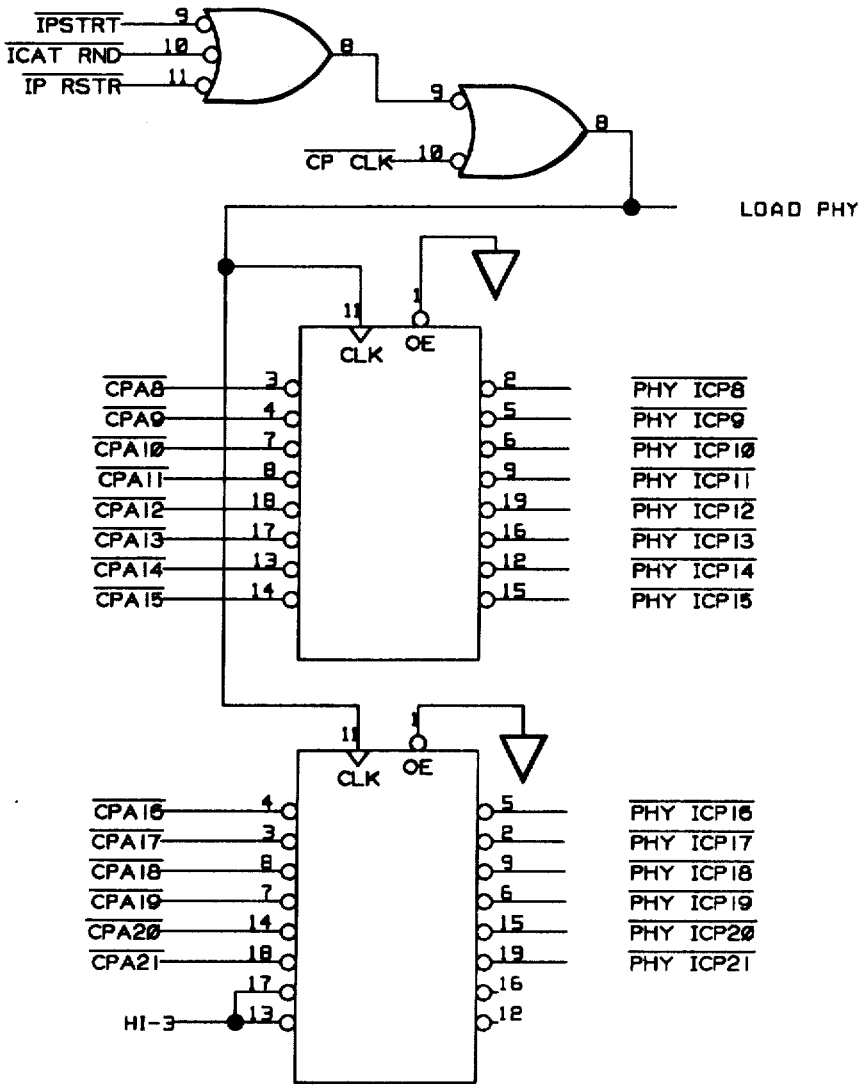
ICP LA DRIVER LOGIC

FIG. 117



I. P. PREFETCH REQUEST CONTROL LOGIC

FIG. 118



I. P. PHYS. TRANSLATION REGISTER 152

FIG. 119

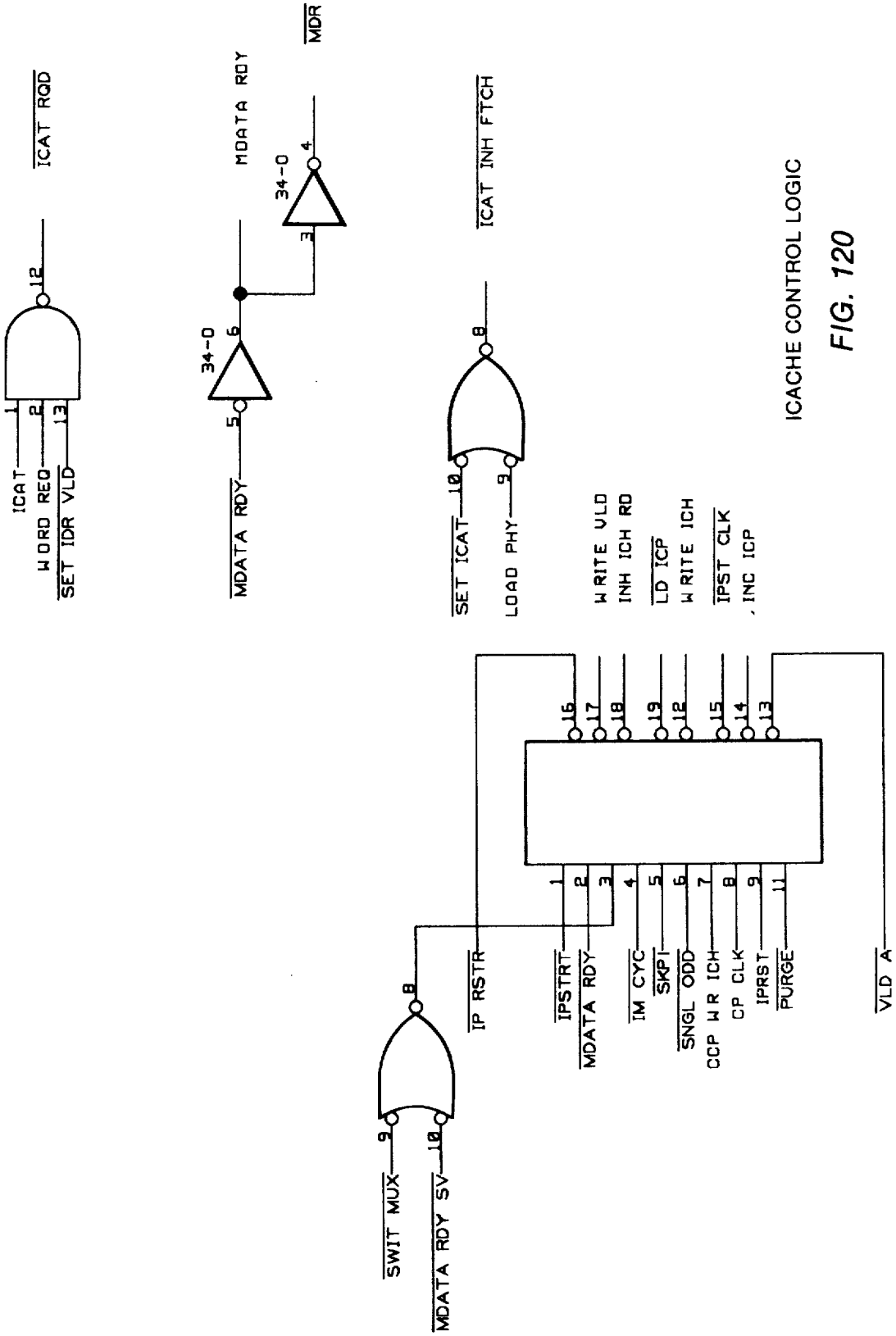

ICAT
WORD REQ
SET IDR VLD
ICAT RQD
MDATA RDY
34-0
MDR
MDATA RDY
SET ICAT
LOAD PHY
ICAT INH FTCH
WRITE VLD
INH ICH RD
LD ICP
WRITE ICH
IPST CLK
INC ICP
ICACHE CONTROL LOGIC
SWIT MUX
MDATA RDY SV
IP RSTR
IPSTRT
MDATA RDY
IM CYC
SKPI
SNGL ODD
CCP WR ICH
CP CLK
IPRST
PURGE
VLD A

FIG. 120

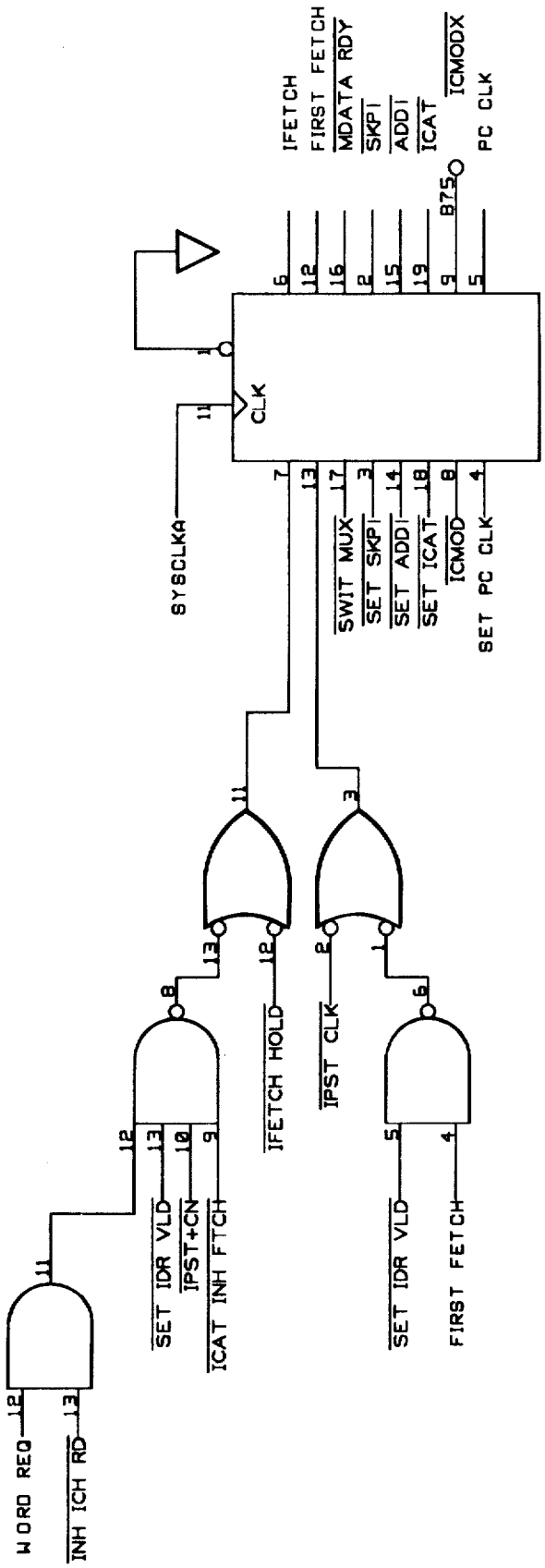
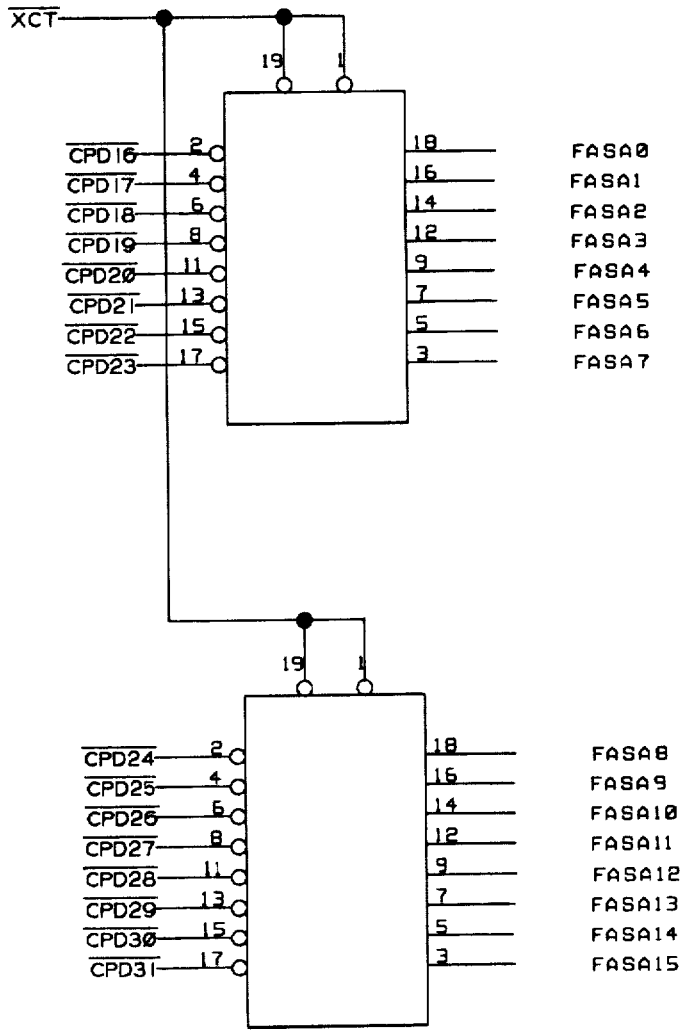
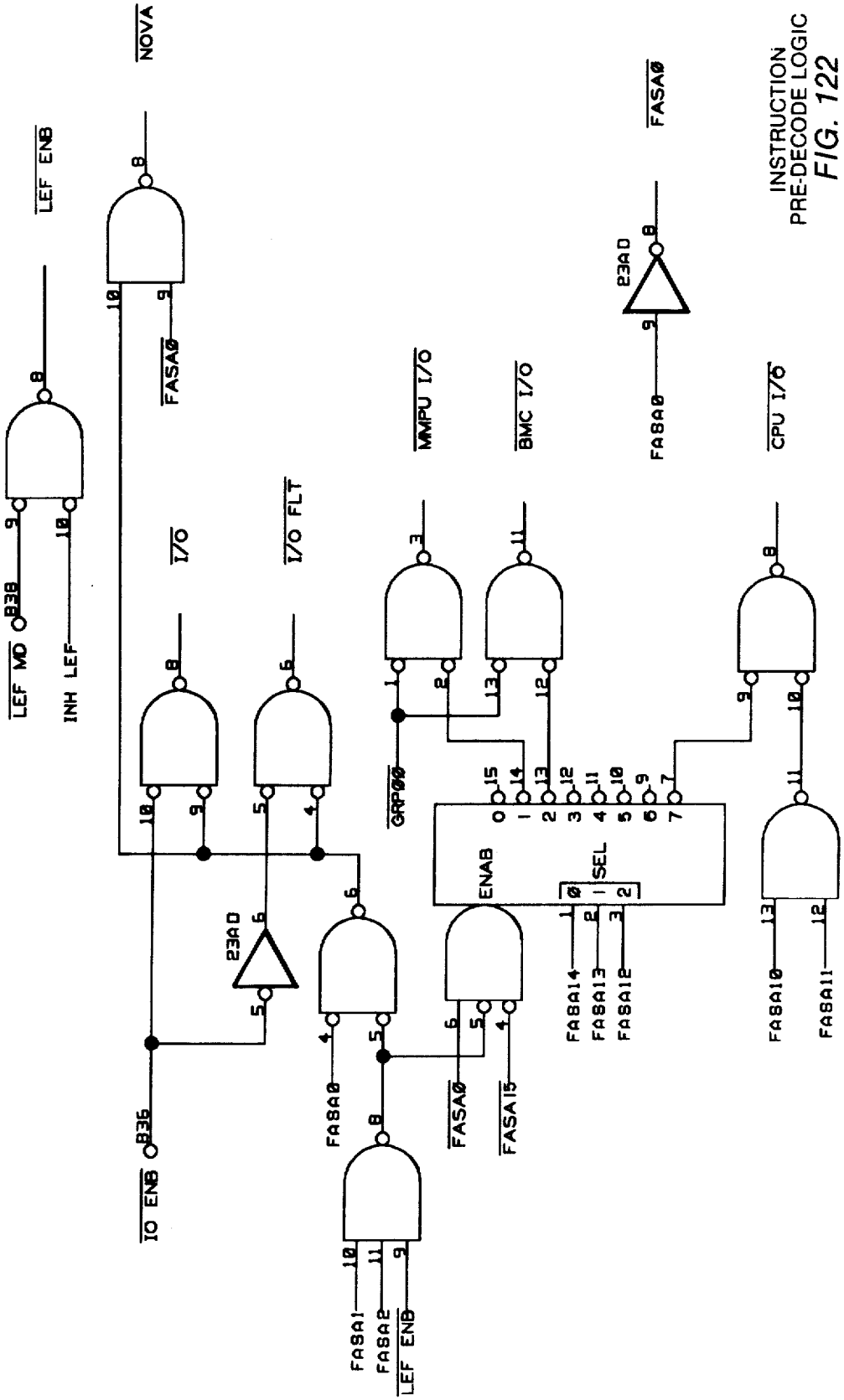

IFETCH
FIRST FETCH
MDATA RDY
SKPI
ADDI
ICAT
ICMODX
PC CLK
B75
CLK
SYSCLKA
SWIT MUX
SET SKPI
SET ADDI
SET ICAT
ICMOD
SET PC CLK
IFETCH HOLD
IPST CLK
SET IDR VLD
IPST+CN
ICAT INH FTCH
SET IDR VLD
FIRST FETCH
WORD REQ
INH ICH RD

FIG. 120A



DRIVER
(FROM CPD BUS)

FIG. 121



INSTRUCTION PRE-DECODE LOGIC
FIG. 122

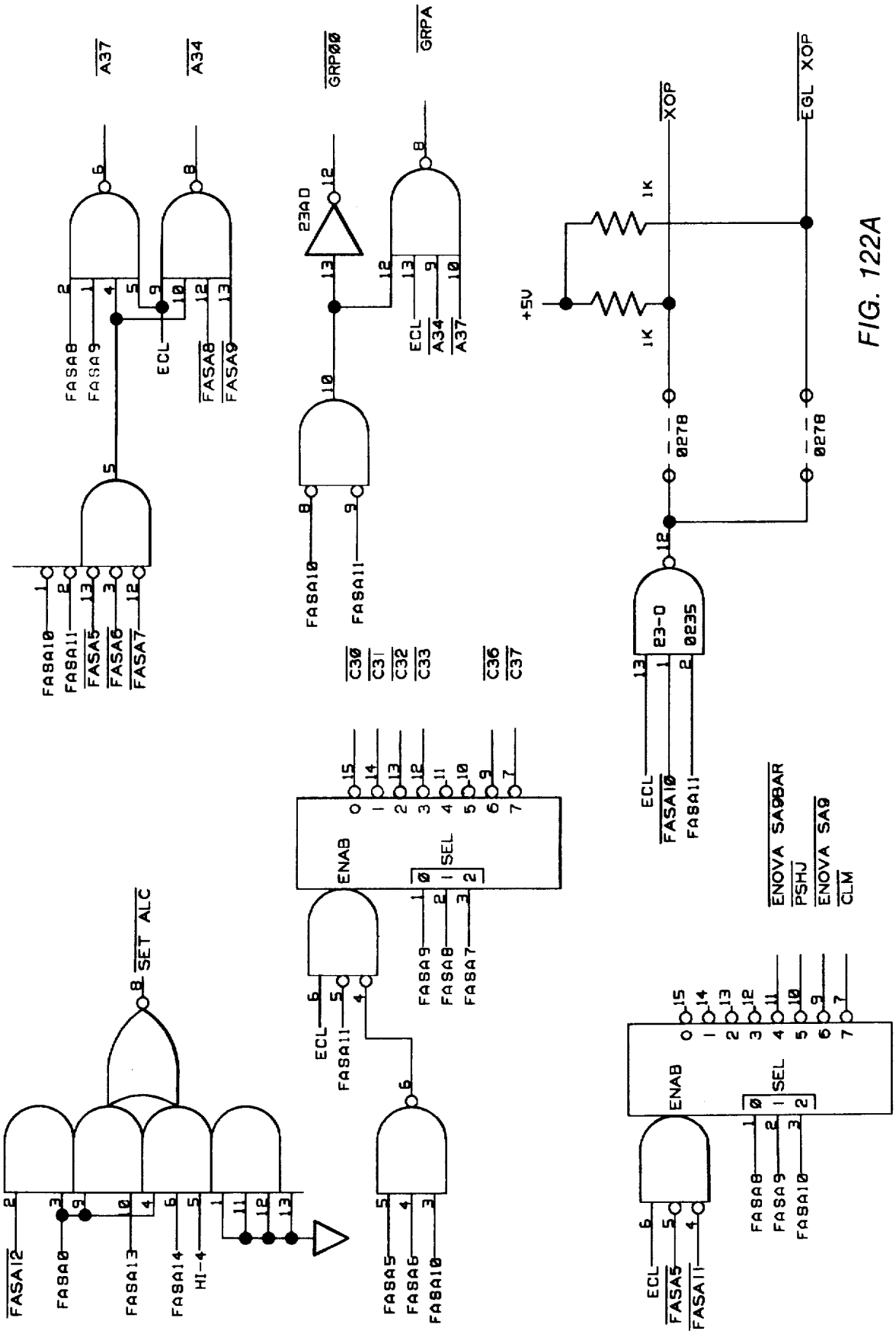

A37
A34
GRP00
GRPA
23AD
FASA8
FASA9
ECL
FASA8
FASA9
FASA10
FASA11
FASA5
FASA6
FASA7
C30
C31
C32
C33
C36
C37
+5V
1K
1K
XOP
EGL XOP
0278
0278
23-0
0235
ECL
FASA10
FASA11
ENOVA SA0BAR
PSHJ
ENOVA SA0
CLM
ENAB
SEL
SET ALC
FASA12
FASA0
FASA13
FASA14
HI-4
FASA5
FASA6
FASA10
ECL
FASA11
FASA9
FASA8
FASA7
ECL
FASA5
FASA11
FASA8
FASA9
FASA10

FIG. 122A

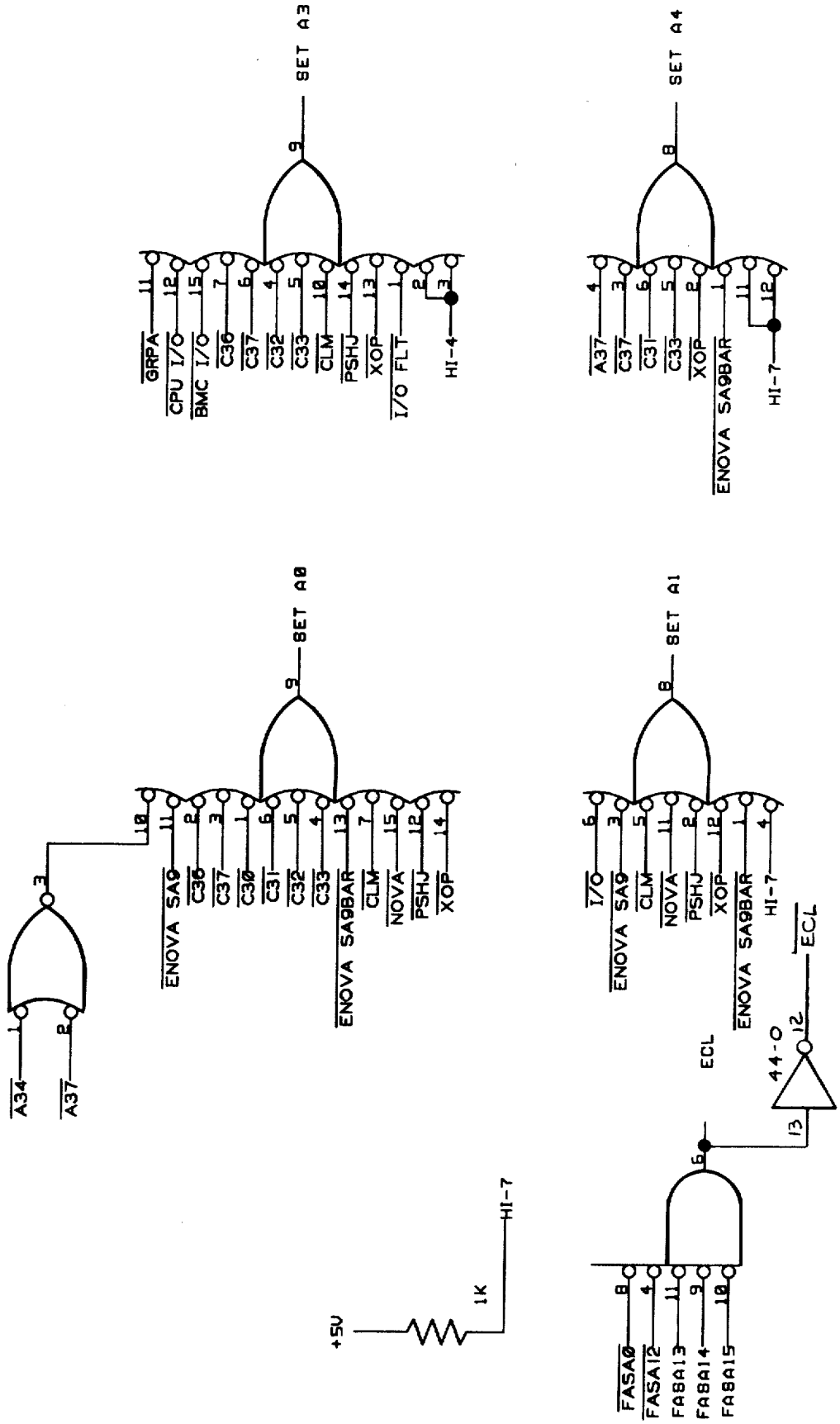

SET A3
GRPA
CPU I/O
BMC I/O
C36
C37
C32
C33
CLM
PSHJ
XOP
I/O FLT
HI-4
SET A4
A37
C37
C31
C33
XOP
ENOVA SA9BAR
HI-7
SET A0
A34
A37
ENOVA SA9
C36
C37
C30
C31
C32
C33
ENOVA SA9BAR
CLM
NOVA
PSHJ
XOP
SET A1
I/O
ENOVA SA9
CLM
NOVA
PSHJ
XOP
ENOVA SA9BAR
HI-7
ECL
44-0
ECL
+5V
1K
HI-7
FASA0
FASA12
FASA13
FASA14
FASA15

FIG. 122B

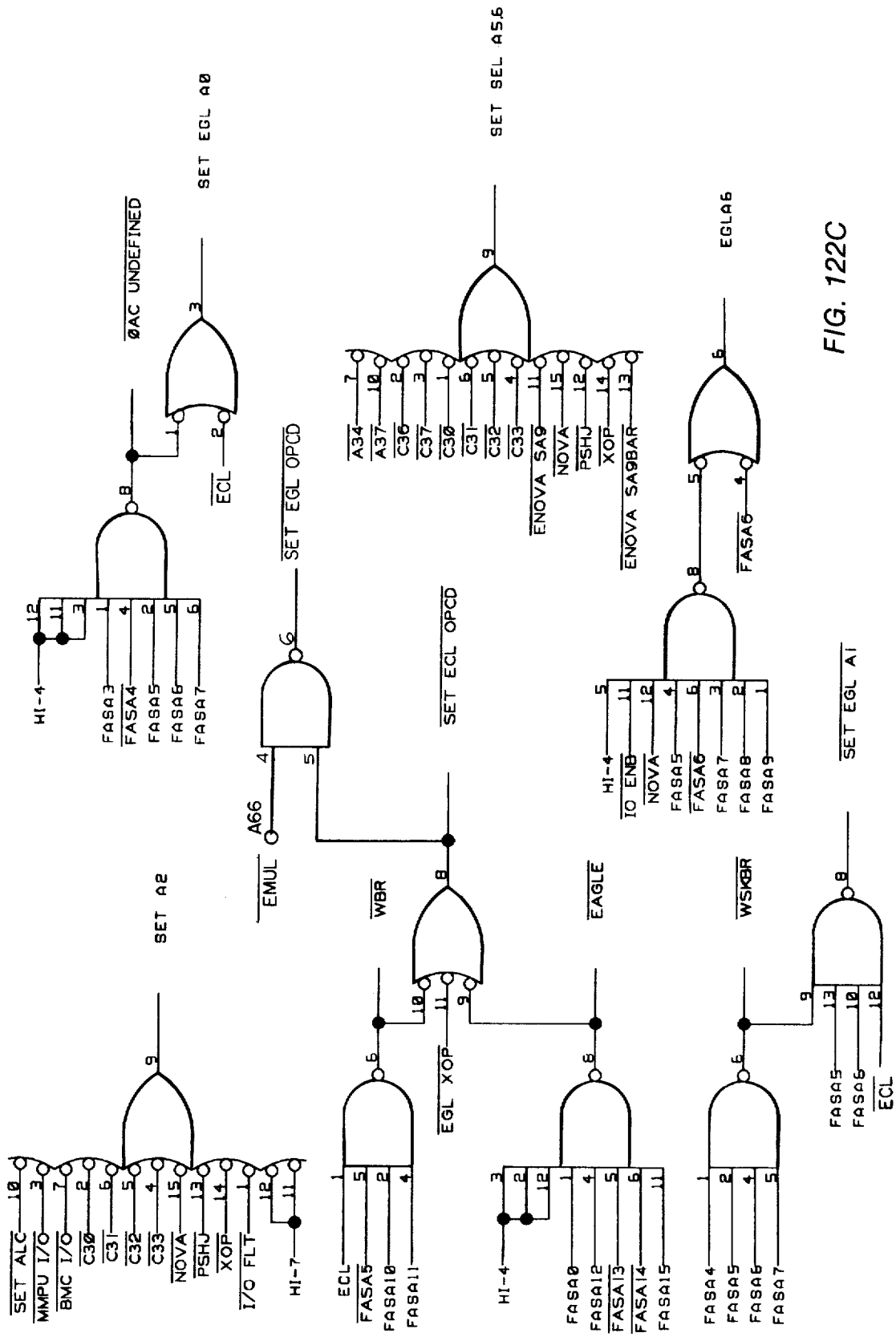

SET EGL A0
SET SEL A5,6
ØAC UNDEFINED
ECL
SET EGL OPCD
SET ECL OPCD
EGLA6
FASA6
HI-4
FASA3
FASA4
FASA5
FASA6
FASA7
A66
EMUL
WBR
EGL XOP
EAGLE
WSKBR
SET EGL A1
SET A2
SET ALC
MMPU I/O
BMC I/O
C30
C31
C32
C33
NOVA
PSHJ
XOP
I/O FLT
HI-7
ECL
FASA5
FASA10
FASA11
FASA0
FASA12
FASA13
FASA14
FASA15
FASA4
FASA5
FASA6
FASA7
A34
A37
C36
C37
C30
C31
C32
C33
ENOVA SA9
NOVA
PSHJ
XOP
ENOVA SA9BAR
HI-4
IO END
NOVA
FASA5
FASA6
FASA7
FASA8
FASA9
FASA5
FASA6
ECL

FIG. 122C

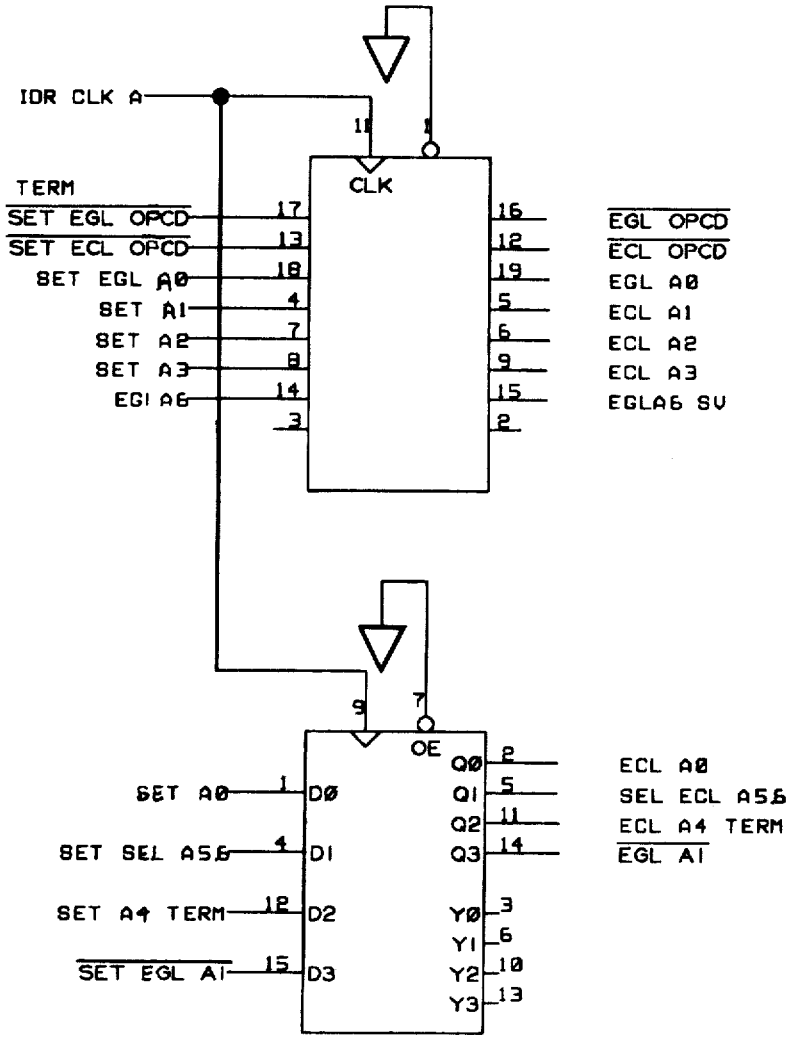
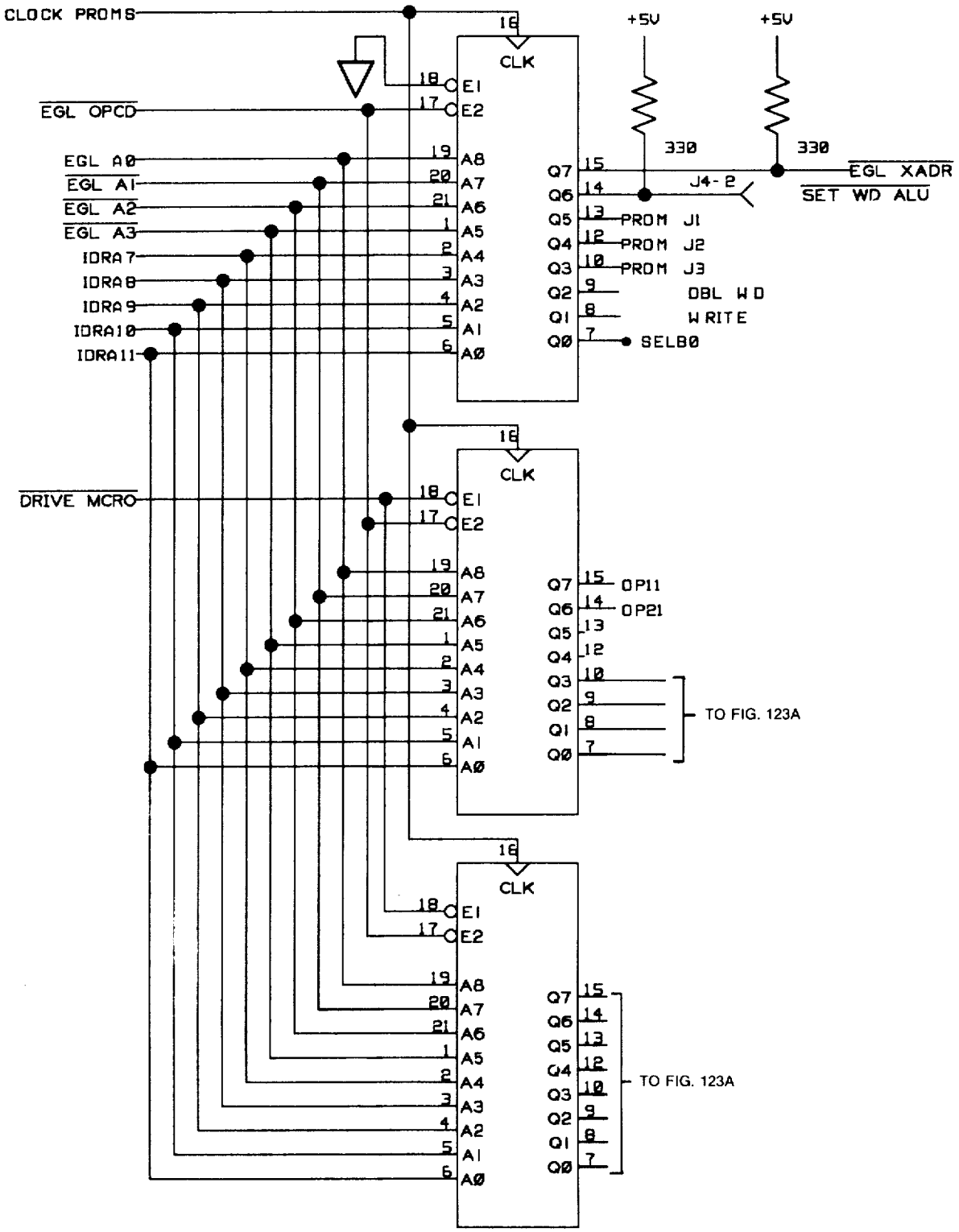

IDR CLK A
TERM
SET EGL OPCD
SET ECL OPCD
SET EGL A0
SET A1
SET A2
SET A3
EGI A6
CLK
EGL OPCD
ECL OPCD
EGL A0
ECL A1
ECL A2
ECL A3
EGLA6 SV
OE
SET A0
SET SEL A5&
SET A4 TERM
SET EGL A1
D0
D1
D2
D3
Q0
Q1
Q2
Q3
Y0
Y1
Y2
Y3
ECL A0
SEL ECL A5&
ECL A4 TERM
EGL A1

FIG. 122D



DECODE PROMS 140 AND 141

FIG. 123

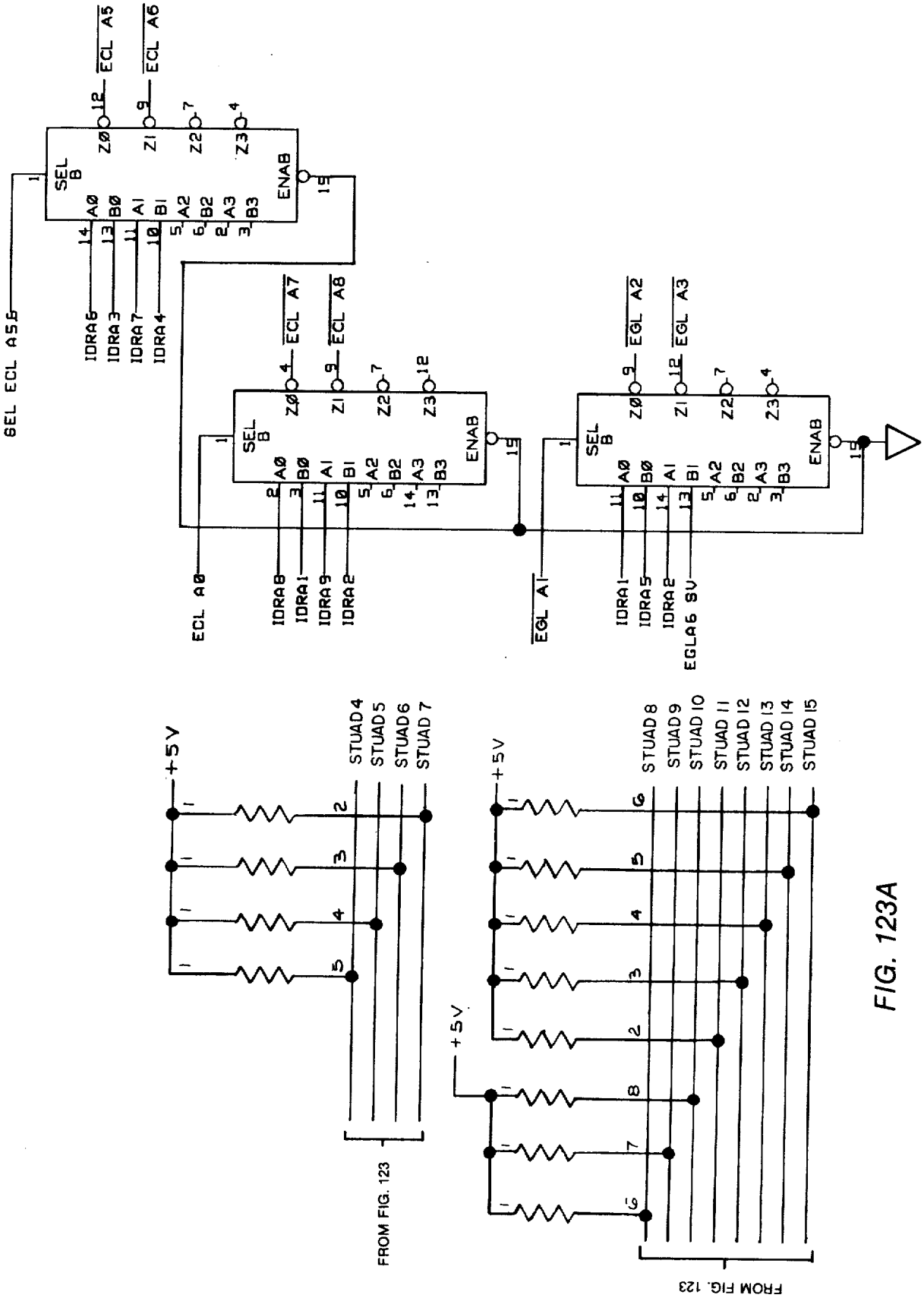

SEL ECL A5
IDRA6
IDRA3
IDRA7
IDRA4
SEL B
A0 B0 A1 B1 A2 B2 A3 B3
Z0 Z1 Z2 Z3
ENAB
ECL A5
ECL A6
ECL A7
ECL A8
ECL A0
IDRA8
IDRA1
IDRA9
IDRA2
EGL A2
EGL A3
EGL A1
IDRA1
IDRA5
IDRA2
EGLA6 SV
+5V
STUAD4
STUAD5
STUAD6
STUAD7
FROM FIG. 123
STUAD8
STUAD9
STUAD10
STUAD11
STUAD12
STUAD13
STUAD14
STUAD15
FROM FIG. 123

FIG. 123A

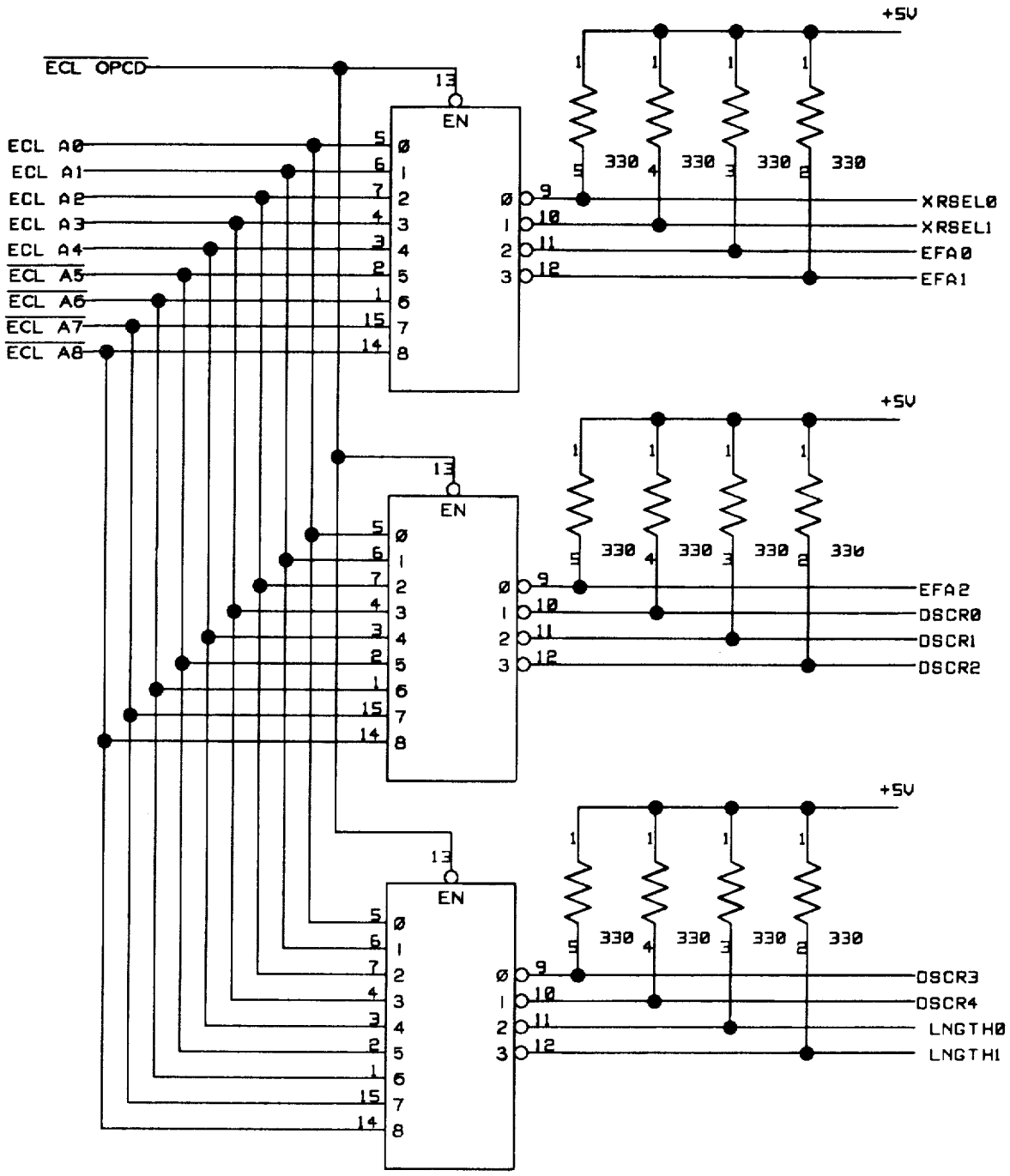

ECL OPCD
ECL A0
ECL A1
ECL A2
ECL A3
ECL A4
ECL A5
ECL A6
ECL A7
ECL A8
EN
+5V
330
XRSEL0
XRSEL1
EFA0
EFA1
EFA2
DSCR0
DSCR1
DSCR2
DSCR3
DSCR4
LNGTH0
LNGTH1

FIG. 123B

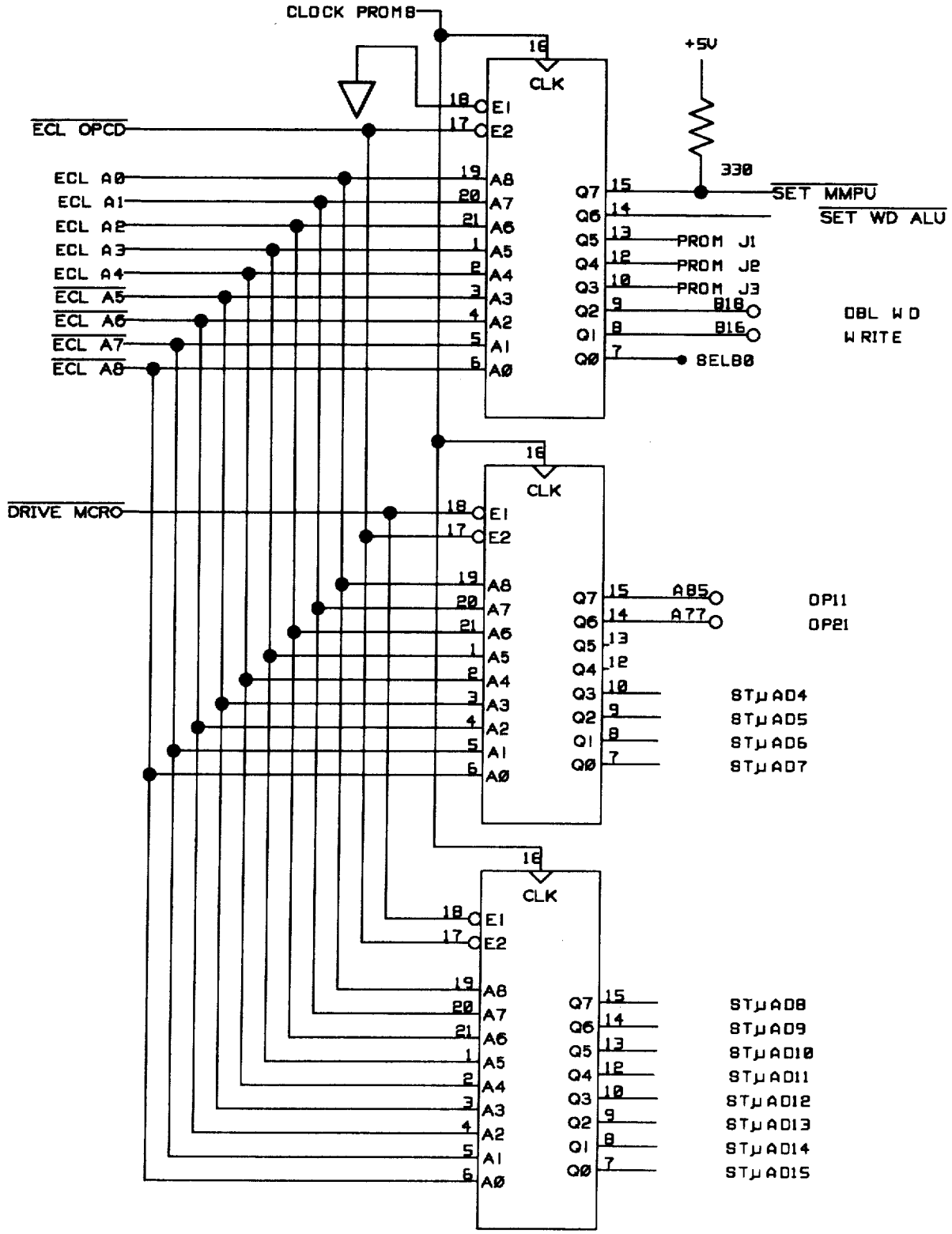

CLOCK PROM8
+5V
330
CLK
E1
E2
A8
A7
A6
A5
A4
A3
A2
A1
A0
Q7
Q6
Q5
Q4
Q3
Q2
Q1
Q0
ECL OPCD
ECL A0
ECL A1
ECL A2
ECL A3
ECL A4
ECL A5
ECL A6
ECL A7
ECL A8
SET MMPU
SET WD ALU
PROM J1
PROM J2
PROM J3
B18
B16
SELB0
DBL WD
WRITE
DRIVE MCRO
A85
A77
OP11
OP21
STμAD4
STμAD5
STμAD6
STμAD7
STμAD8
STμAD9
STμAD10
STμAD11
STμAD12
STμAD13
STμAD14
STμAD15

FIG. 123C

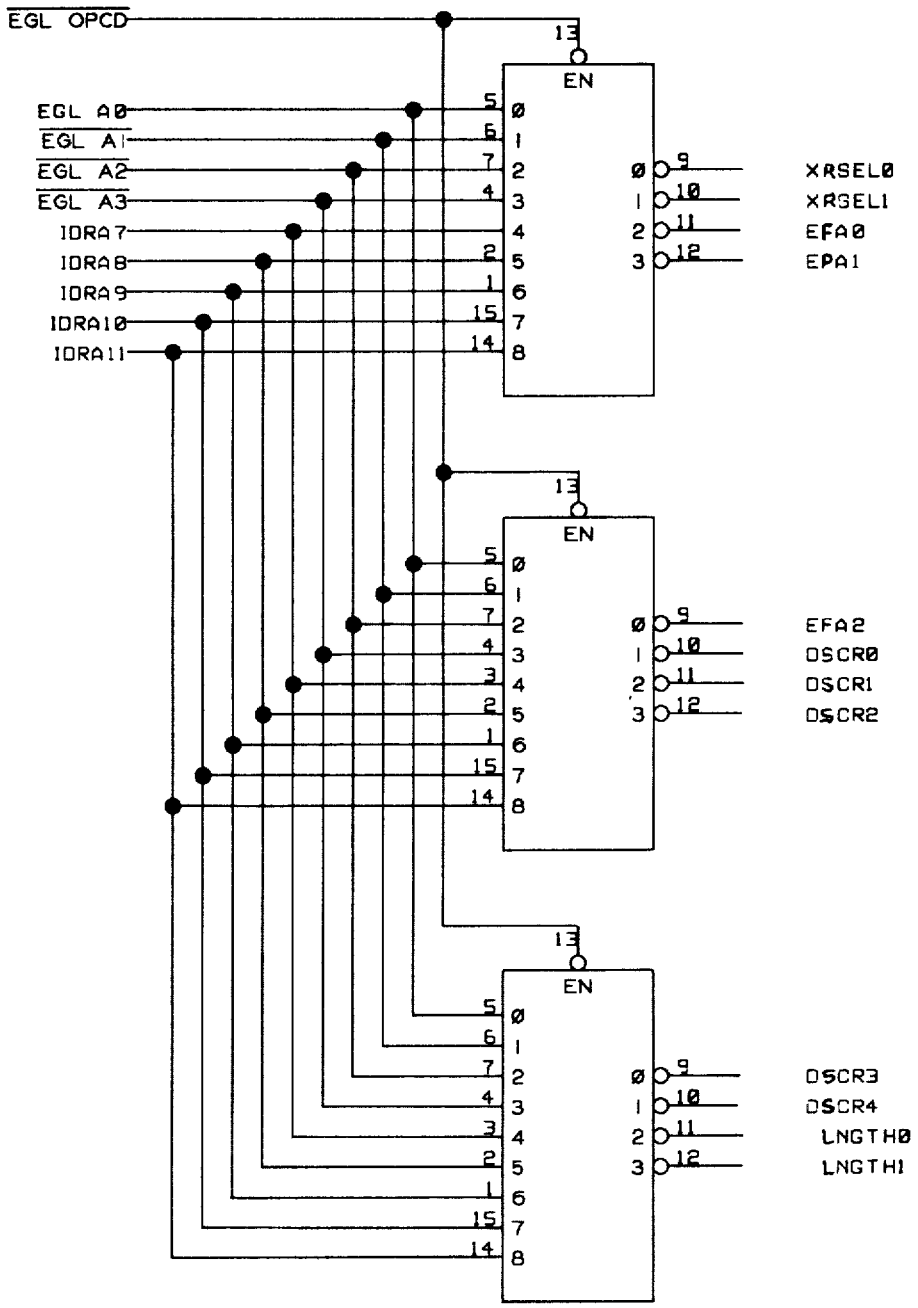

EGL OPCD
EGL A0
EGL A1
EGL A2
EGL A3
IDRA7
IDRA8
IDRA9
IDRA10
IDRA11
EN
XRSEL0
XRSEL1
EFA0
EPA1
EFA2
DSCR0
DSCR1
DSCR2
DSCR3
DSCR4
LNGTH0
LNGTH1

FIG. 123D

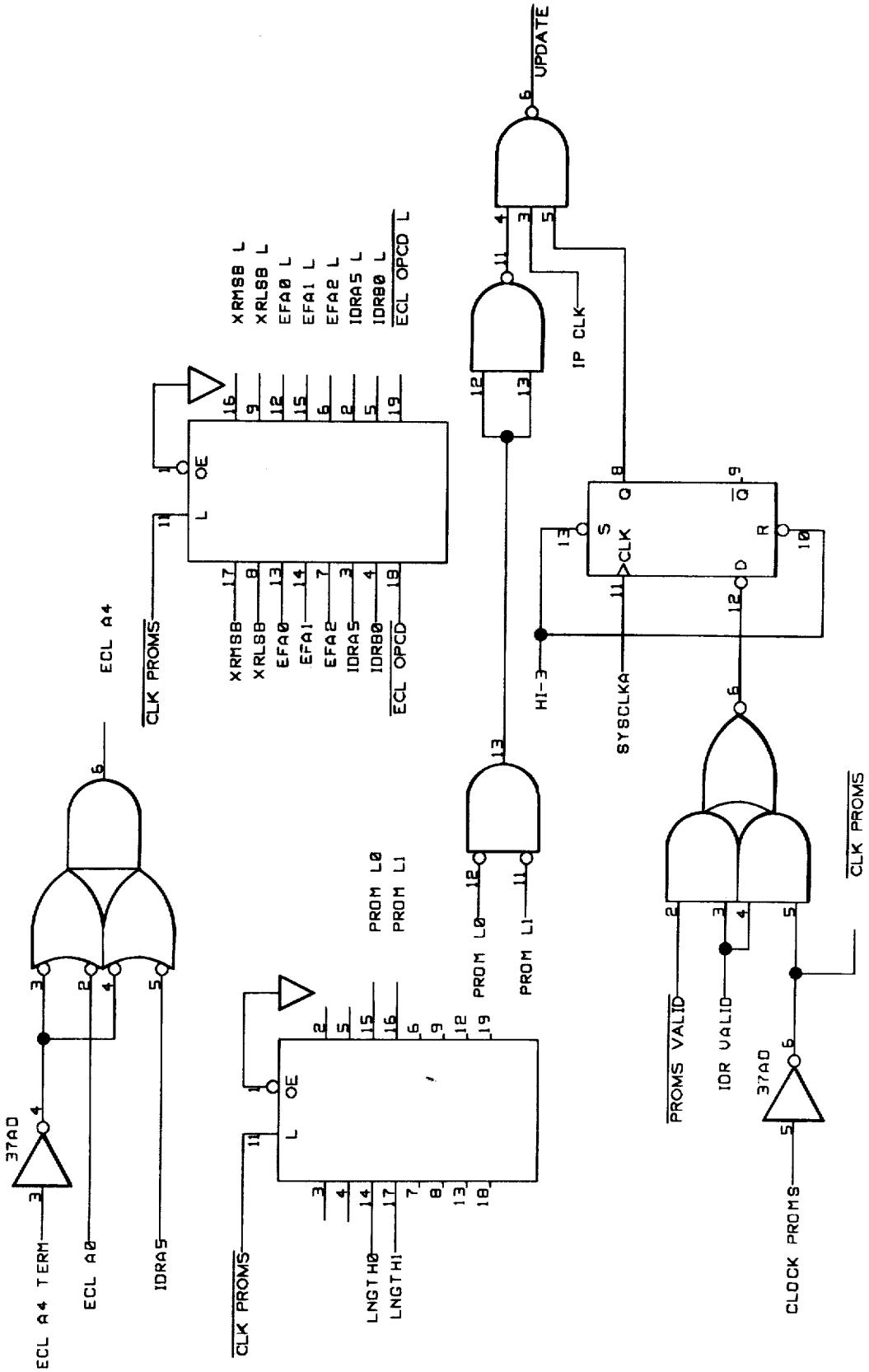
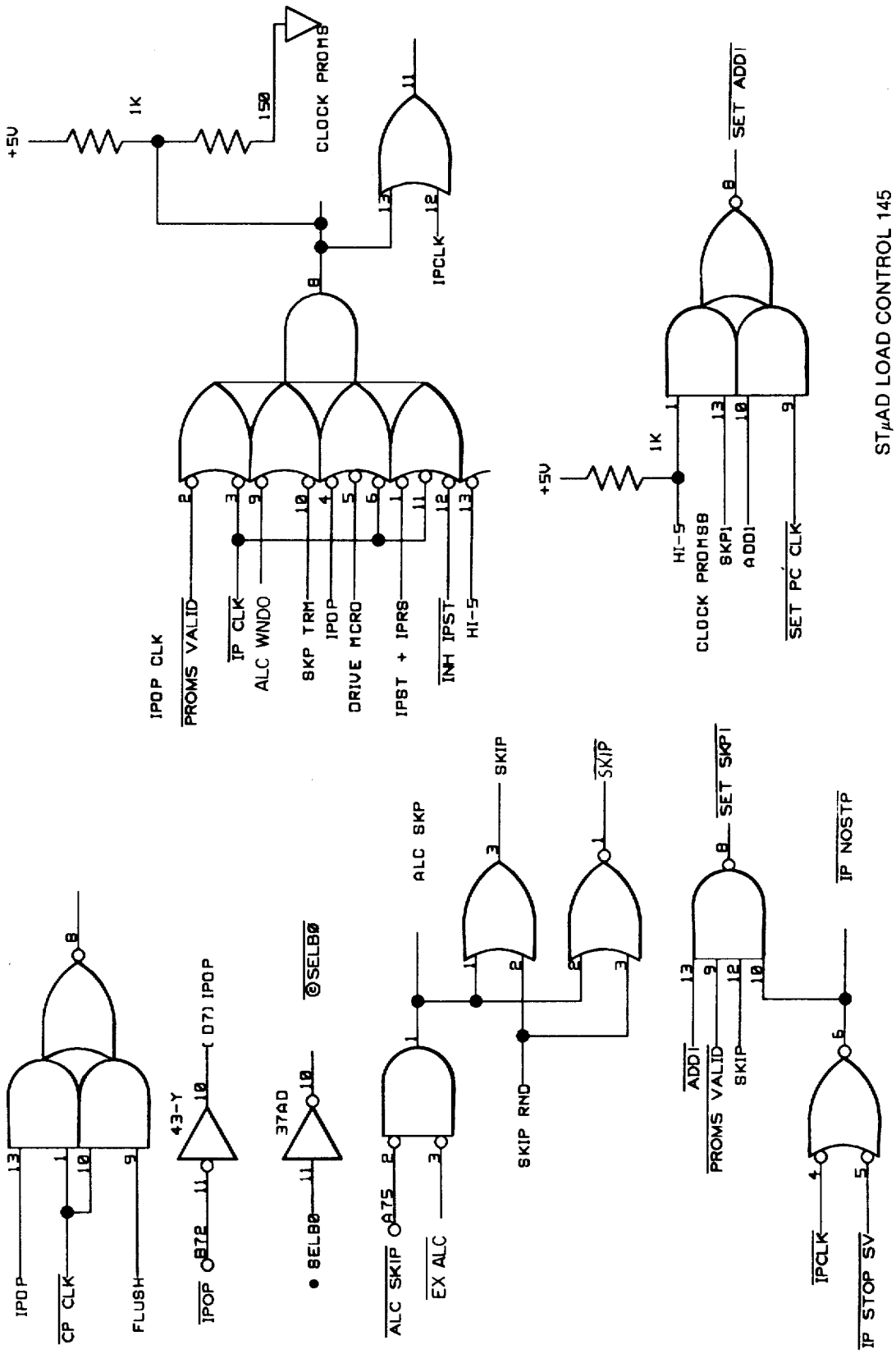

UPDATE
IP CLK
XRMSB L
XRLSB L
EFA0 L
EFA1 L
EFA2 L
IDRA5 L
IDRB0 L
ECL OPCD L
OE
L
CLK PROMS
XRMSB
XRLSB
EFA0
EFA1
EFA2
IDRA5
IDRB0
ECL OPCD
ECL A4
S
CLK
Q
R
D
HI-3
SYSCLKA
PROM L0
PROM L1
LNGTH0
LNGTH1
37AD
ECL A4 TERM
ECL A0
IDRA5
PROMS VALID
IDR VALID
CLOCK PROMS

FIG. 123E



STμAD LOAD CONTROL 145

FIG. 124

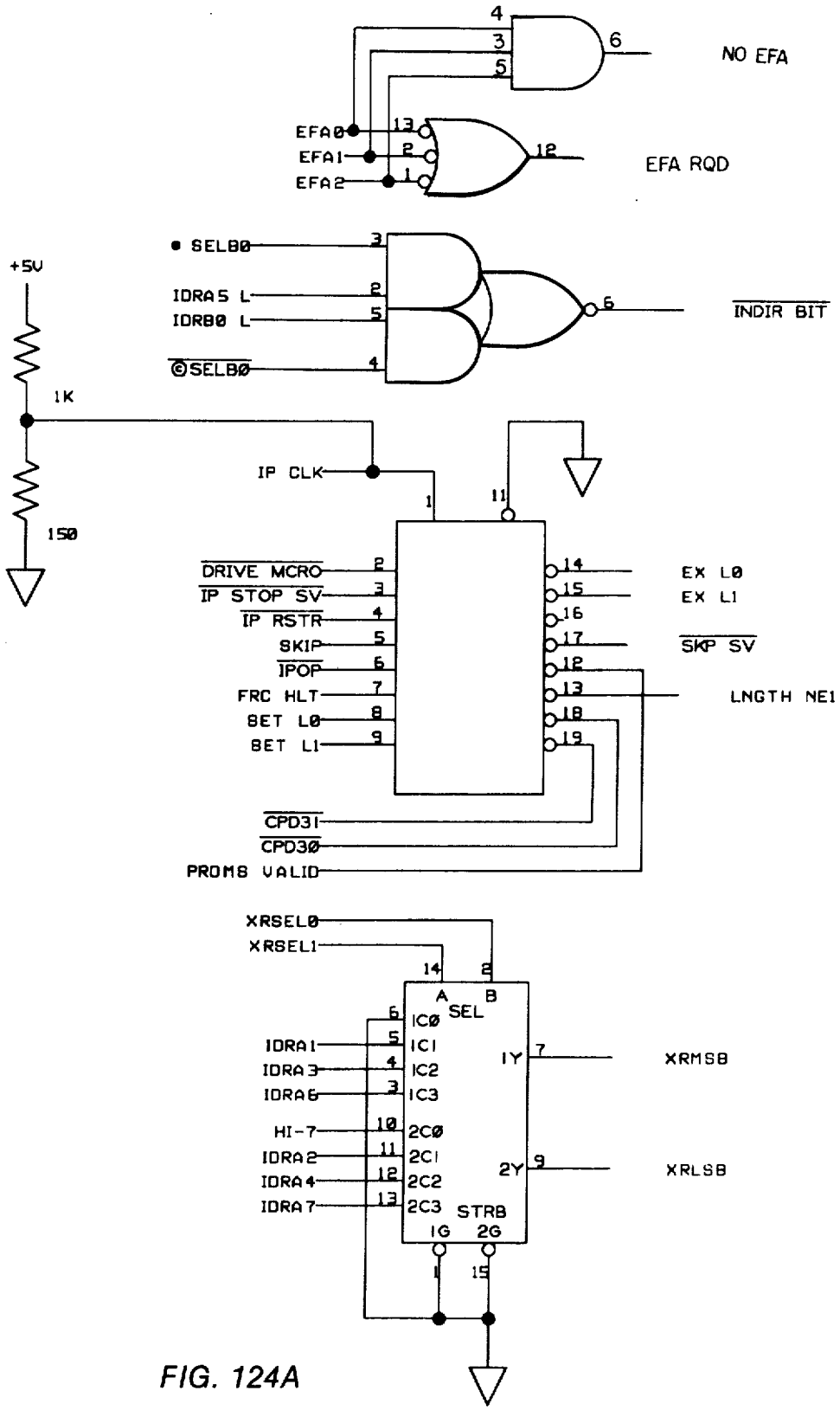

NO EFA
EFA0
EFA1
EFA2
EFA RQD
SELB0
IDRA5 L
IDRB0 L
©SELB0
INDIR BIT
+5V
1K
150
IP CLK
DRIVE MCRO
IP STOP SV
IP RSTR
SKIP
IPOP
FRC HLT
SET L0
SET L1
EX L0
EX L1
SKP SV
LNGTH NE1
CPD31
CPD30
PROMS VALID
XRSEL0
XRSEL1
SEL
IDRA1
IDRA3
IDRA6
HI-7
IDRA2
IDRA4
IDRA7
XRMSB
XRLSB
STRB

FIG. 124A

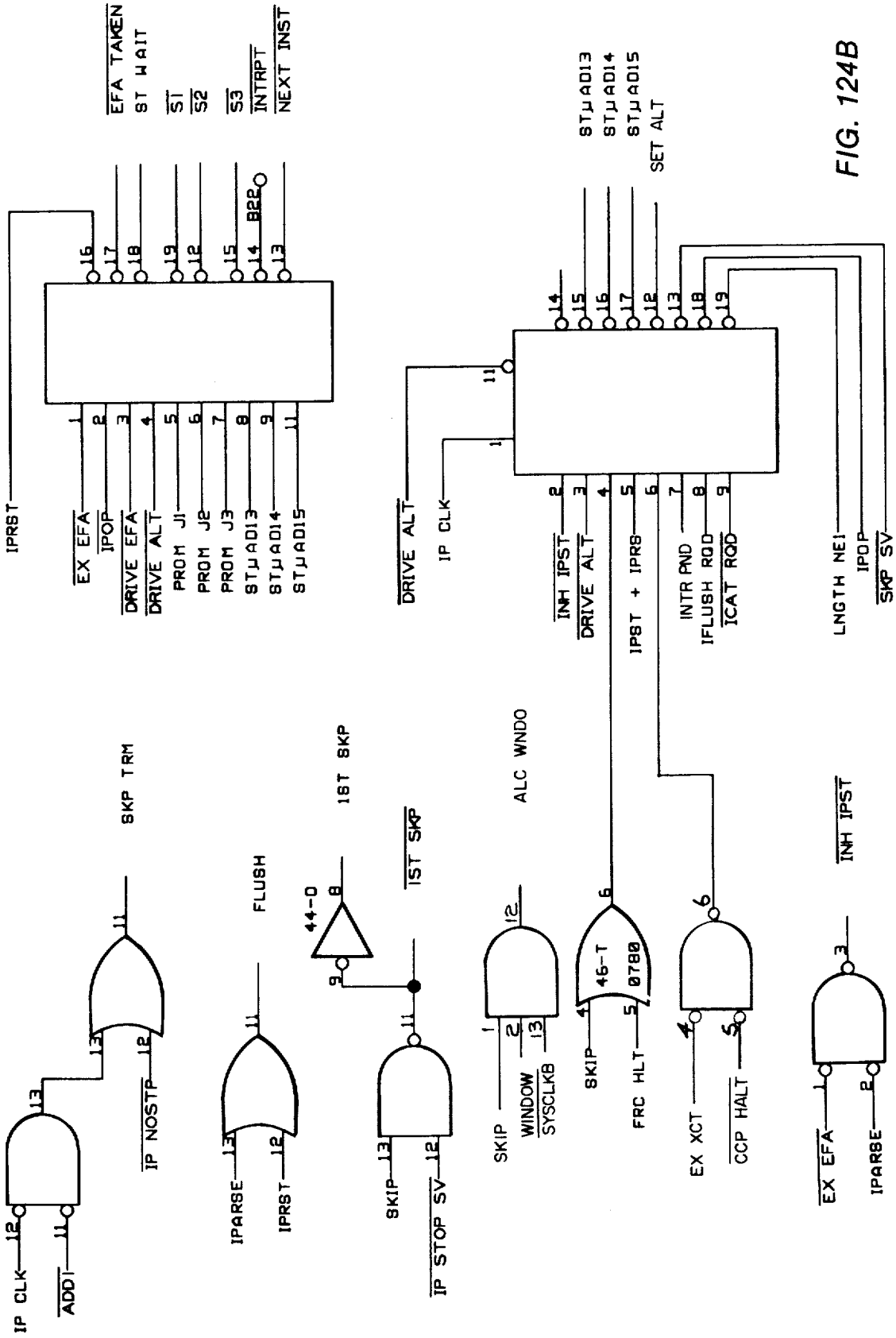

IPRST
EX EFA
IPOP
DRIVE EFA
DRIVE ALT
PROM J1
PROM J2
PROM J3
STμAD13
STμAD14
STμAD15
EFA TAKEN
ST WAIT
S1
S2
S3
INTRPT
NEXT INST
B22
DRIVE ALT
IP CLK
INH IPST
DRIVE ALT
IPST + IPRS
INTR PND
IFLUSH RQD
ICAT RQD
LNGTH NEI
IPOP
SKP SV
STμAD13
STμAD14
STμAD15
SET ALT
IP CLK
ADD1
SKP TRM
IP NOSTP
IPARSE
IPRST
FLUSH
44-0
1ST SKP
1ST SKP
SKIP
IP STOP SV
ALC WNDO
SKIP
WINDOW
SYSCLKB
46-T
0780
SKIP
FRC HLT
EX XCT
CCP HALT
INH IPST
EX EFA
IPARSE

FIG. 124B

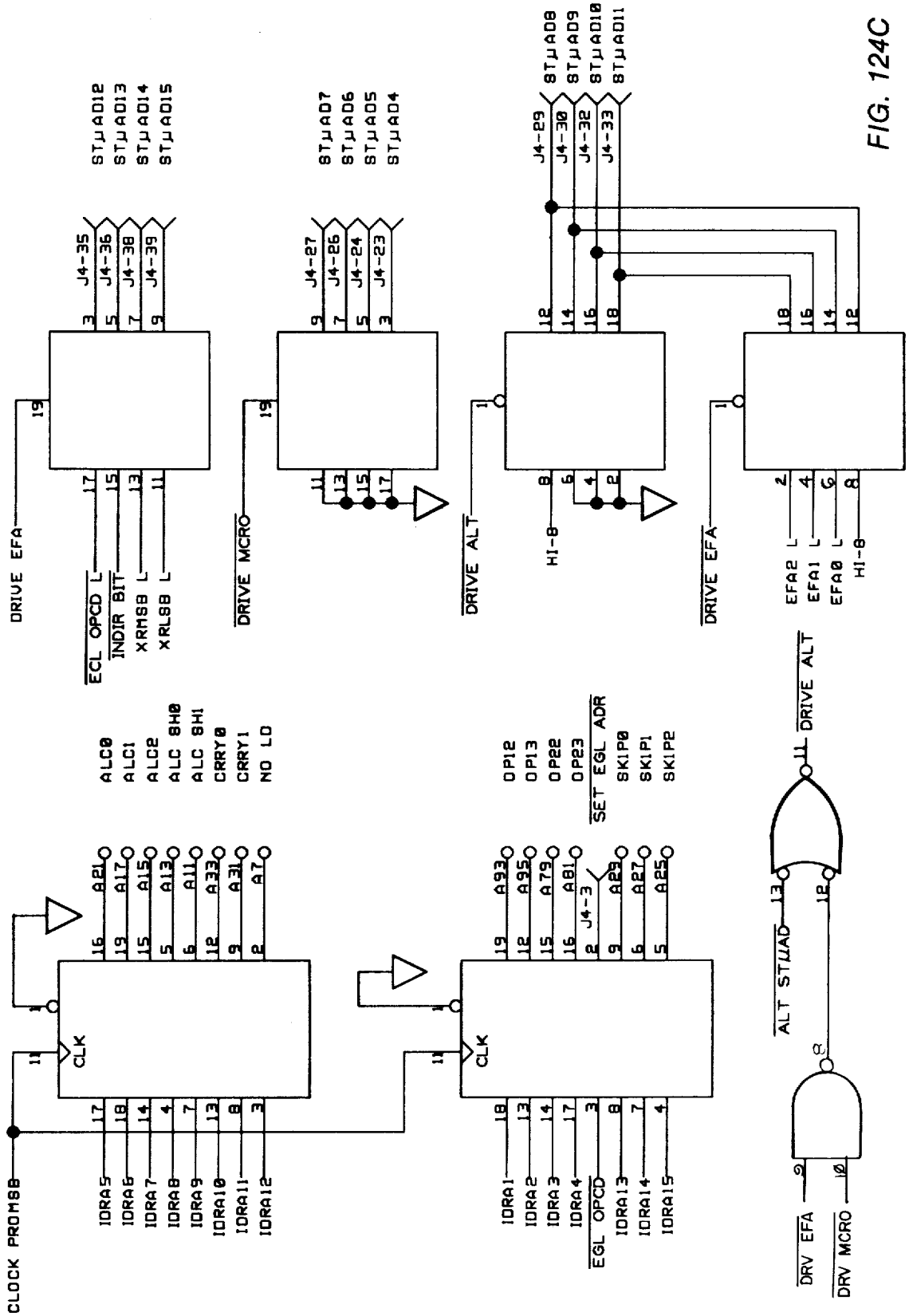

DRIVE EFA
ECL OPCD L
INDIR BIT
XRMSB L
XRLSB L
J4-35
J4-36
J4-38
J4-39
ST μAD12
ST μAD13
ST μAD14
ST μAD15
DRIVE MCRO
J4-27
J4-26
J4-24
J4-23
ST μAD7
ST μAD6
ST μAD5
ST μAD4
DRIVE ALT
HI-8
J4-29
J4-30
J4-32
J4-33
ST μAD8
ST μAD9
ST μAD10
ST μAD11
DRIVE EFA
EFA2 L
EFA1 L
EFA0 L
HI-8
CLOCK PROMSB
CLK
IDRA5
IDRA6
IDRA7
IDRA8
IDRA9
IDRA10
IDRA11
IDRA12
A21
A17
A15
A13
A11
A33
A31
A7
ALC0
ALC1
ALC2
ALC SH0
ALC SH1
CRRY0
CRRY1
NO LO
CLK
IDRA1
IDRA2
IDRA3
IDRA4
EGL OPCD
IDRA13
IDRA14
IDRA15
A93
A95
A79
A81
J4-3
A29
A27
A25
OP12
OP13
OP22
OP23
SET EGL ADR
SKIP0
SKIP1
SKIP2
ALT STμAD
DRIVE ALT
DRV EFA
DRV MCRO

FIG. 124C

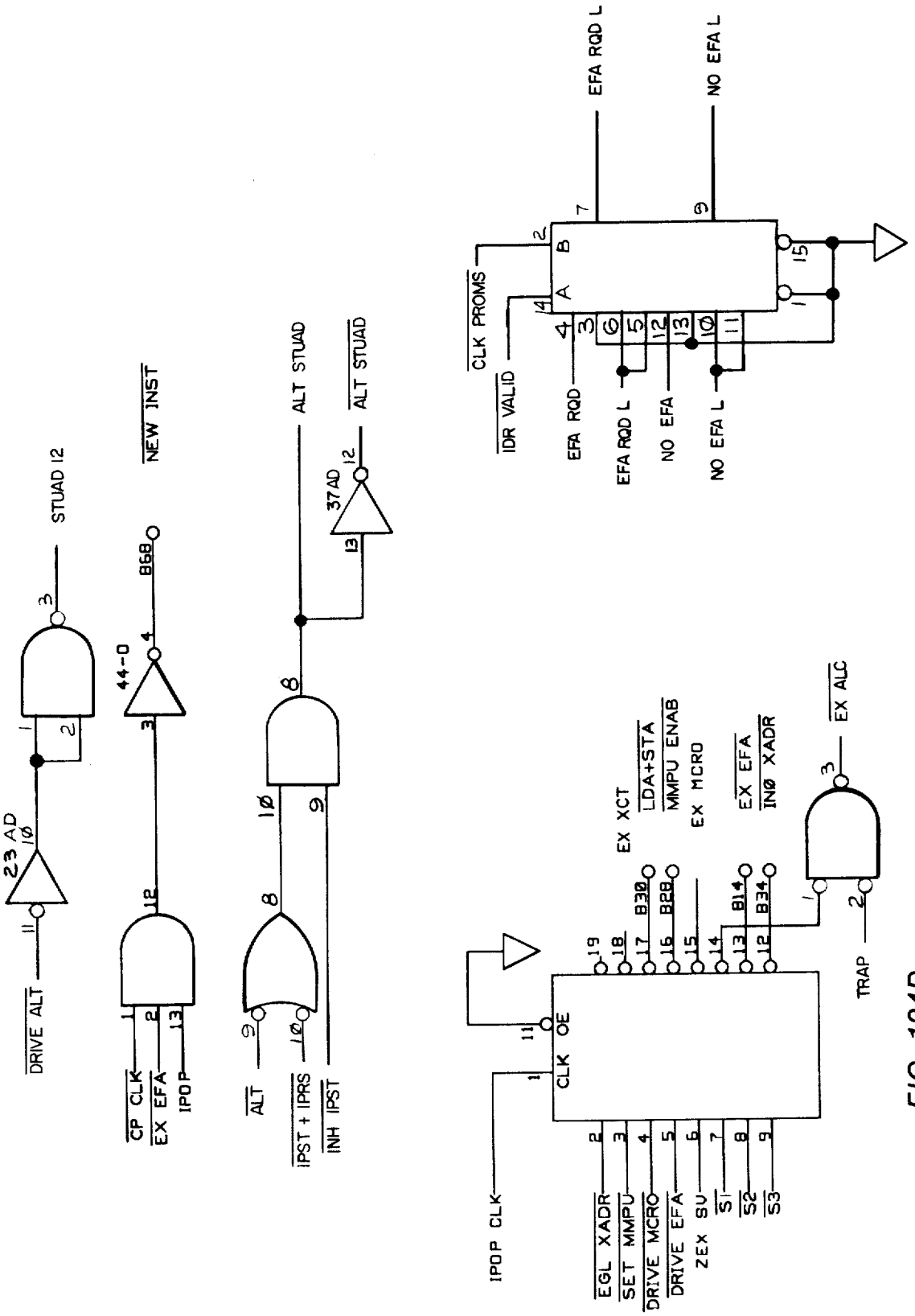

STUAD 12
23 AD
DRIVE ALT
NEW INST
B68
44-D
CP CLK
EX EFA
IPOP
ALT STUAD
37 AD
ALT
IPST + IPRS
INH IPST
EFA RQD L
NO EFA L
CLK PROMS
IDR VALID
EFA RQD
EFA RQD L
NO EFA
NO EFA L
EX XCT
LDA+STA
MMPU ENAB
EX MCRO
EX EFA
INØ XADR
EX ALC
B30
B28
B14
B34
TRAP
IPOP CLK
CLK
OE
EGL XADR
SET MMPU
DRIVE MCRO
DRIVE EFA
ZEX SU
S1
S2
S3

FIG. 124D

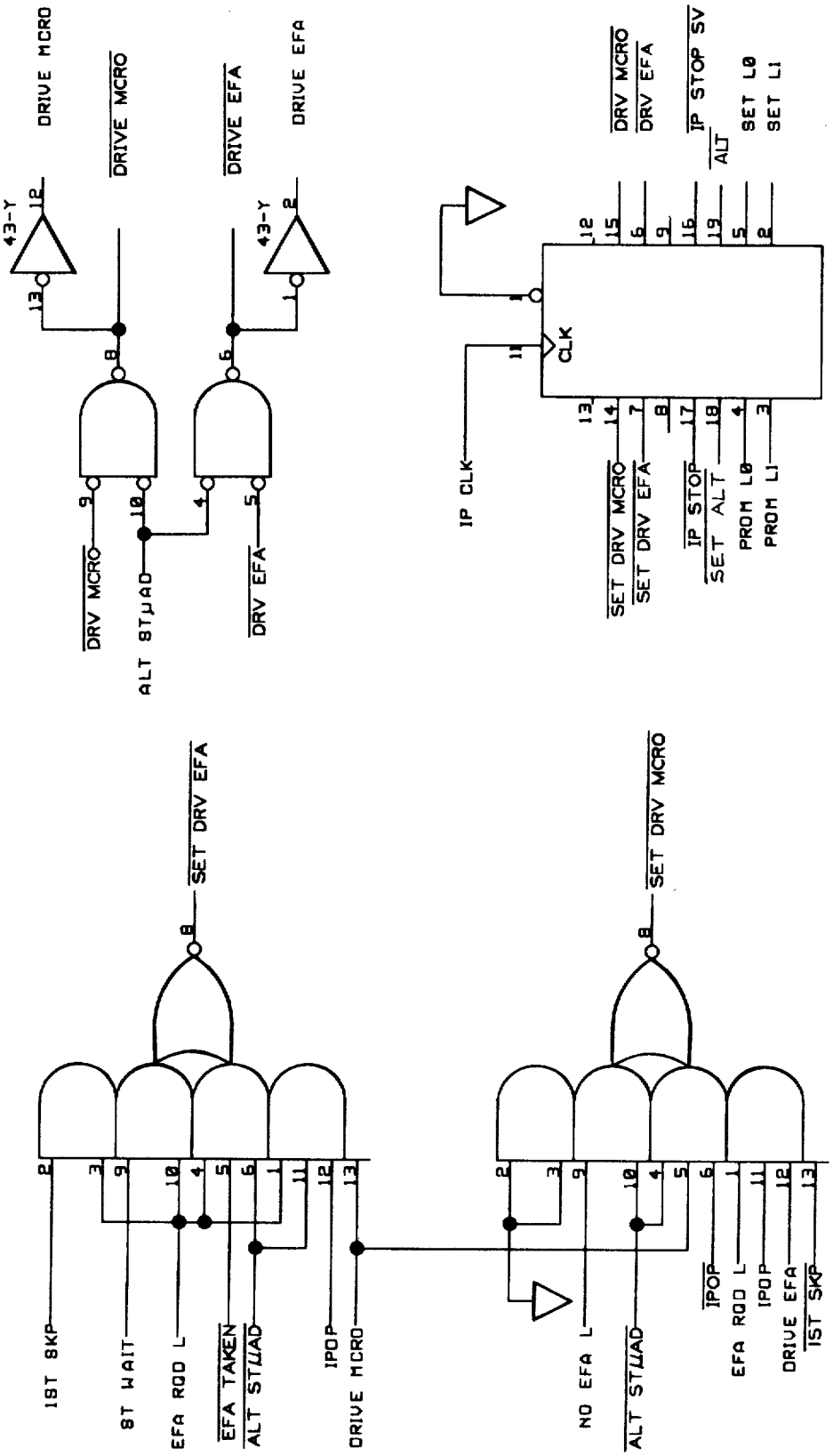
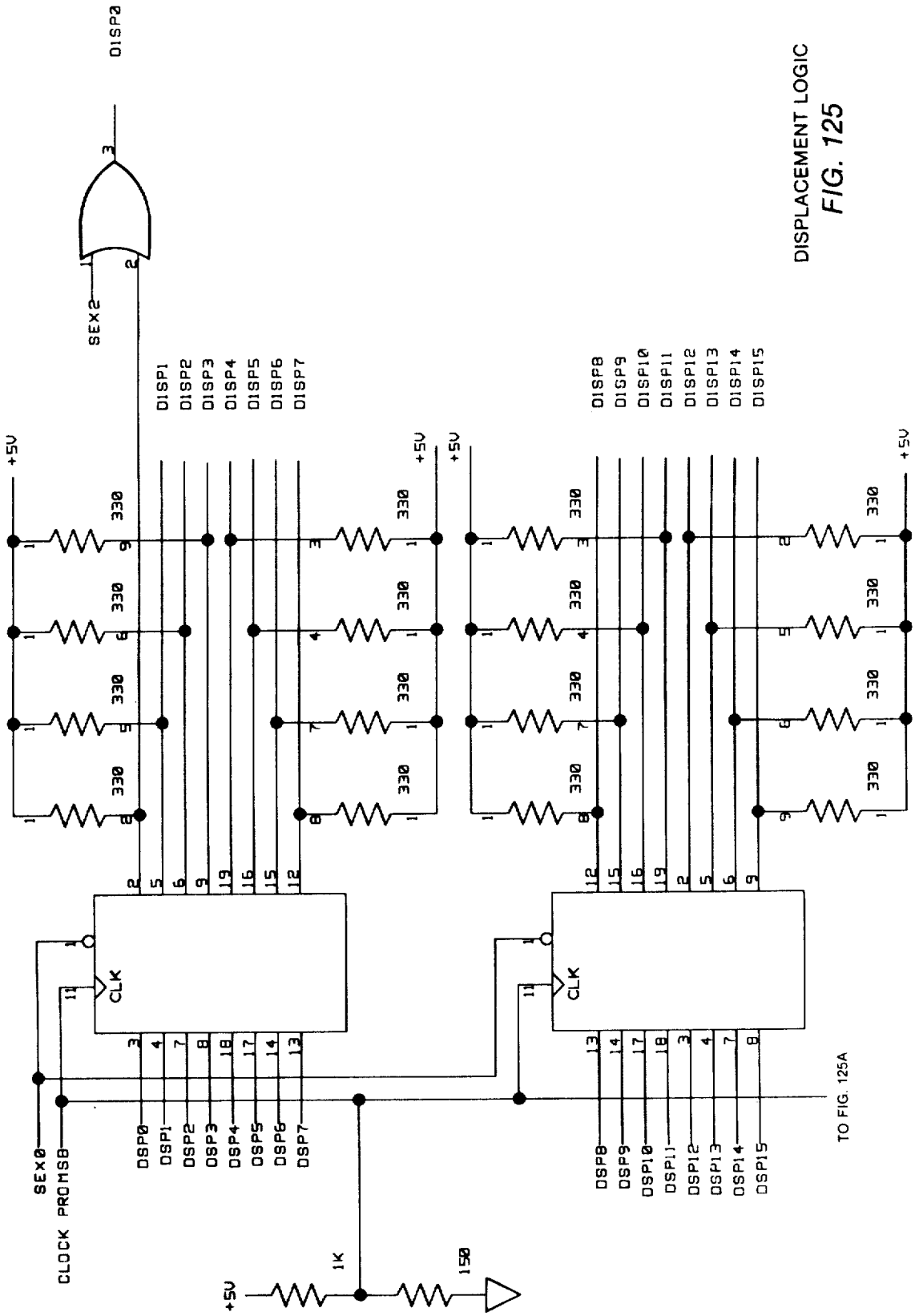

DRIVE MCRO
43-Y
DRIVE MCRO
DRV MCRO
ALT STμAD
DRIVE EFA
43-Y
DRIVE EFA
DRV EFA
IP CLK
CLK
DRV MCRO
DRV EFA
IP STOP SV
ALT
SET L0
SET L1
SET DRV MCRO
SET DRV EFA
IP STOP
SET ALT
PROM L0
PROM L1
SET DRV EFA
SET DRV MCRO
IST SKP
ST WAIT
EFA RQD L
EFA TAKEN
ALT STμAD
IPOP
DRIVE MCRO
NO EFA L
ALT STμAD
IPOP
EFA RQD L
IPOP
DRIVE EFA
IST SKP

FIG. 124E



DISPLACEMENT LOGIC
FIG. 125

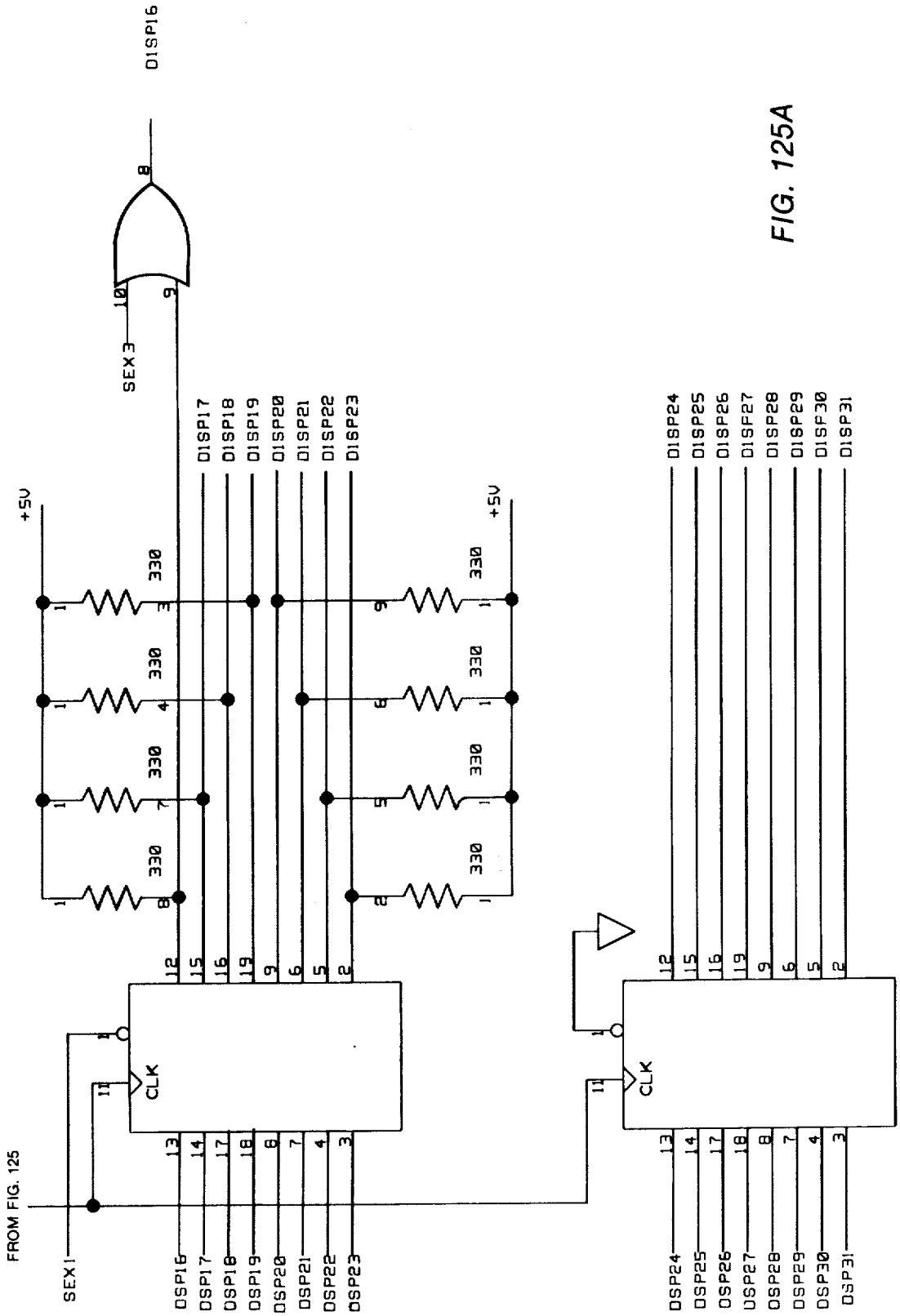
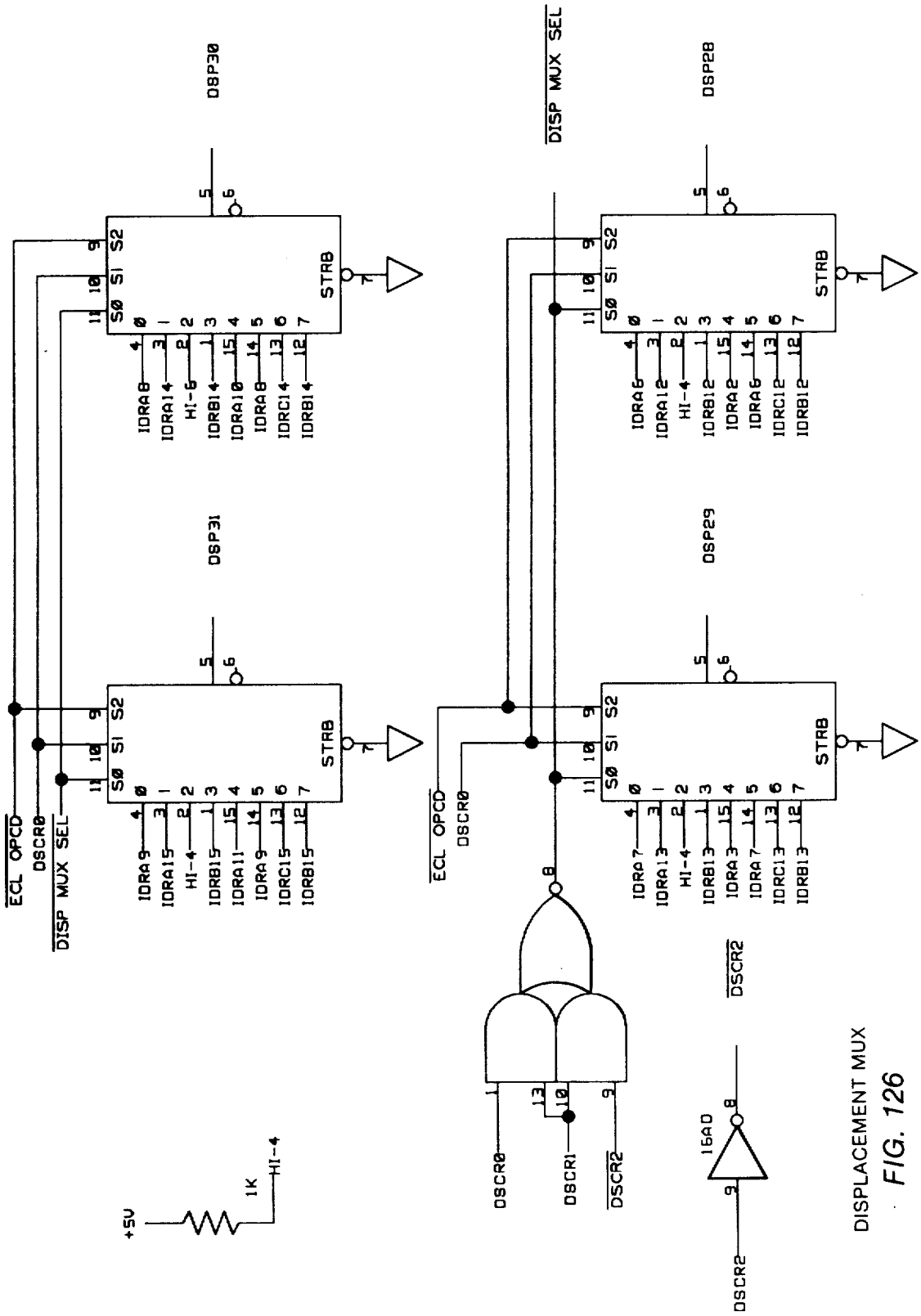

FROM FIG. 125
SEX1
SEX3
DISP16
DISP17
DISP18
DISP19
DISP20
DISP21
DISP22
DISP23
DISP24
DISP25
DISP26
DISP27
DISP28
DISP29
DISP30
DISP31
OSP16
OSP17
OSP18
OSP19
OSP20
OSP21
OSP22
OSP23
OSP24
OSP25
OSP26
OSP27
OSP28
OSP29
OSP30
OSP31
CLK
CLK
+5V
+5V
330
330
330
330
330
330
330
330

FIG. 125A



DISPLACEMENT MUX

FIG. 126

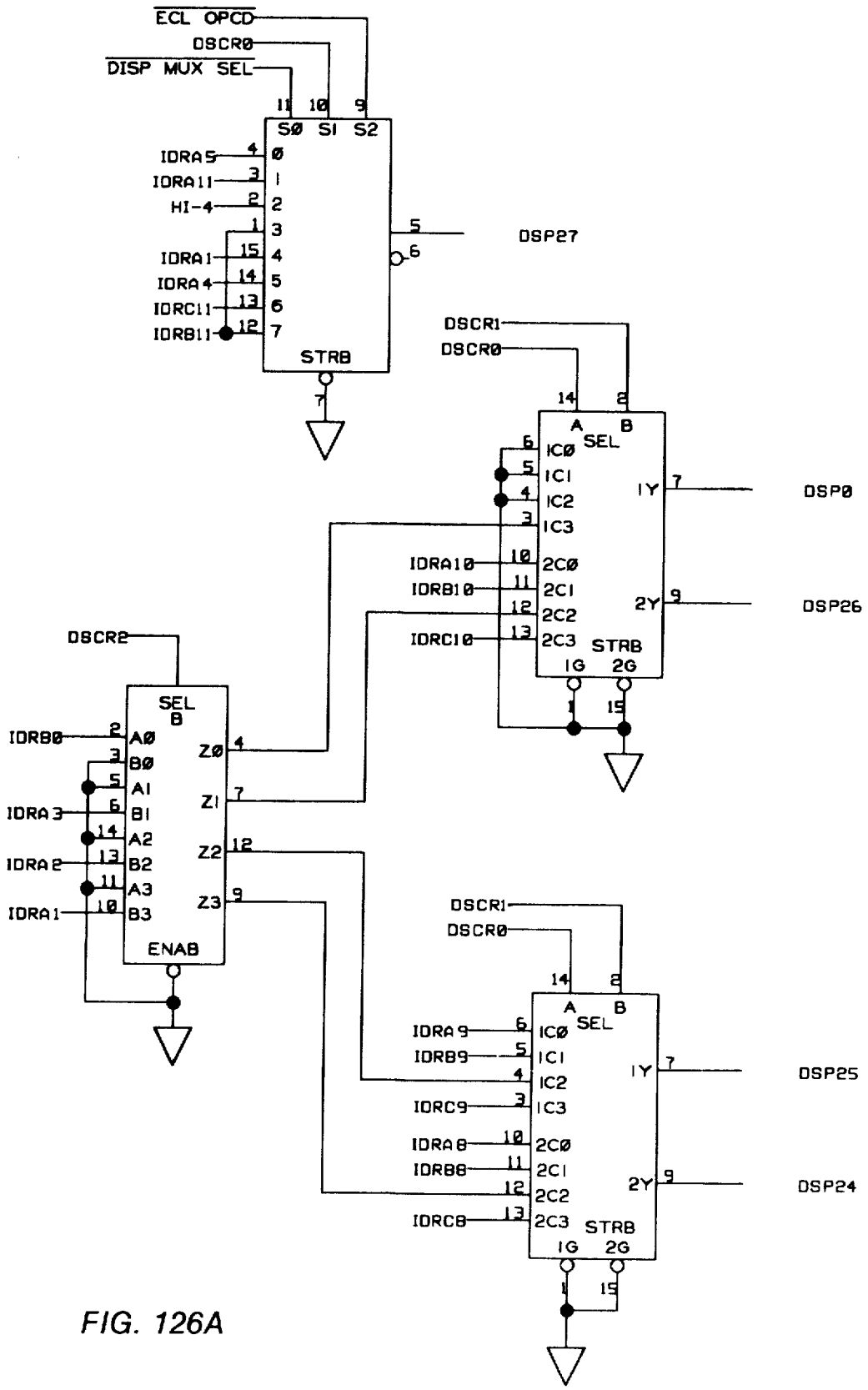

ECL OPCD
DSCR0
DISP MUX SEL
S0 S1 S2
IDRA5
IDRA11
HI-4
IDRA1
IDRA4
IDRC11
IDRB11
STRB
DSP27
DSCR1
DSCR0
A B
SEL
IC0
IC1
IC2
IC3
1Y
DSP0
IDRA10
IDRB10
IDRC10
2C0
2C1
2C2
2C3
2Y
DSP26
STRB
1G 2G
DSCR2
SEL B
IDRB0
IDRA3
IDRA2
IDRA1
A0
B0
A1
B1
A2
B2
A3
B3
Z0
Z1
Z2
Z3
ENAB
DSCR1
DSCR0
A B
SEL
IDRA9
IDRB9
IDRC9
IDRA8
IDRB8
IDRC8
IC0
IC1
IC2
IC3
2C0
2C1
2C2
2C3
1Y
DSP25
2Y
DSP24
STRB
1G 2G

FIG. 126A

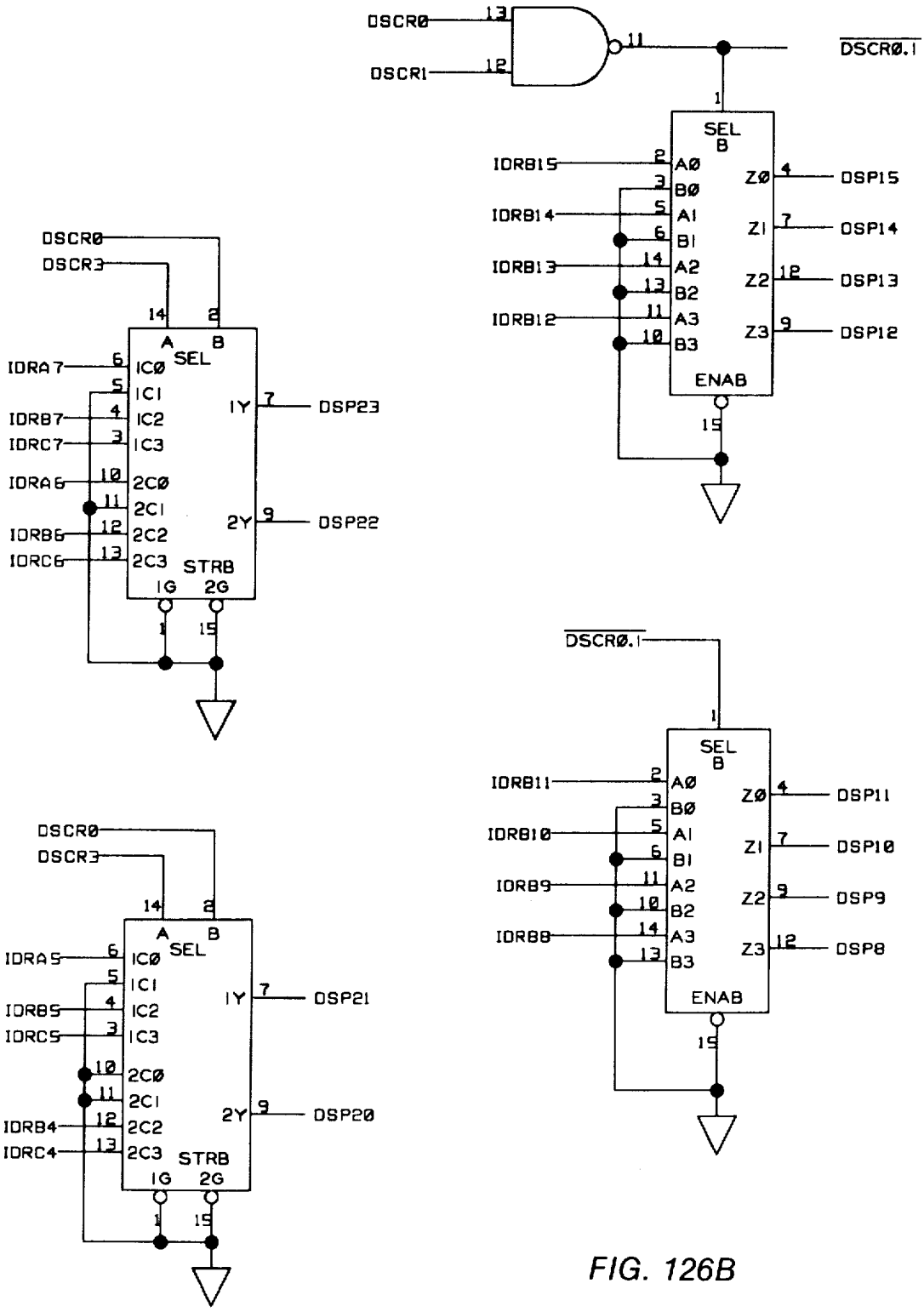

DSCR0
DSCR1
DSCR0.1
SEL
B
IDRB15
IDRB14
IDRB13
IDRB12
A0
B0
A1
B1
A2
B2
A3
B3
Z0
Z1
Z2
Z3
DSP15
DSP14
DSP13
DSP12
ENAB
DSCR0
DSCR3
A
B
SEL
IDRA7
IC0
IC1
IC2
IC3
IDRB7
IDRC7
IDRA6
2C0
2C1
2C2
2C3
IDRB6
IDRC6
1Y
2Y
DSP23
DSP22
STRB
1G
2G
DSCR0.1
IDRB11
IDRB10
IDRB9
IDRB8
DSP11
DSP10
DSP9
DSP8
IDRA5
IDRB5
IDRC5
IDRB4
IDRC4
DSP21
DSP20

FIG. 126B

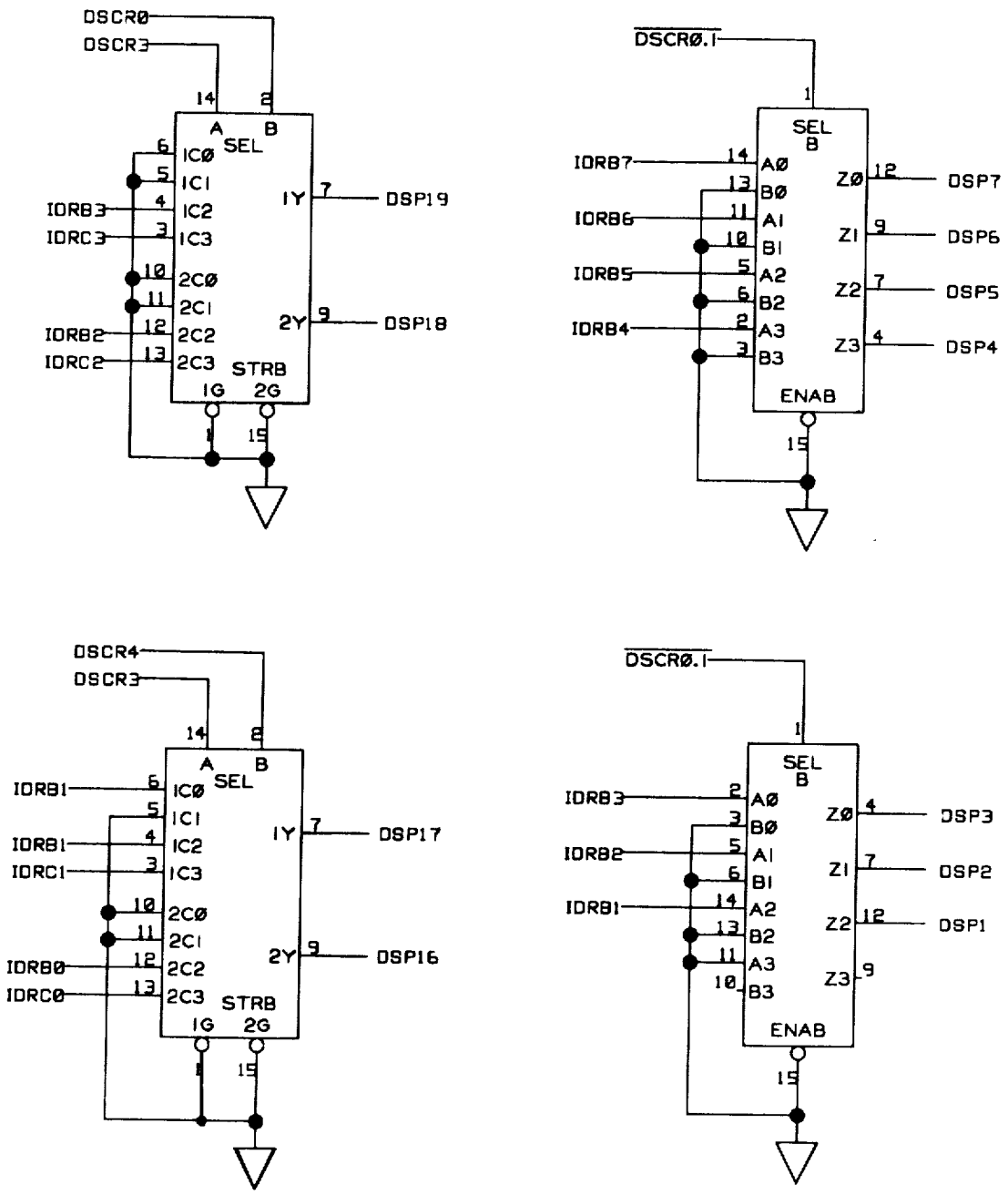
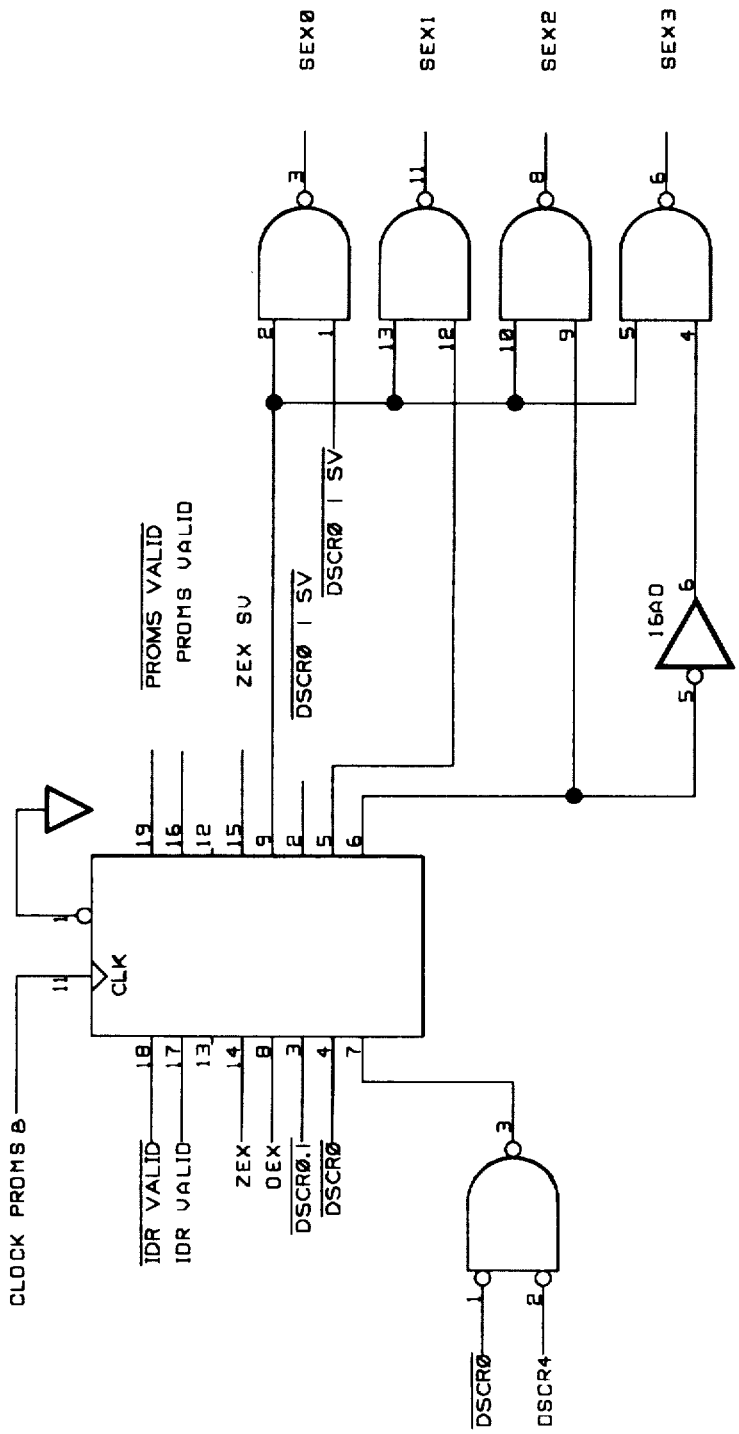
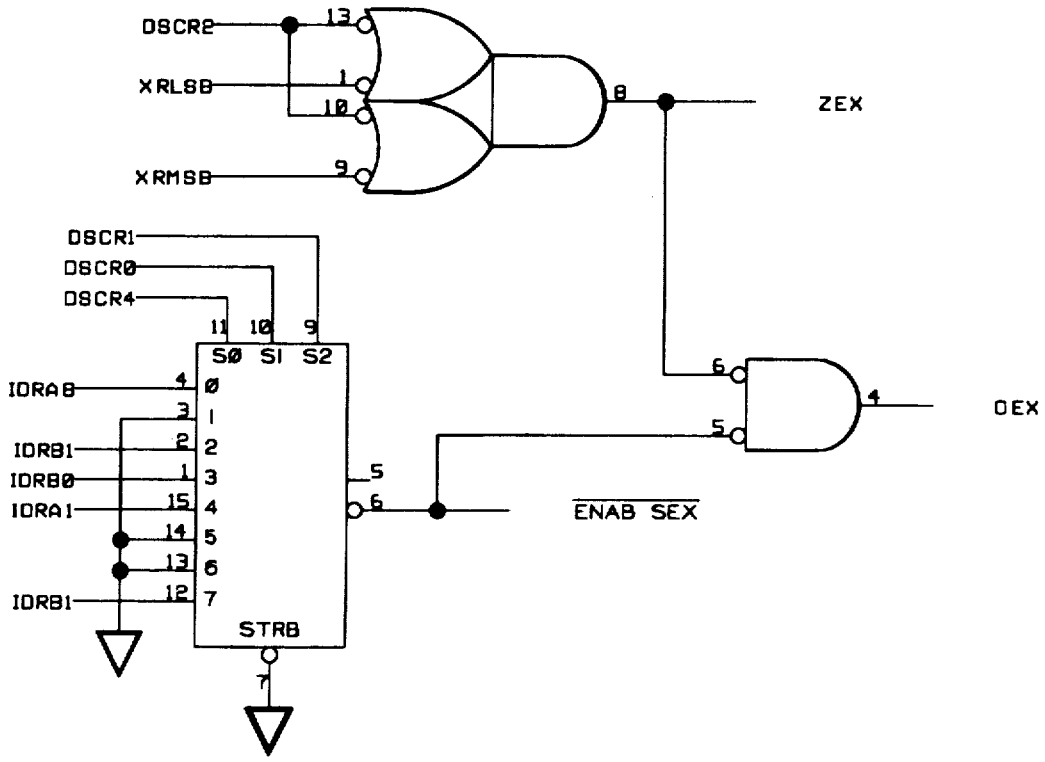

DSCRØ
DSCR3
14
2
A
B
SEL
6
ICØ
5
IC1
IORB3
4
IC2
IORC3
3
IC3
1Y
7
DSP19
10
2CØ
11
2C1
IORB2
12
2C2
IORC2
13
2C3
2Y
9
DSP18
STRB
1G
2G
1
15
DSCRØ.I
1
SEL
B
IORB7
14
AØ
13
BØ
ZØ
12
DSP7
IORB6
11
A1
10
B1
Z1
9
DSP6
IORB5
5
A2
6
B2
Z2
7
DSP5
IORB4
2
A3
3
B3
Z3
4
DSP4
ENAB
15
DSCR4
DSCR3
14
2
A
B
SEL
IORB1
6
ICØ
5
IC1
IORB1
4
IC2
IORC1
3
IC3
1Y
7
DSP17
10
2CØ
11
2C1
IORBØ
12
2C2
IORCØ
13
2C3
2Y
9
DSP16
STRB
1G
2G
1
15
DSCRØ.I
1
SEL
B
IORB3
2
AØ
3
BØ
ZØ
4
DSP3
IORB2
5
A1
6
B1
Z1
7
DSP2
IORB1
14
A2
13
B2
Z2
12
DSP1
11
A3
10
B3
Z3
9
ENAB
15

FIG. 126C



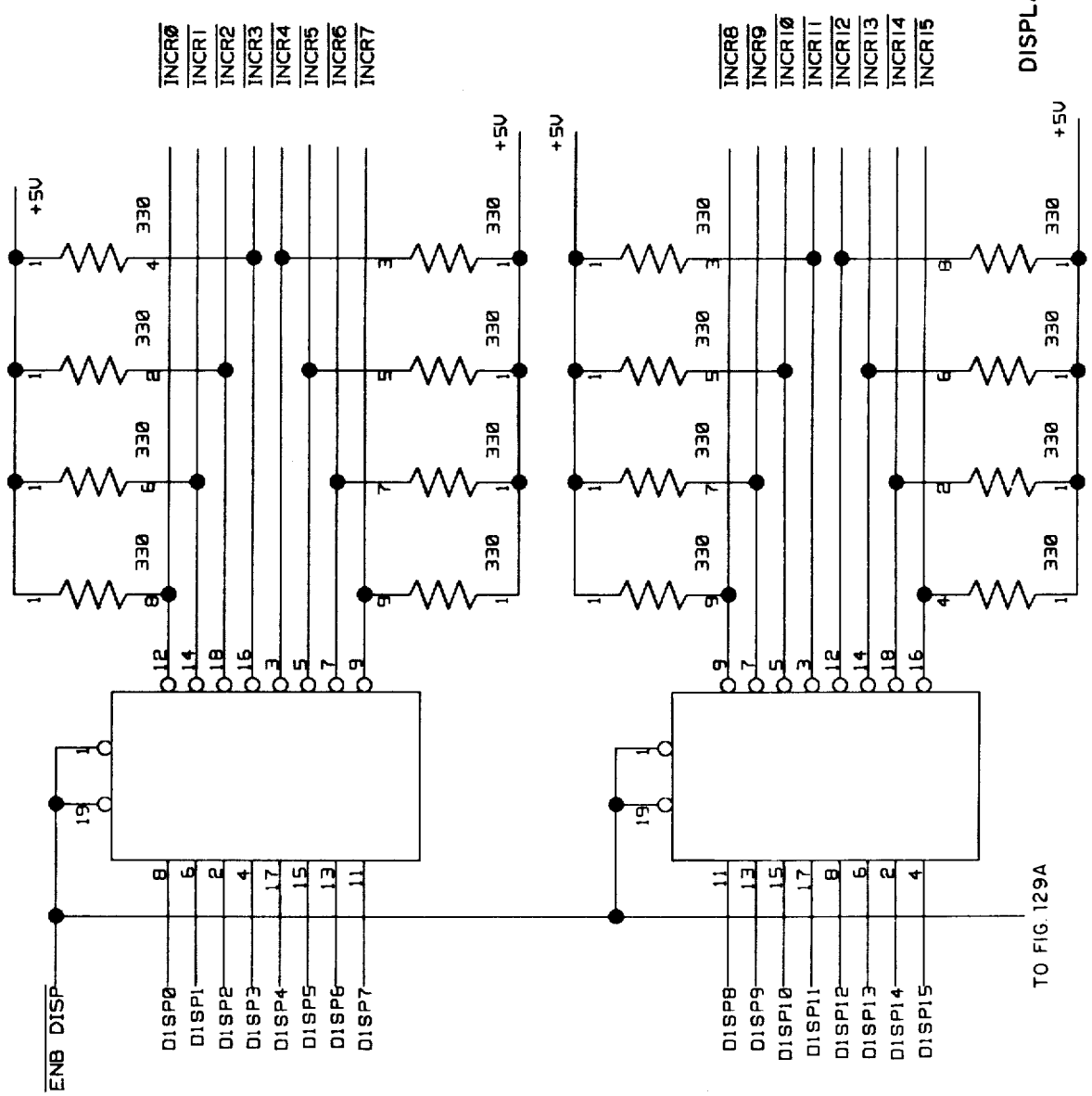
SEX LOGIC

FIG. 127



ZERO/ONES EXTEND LOGIC

FIG. 128



DISPLACEMENT/INCREMENT BUFFER

FIG. 129

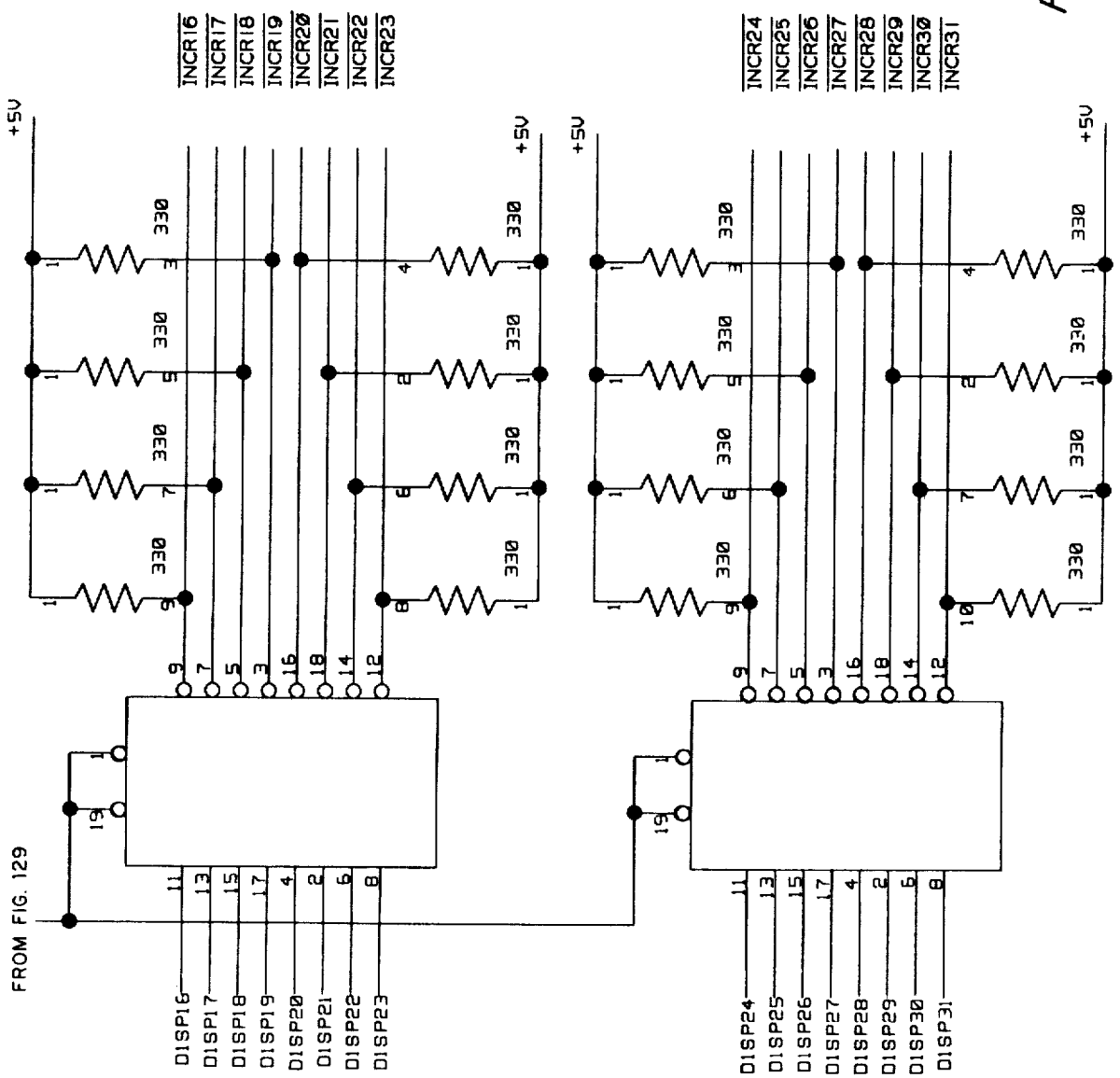
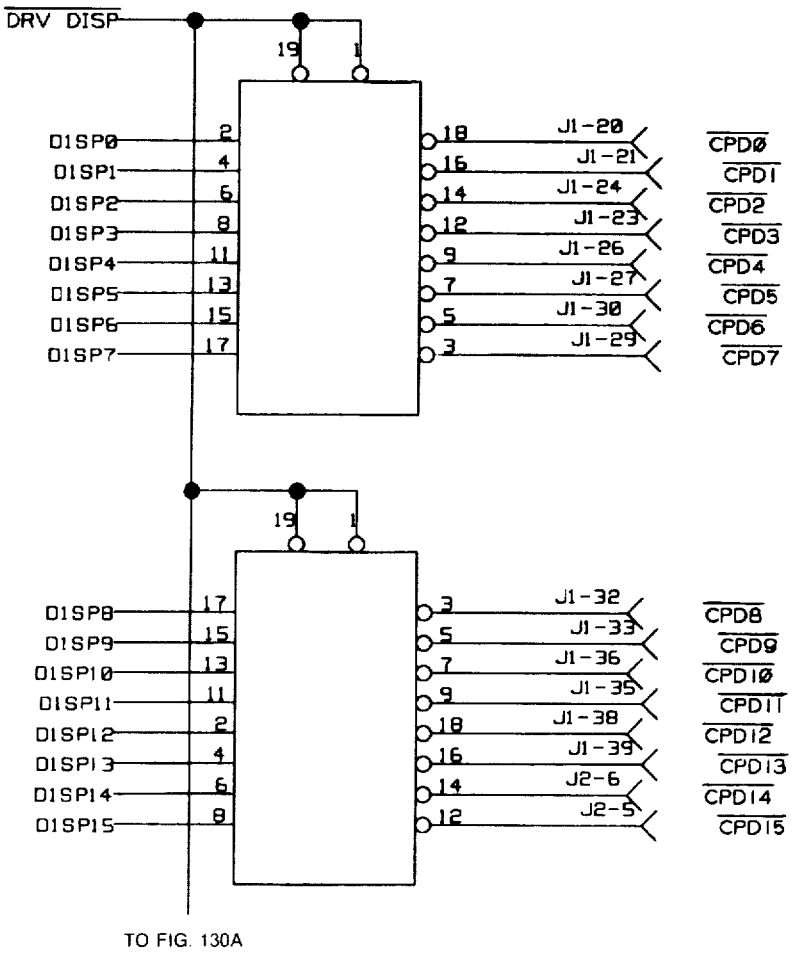

FROM FIG. 129
+5V
330
INCR16
INCR17
INCR18
INCR19
INCR20
INCR21
INCR22
INCR23
INCR24
INCR25
INCR26
INCR27
INCR28
INCR29
INCR30
INCR31
DISP16
DISP17
DISP18
DISP19
DISP20
DISP21
DISP22
DISP23
DISP24
DISP25
DISP26
DISP27
DISP28
DISP29
DISP30
DISP31

FIG. 129A



DISPLACEMENT LATCH AND DRIVERS 153

FIG. 130

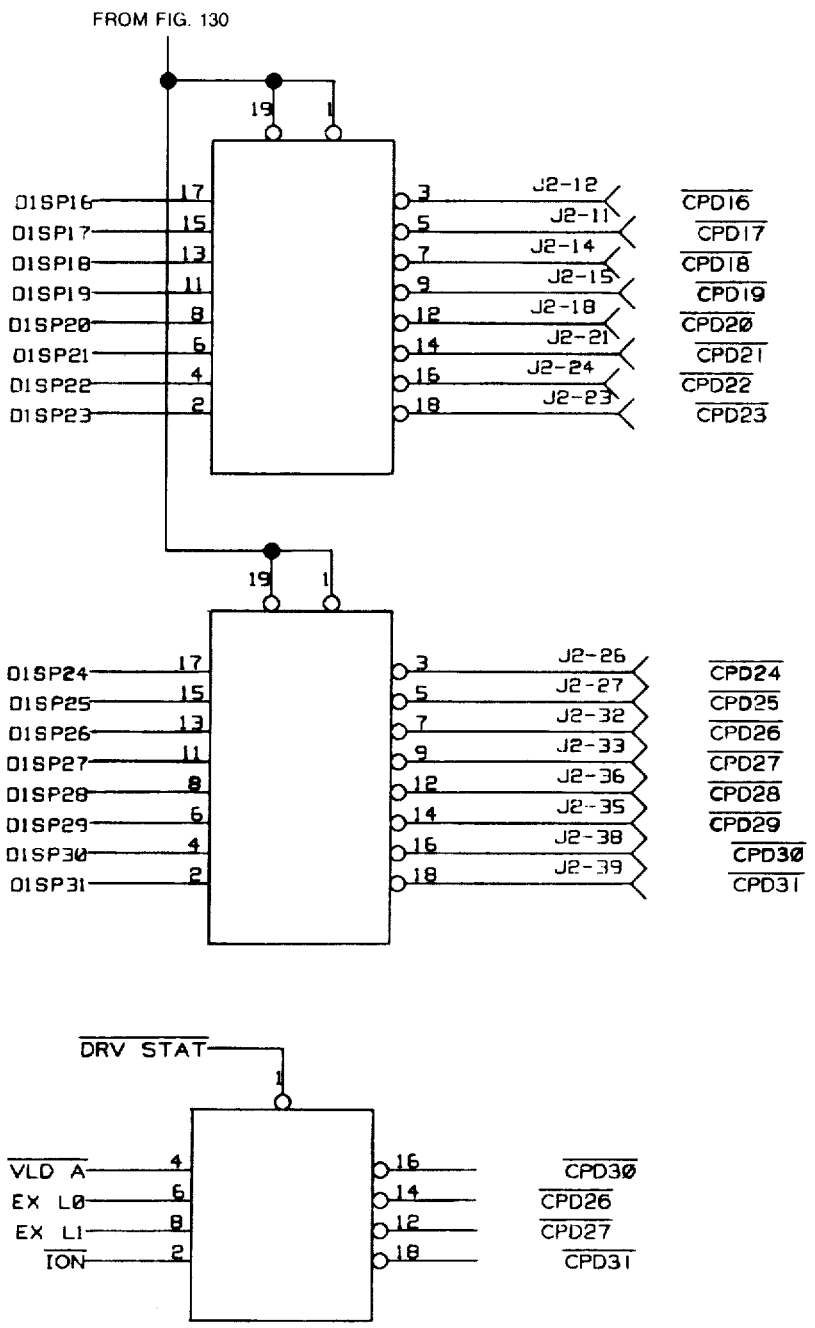
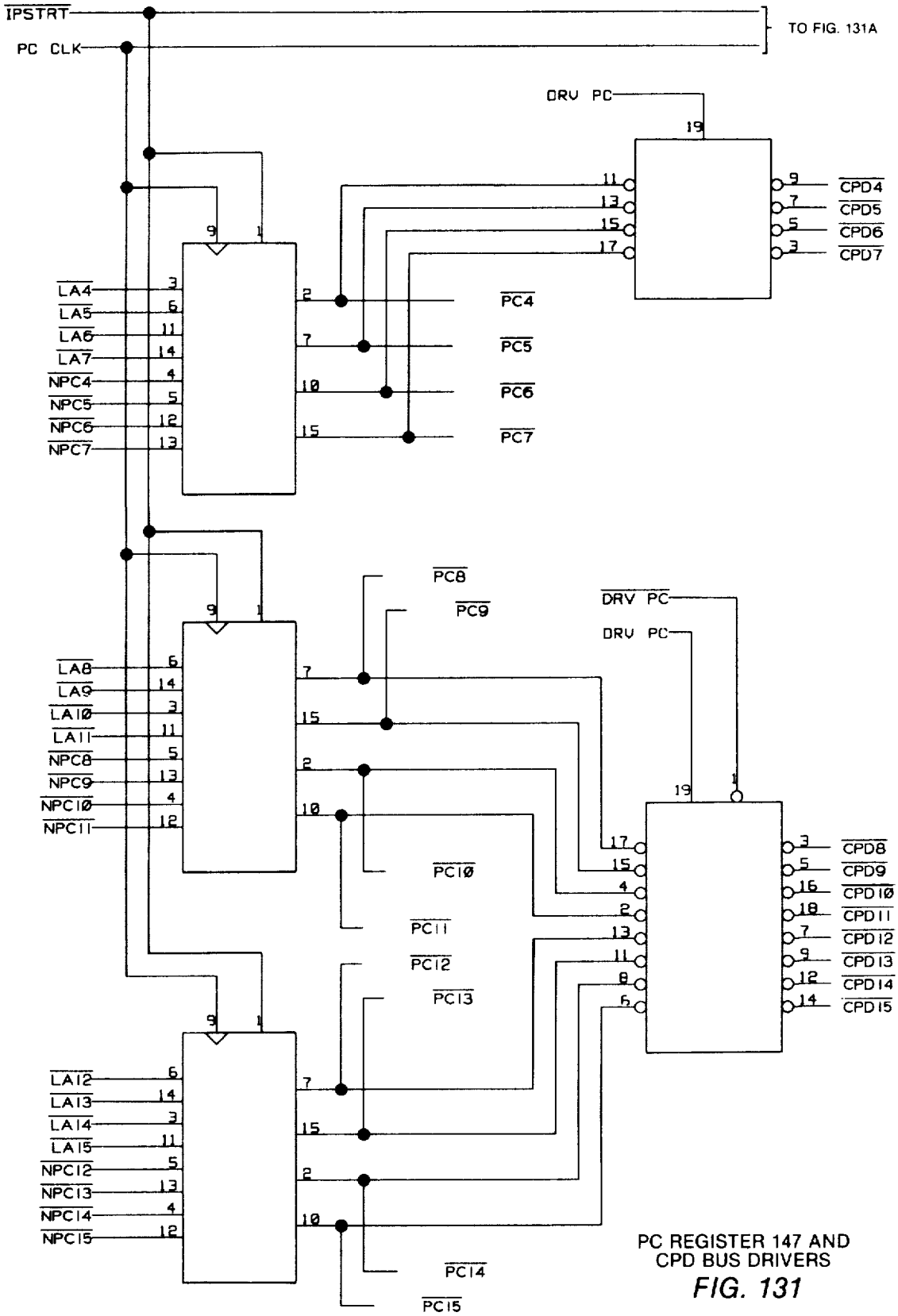

FROM FIG. 130
19
1
DISP16
DISP17
DISP18
DISP19
DISP20
DISP21
DISP22
DISP23
17
15
13
11
8
6
4
2
3
5
7
9
12
14
16
18
J2-12
J2-11
J2-14
J2-15
J2-18
J2-21
J2-24
J2-23
CPD16
CPD17
CPD18
CPD19
CPD20
CPD21
CPD22
CPD23
19
1
DISP24
DISP25
DISP26
DISP27
DISP28
DISP29
DISP30
DISP31
17
15
13
11
8
6
4
2
3
5
7
9
12
14
16
18
J2-26
J2-27
J2-32
J2-33
J2-36
J2-35
J2-38
J2-39
CPD24
CPD25
CPD26
CPD27
CPD28
CPD29
CPD30
CPD31
DRV STAT
1
VLD A
EX L0
EX L1
ION
4
6
8
2
16
14
12
18
CPD30
CPD26
CPD27
CPD31

FIG. 130A



PC REGISTER 147 AND CPD BUS DRIVERS

FIG. 131

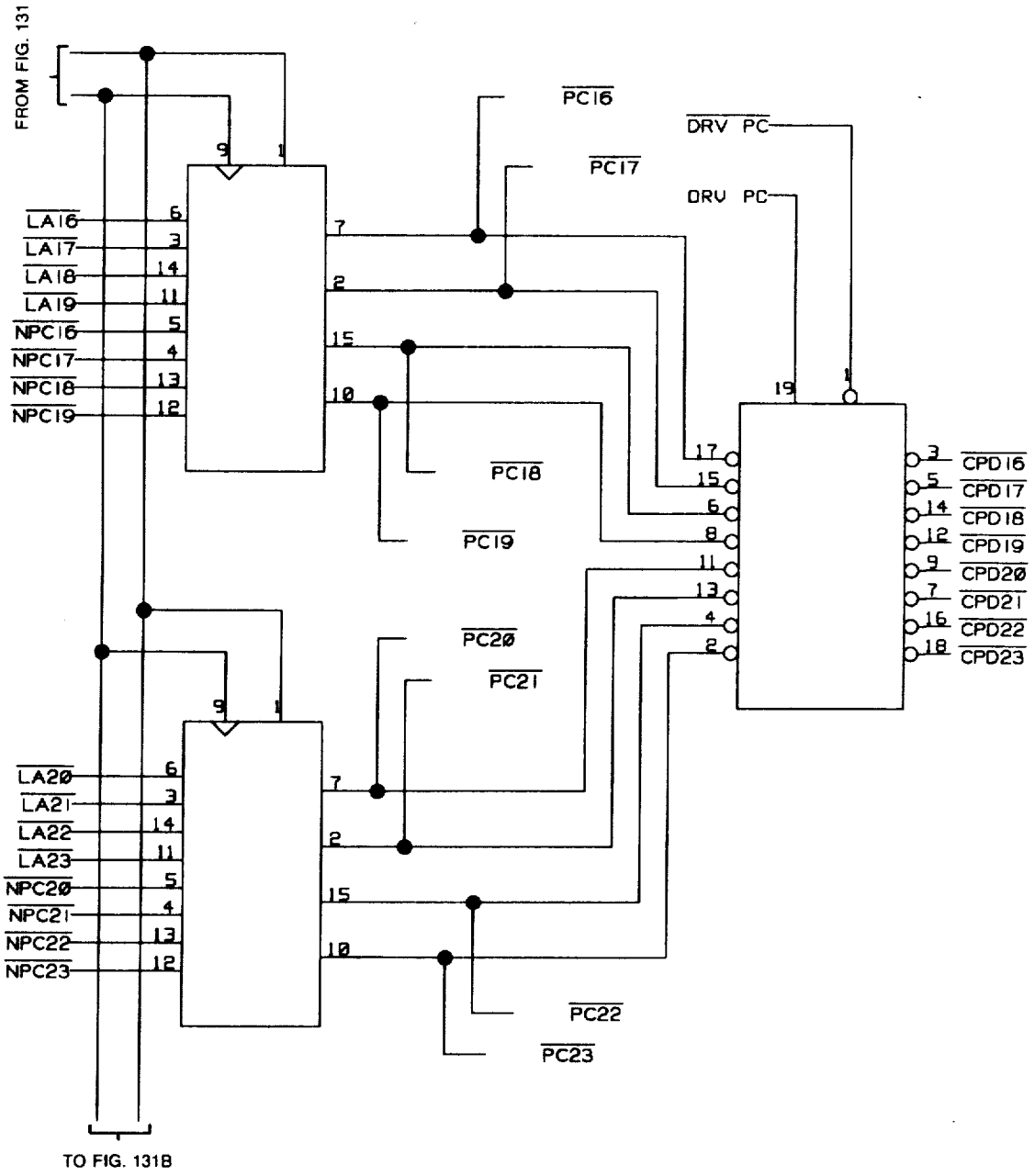

FROM FIG. 131
LA16
LA17
LA18
LA19
NPC16
NPC17
NPC18
NPC19
PC16
PC17
PC18
PC19
DRV PC
DRV PC
CPD16
CPD17
CPD18
CPD19
CPD20
CPD21
CPD22
CPD23
LA20
LA21
LA22
LA23
NPC20
NPC21
NPC22
NPC23
PC20
PC21
PC22
PC23
TO FIG. 131B

FIG. 131A

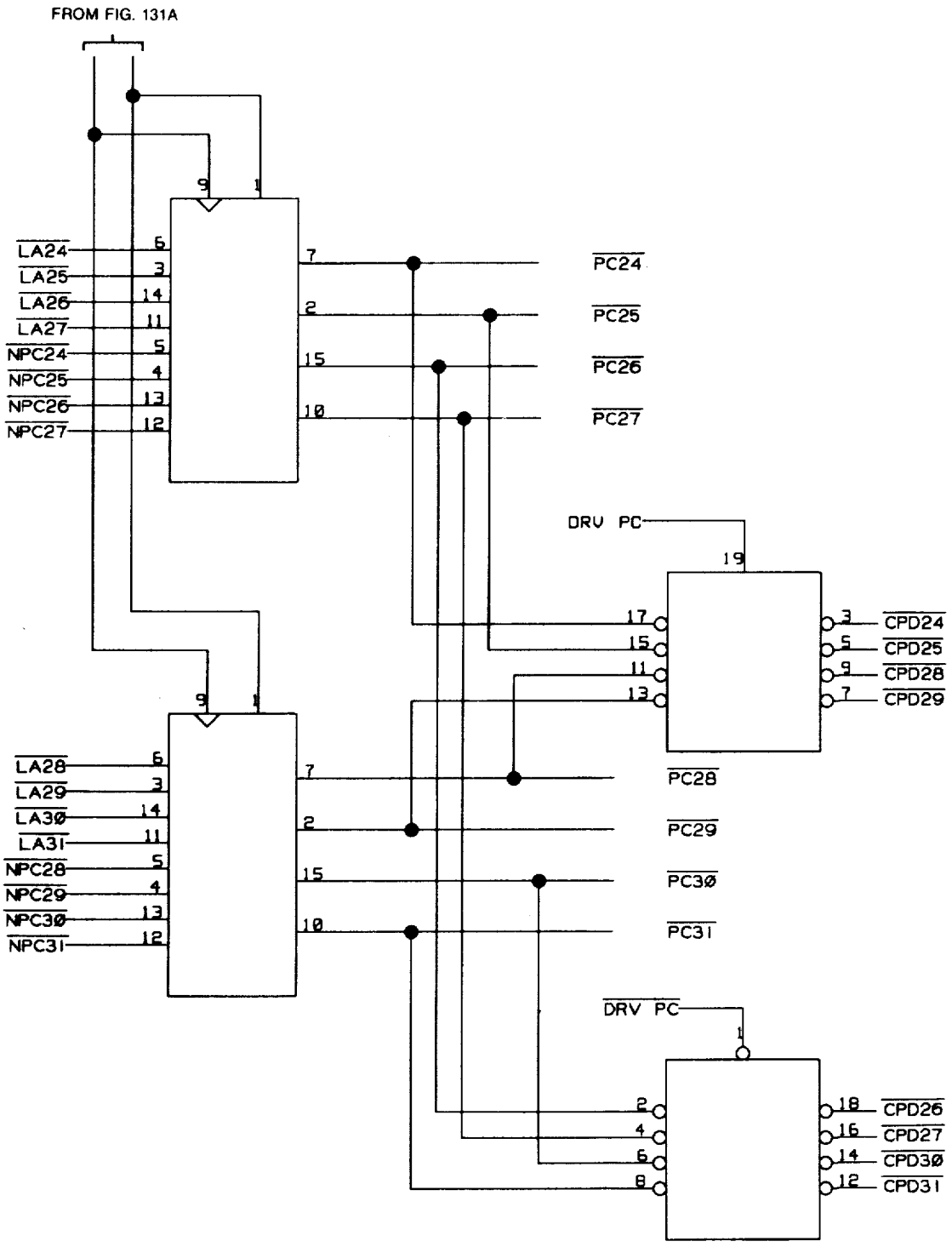
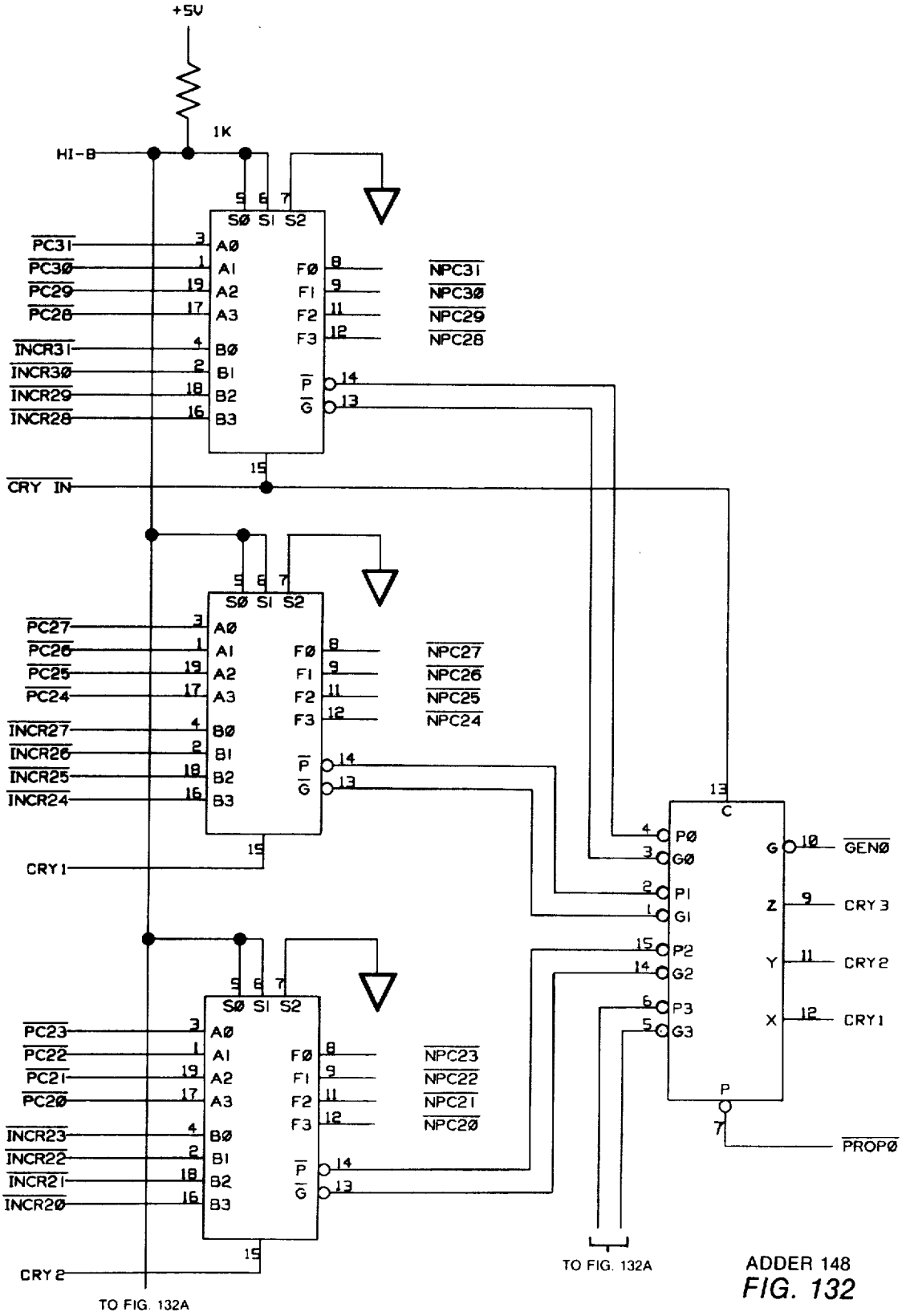

FROM FIG. 131A
LA24
LA25
LA26
LA27
NPC24
NPC25
NPC26
NPC27
PC24
PC25
PC26
PC27
DRV PC
CPD24
CPD25
CPD28
CPD29
LA28
LA29
LA30
LA31
NPC28
NPC29
NPC30
NPC31
PC28
PC29
PC30
PC31
DRV PC
CPD26
CPD27
CPD30
CPD31

FIG. 131B



ADDER 148
FIG. 132

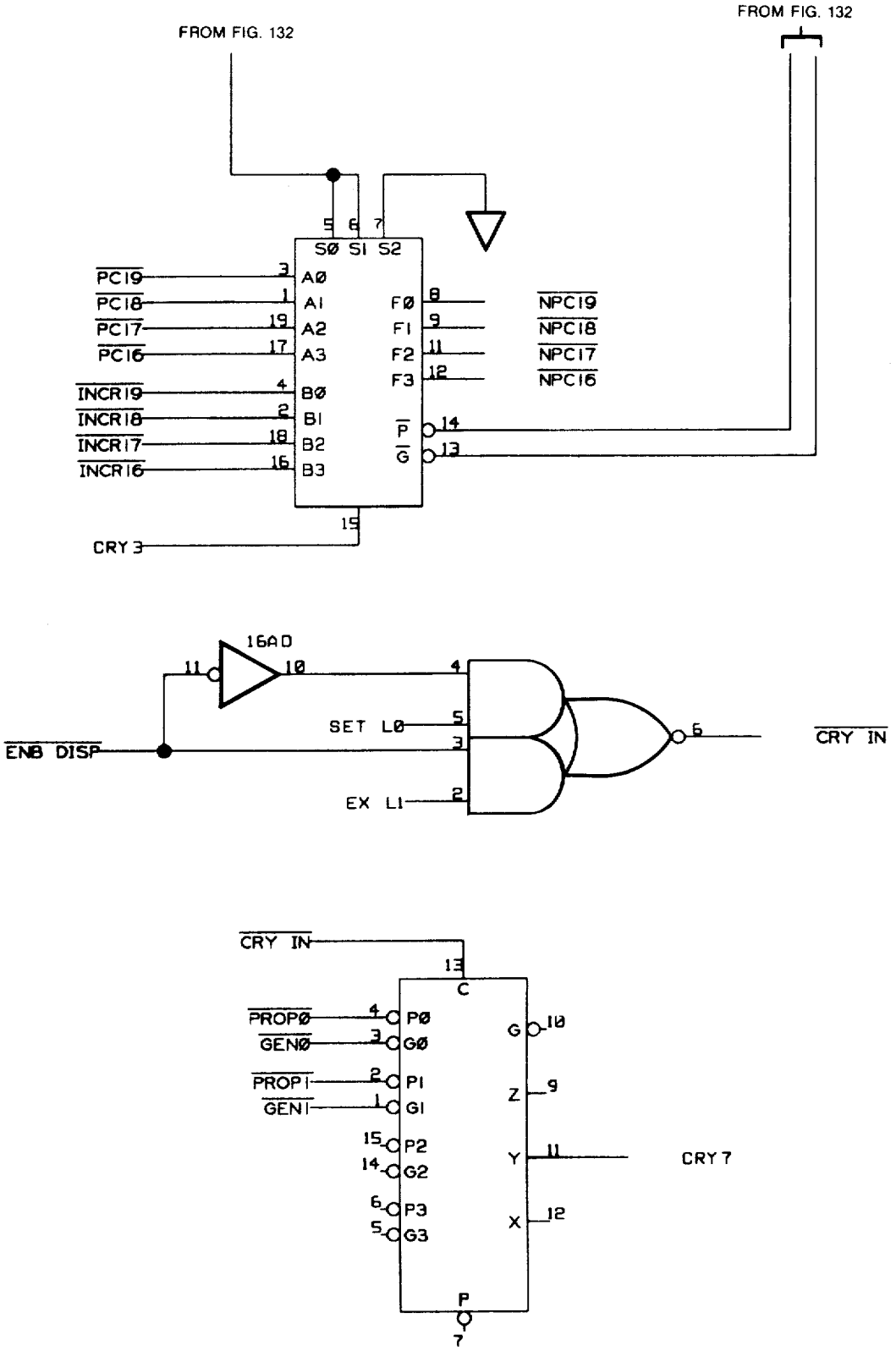

FROM FIG. 132
FROM FIG. 132
S0 S1 S2
A0
A1
A2
A3
B0
B1
B2
B3
F0
F1
F2
F3
P
G
PC19
PC18
PC17
PC16
INCR19
INCR18
INCR17
INCR16
NPC19
NPC18
NPC17
NPC16
CRY 3
16AD
ENB DISP
SET L0
EX L1
CRY IN
CRY IN
C
PROP0
GEN0
PROP1
GEN1
P0
G0
P1
G1
P2
G2
P3
G3
G
Z
Y
X
P
CRY 7

FIG. 132A

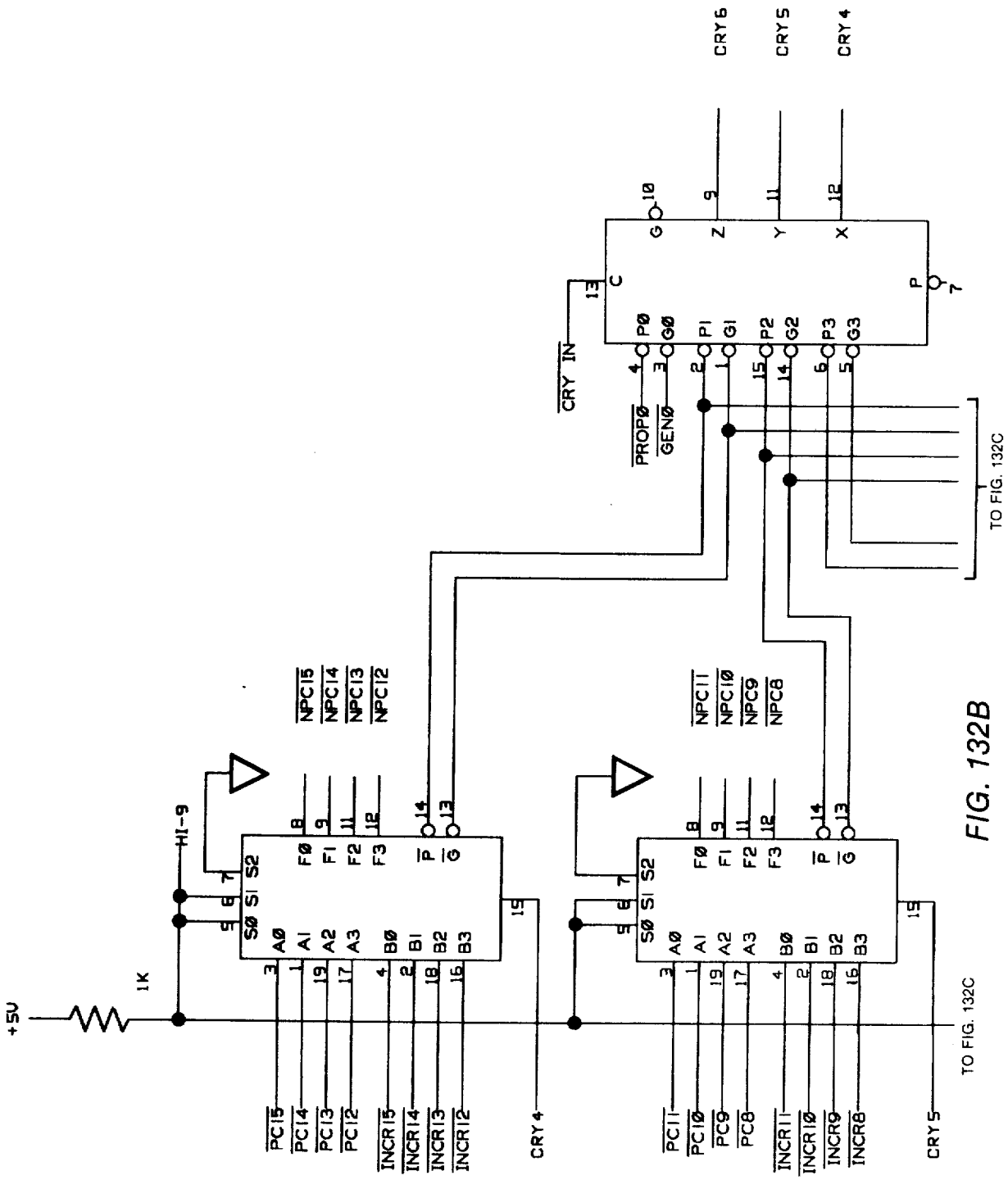

+5V
1K
HI-9
S0 S1 S2
A0 A1 A2 A3
B0 B1 B2 B3
F0 F1 F2 F3
P G
PC15 PC14 PC13 PC12
INCR15 INCR14 INCR13 INCR12
NPC15 NPC14 NPC13 NPC12
CRY4
PC11 PC10 PC9 PC8
INCR11 INCR10 INCR9 INCR8
NPC11 NPC10 NPC9 NPC8
CRY5
TO FIG. 132C
CRY IN
PROP0
GEN0
P0 G0 P1 G1 P2 G2 P3 G3
C G Z Y X P
CRY6
CRY5
CRY4
TO FIG. 132C

FIG. 132B

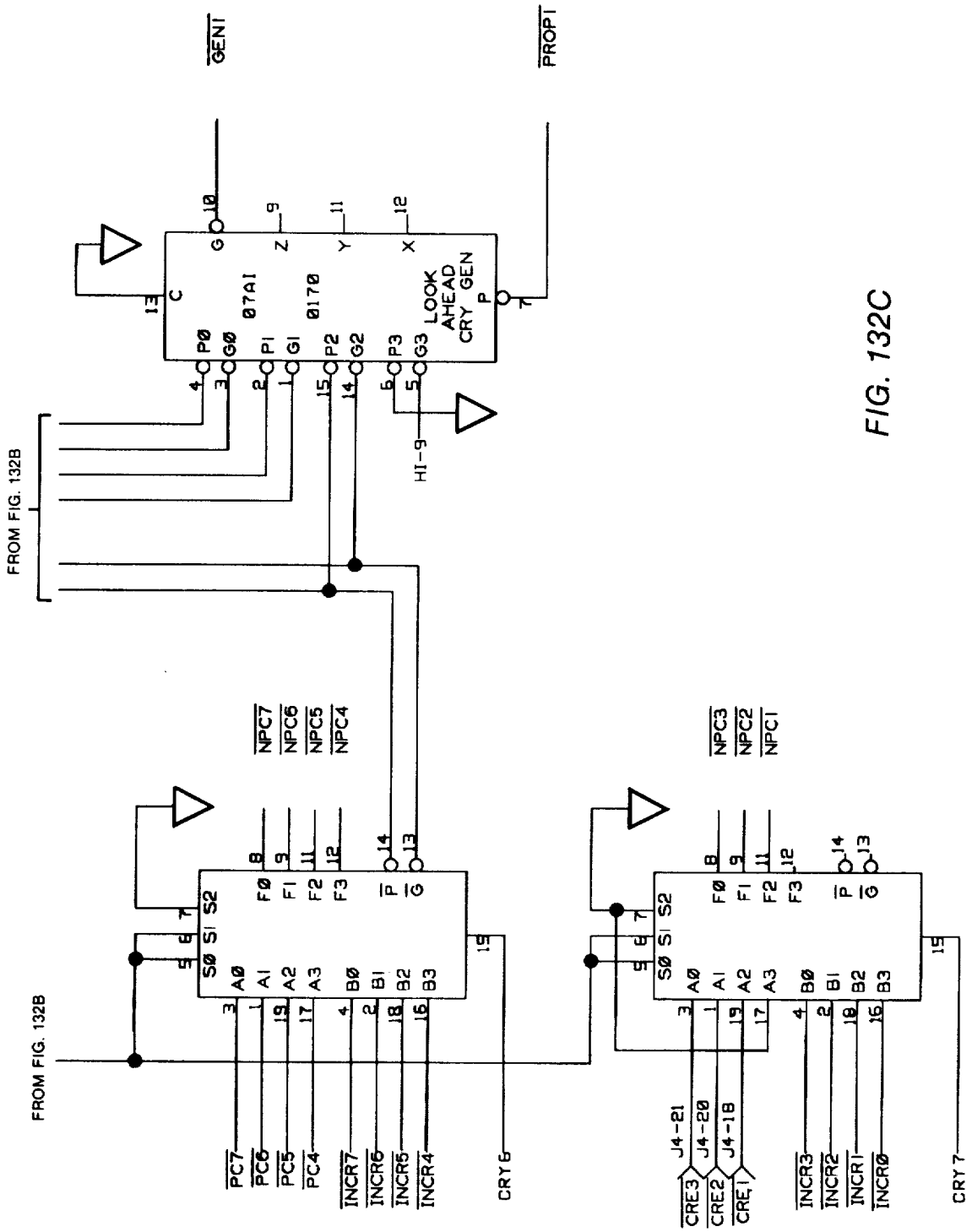
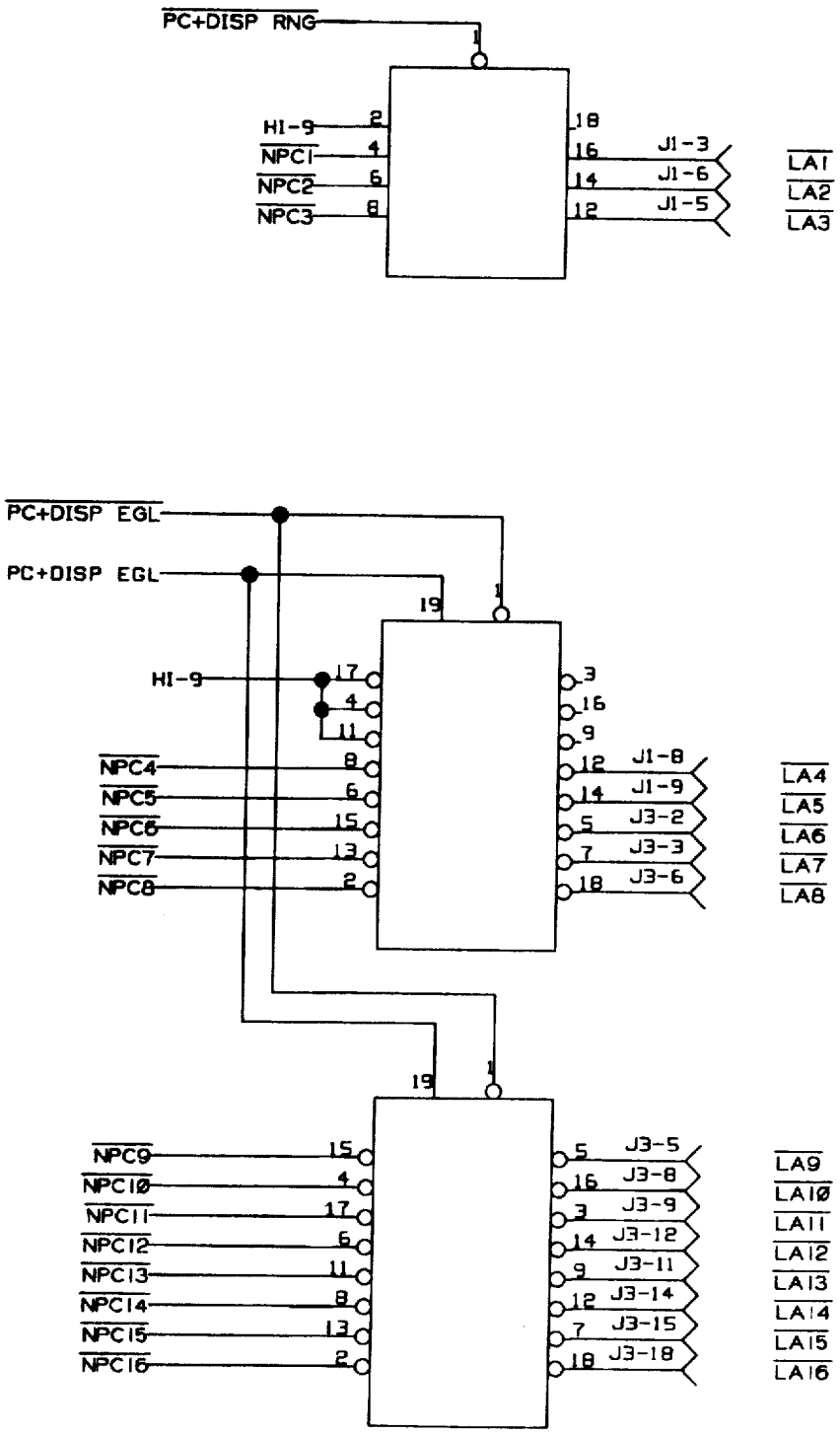

GENI
PROPI
FROM FIG. 132B
FROM FIG. 132B
LOOK AHEAD CRY GEN
07A1
0170
HI-9
NPC7
NPC6
NPC5
NPC4
NPC3
NPC2
NPC1
PC7
PC6
PC5
PC4
INCR7
INCR6
INCR5
INCR4
CRY6
J4-21
J4-20
J4-18
CRE3
CRE2
CRE1
INCR3
INCR2
INCR1
INCR0
CRY7

FIG. 132C



PC AND DISP LATCHES AND DRIVERS 149

FIG. 133

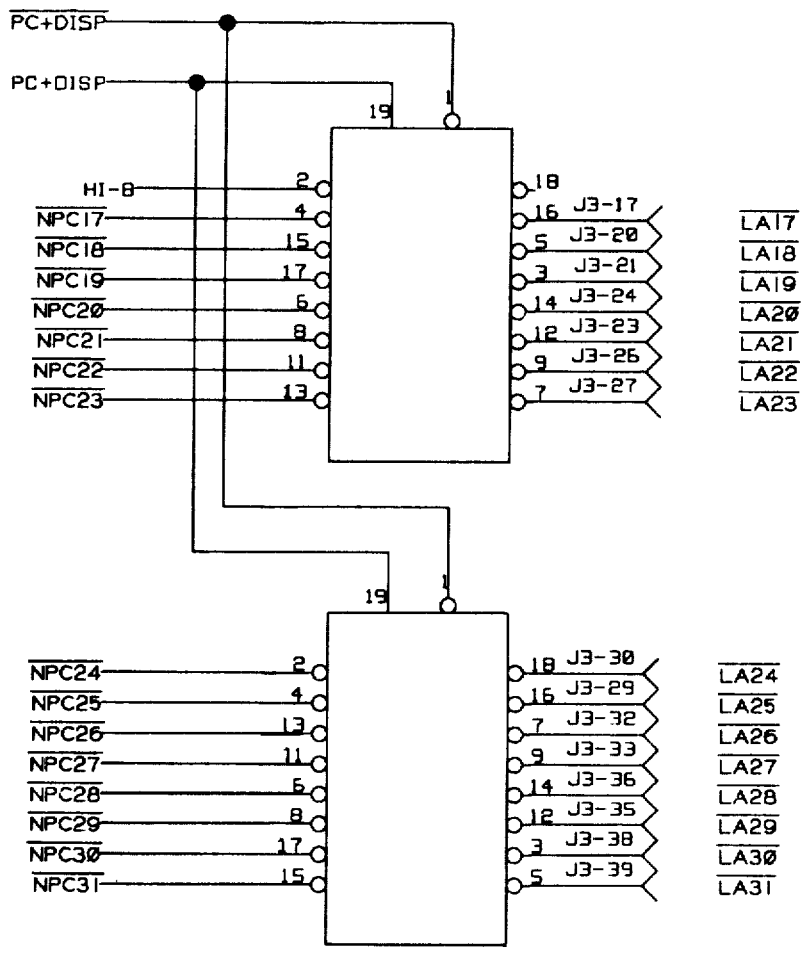
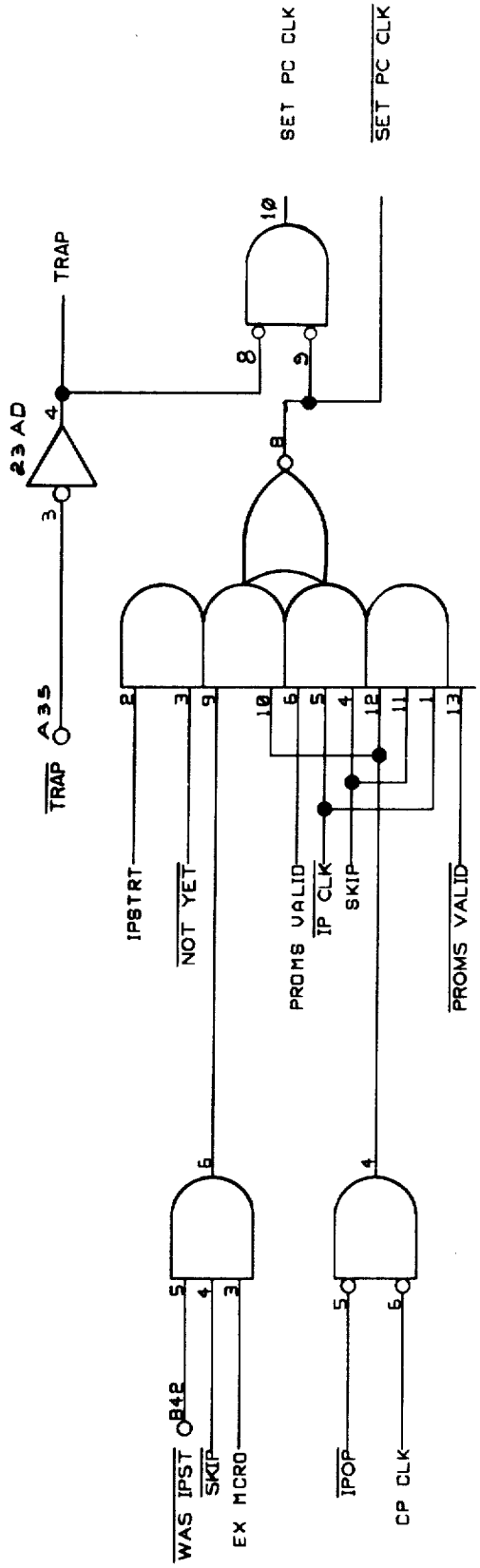
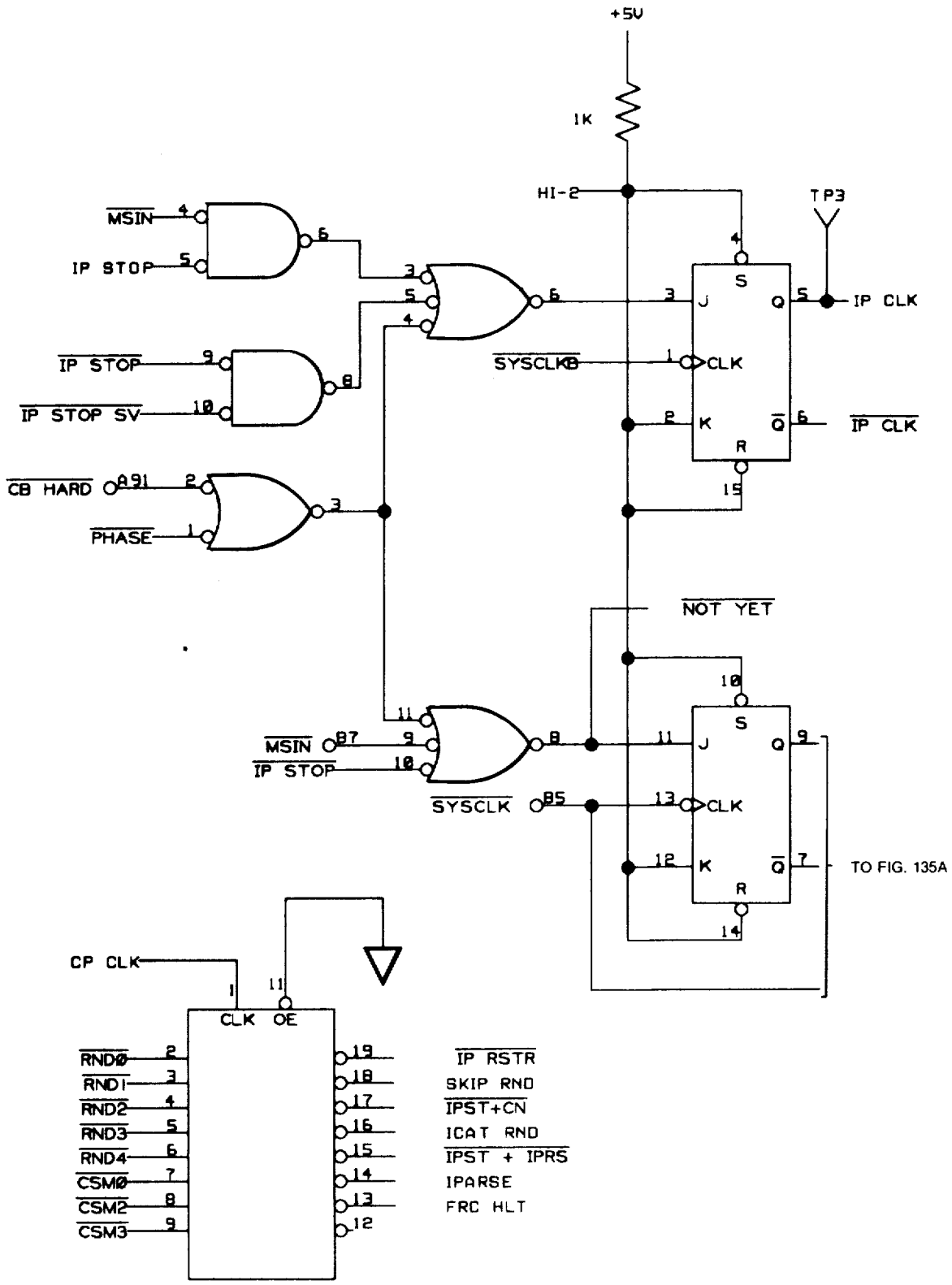

PC+DISP
PC+DISP
HI-8
NPC17
NPC18
NPC19
NPC20
NPC21
NPC22
NPC23
J3-17
J3-20
J3-21
J3-24
J3-23
J3-26
J3-27
LA17
LA18
LA19
LA20
LA21
LA22
LA23
NPC24
NPC25
NPC26
NPC27
NPC28
NPC29
NPC30
NPC31
J3-30
J3-29
J3-32
J3-33
J3-36
J3-35
J3-38
J3-39
LA24
LA25
LA26
LA27
LA28
LA29
LA30
LA31

FIG. 133A



PC CLOCK LOGIC
FIG. 134



INSTRUCTION PROCESSOR TIMING AND CONTROL LOGIC

FIG. 135

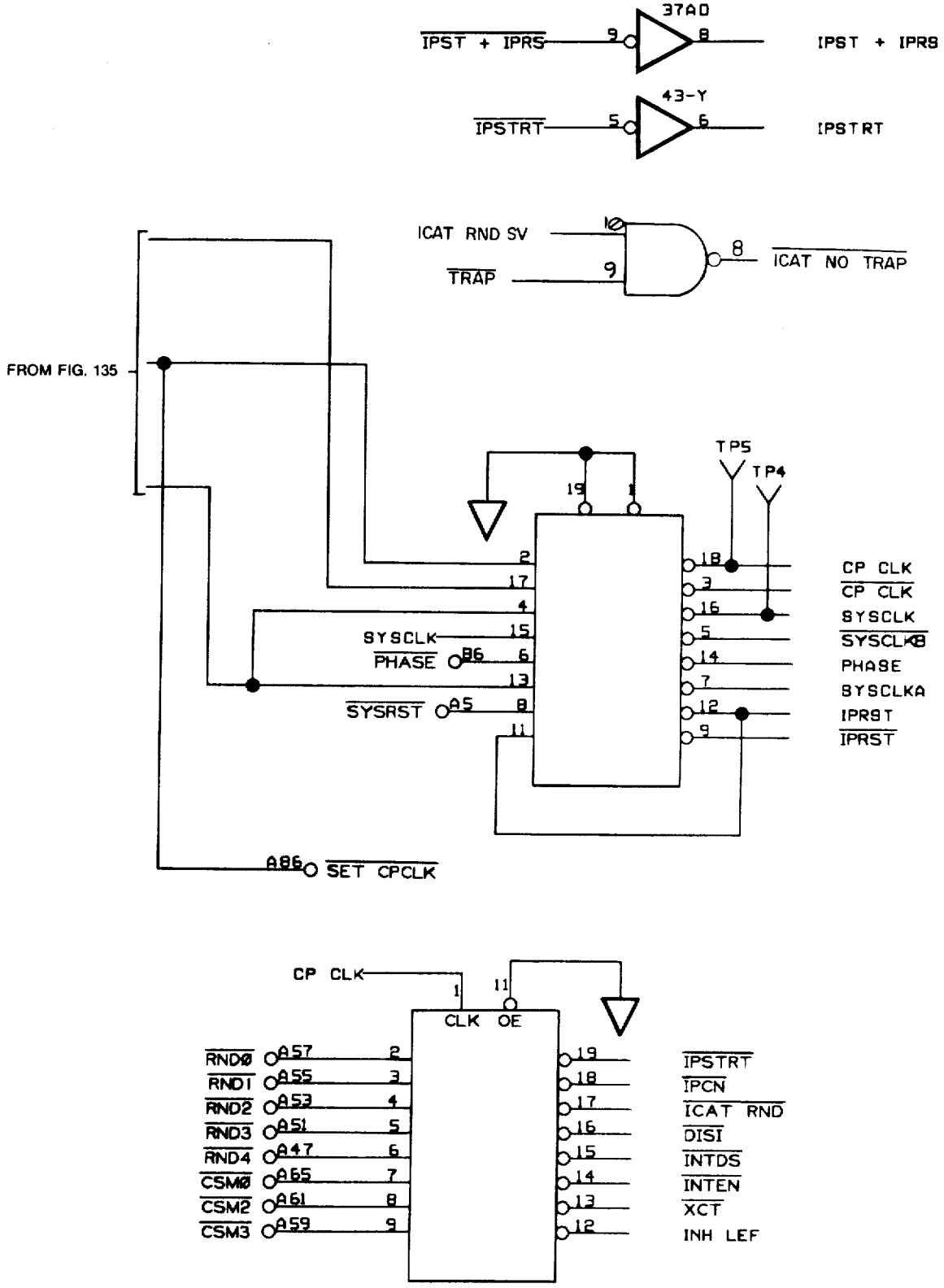

37A0
IPST + IPRS
IPST + IPRS
43-Y
IPSTRT
IPSTRT
ICAT RND SV
TRAP
ICAT NO TRAP
FROM FIG. 135
TP5
TP4
SYSCLK
PHASE
SYSRST
CP CLK
CP CLK
SYSCLK
SYSCLKB
PHASE
SYSCLKA
IPRST
IPRST
SET CPCLK
CP CLK
CLK
OE
RND0
RND1
RND2
RND3
RND4
CSM0
CSM2
CSM3
IPSTRT
IPCN
ICAT RND
DISI
INTDS
INTEN
XCT
INH LEF

FIG. 135A

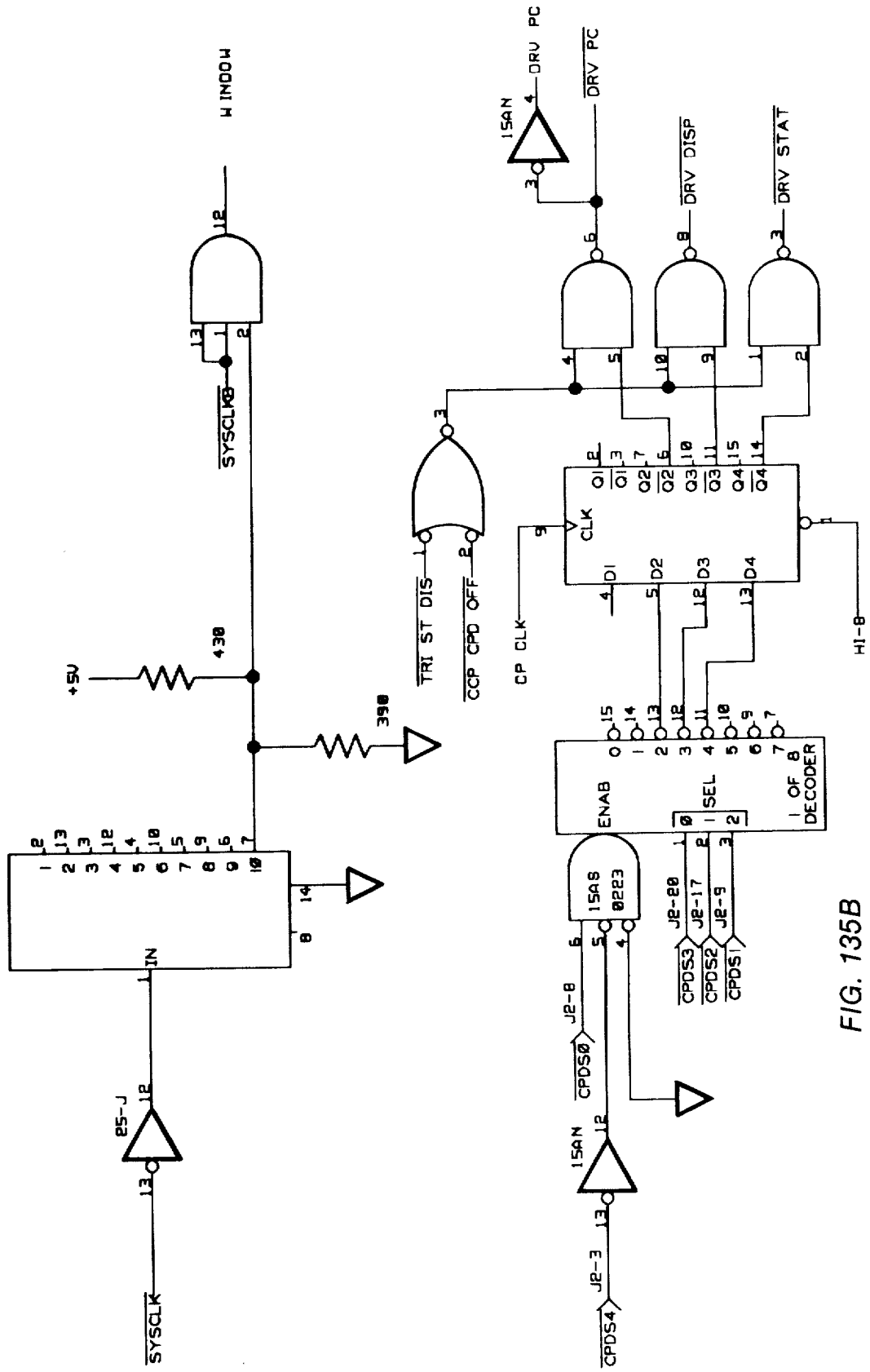

WINDOW
SYSCLK
+5V
430
390
TRI ST DIS
CCP CPD OFF
CP CLK
CLK
DRV PC
DRV DISP
DRV STAT
HI-B
ENAB
SEL
1 OF 8 DECODER
J2-20
J2-17
J2-9
CPDS3
CPDS2
CPDS1
J2-8
CPDS0
J2-3
CPDS4
85-J
SYSCLK

FIG. 135B

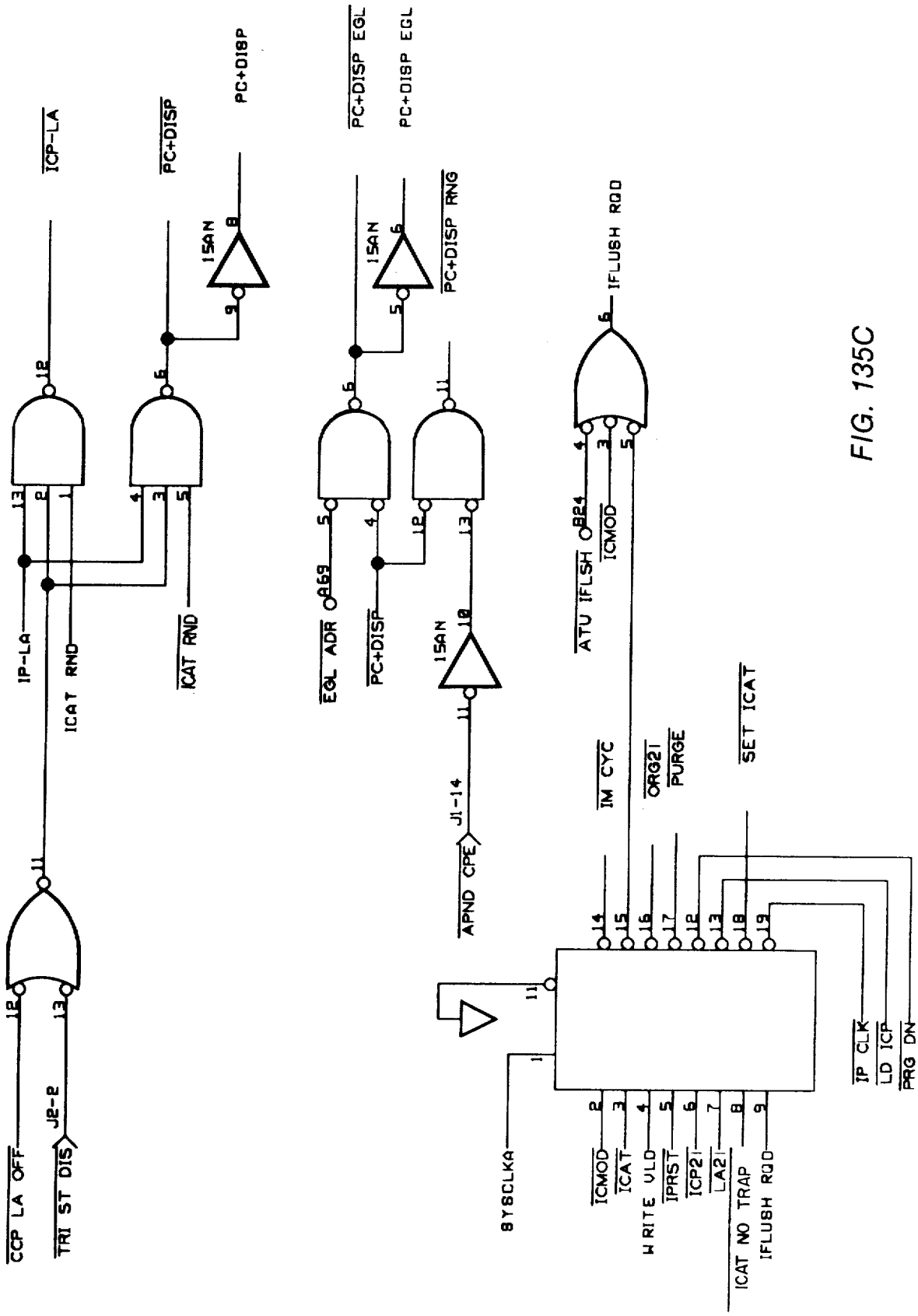
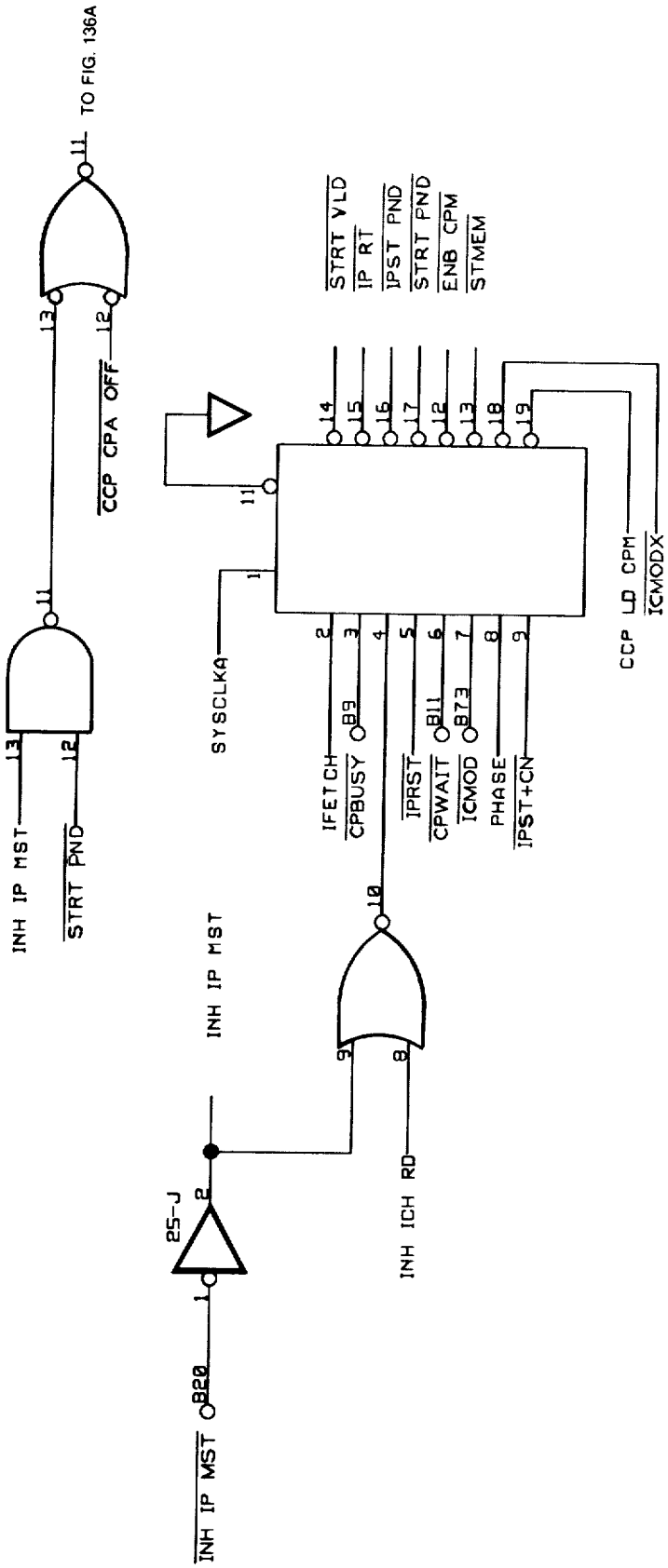

ICP-LA
PC+DISP
PC+DISP
PC+DISP EGL
PC+DISP EGL
15AN
15AN
PC+DISP RNG
IFLUSH RQD
IP-LA
ICAT RND
ICAT RND
EGL ADR
PC+DISP
15AN
ATU IFLSH
ICMOD
J1-14
APND CPE
IM CYC
ORG21
PURGE
SET ICAT
CCP LA OFF
TRI ST DIS
J2-2
BYSCLKA
ICMOD
ICAT
WRITE VLD
IPRST
ICP21
LA21
ICAT NO TRAP
IFLUSH RQD
IP CLK
LD ICP
PRG DN

FIG. 135C



SYSTEM CACHE INTERFACE LOGIC

FIG. 136

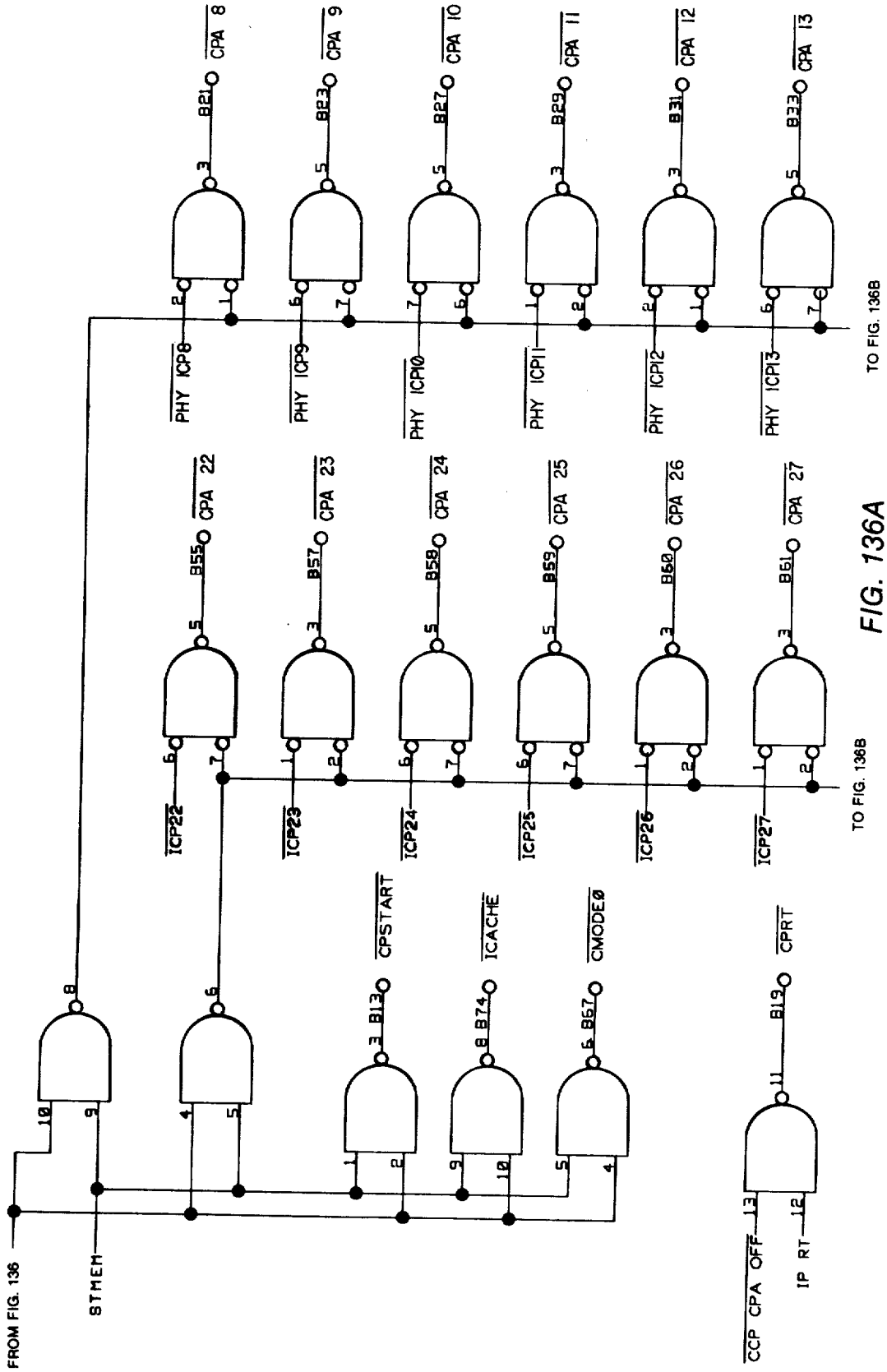

FROM FIG. 136
STMEM
CCP CPA OFF
IP RT
CPSTART
ICACHE
CMODE0
CPRT
B13
B74
B67
B19
ICP22
ICP23
ICP24
ICP25
ICP26
ICP27
B55
B57
B58
B59
B60
B61
CPA 22
CPA 23
CPA 24
CPA 25
CPA 26
CPA 27
TO FIG. 136B
PHY ICP8
PHY ICP9
PHY ICP10
PHY ICP11
PHY ICP12
PHY ICP13
B21
B23
B27
B29
B31
B33
CPA 8
CPA 9
CPA 10
CPA 11
CPA 12
CPA 13
TO FIG. 136B

FIG. 136A

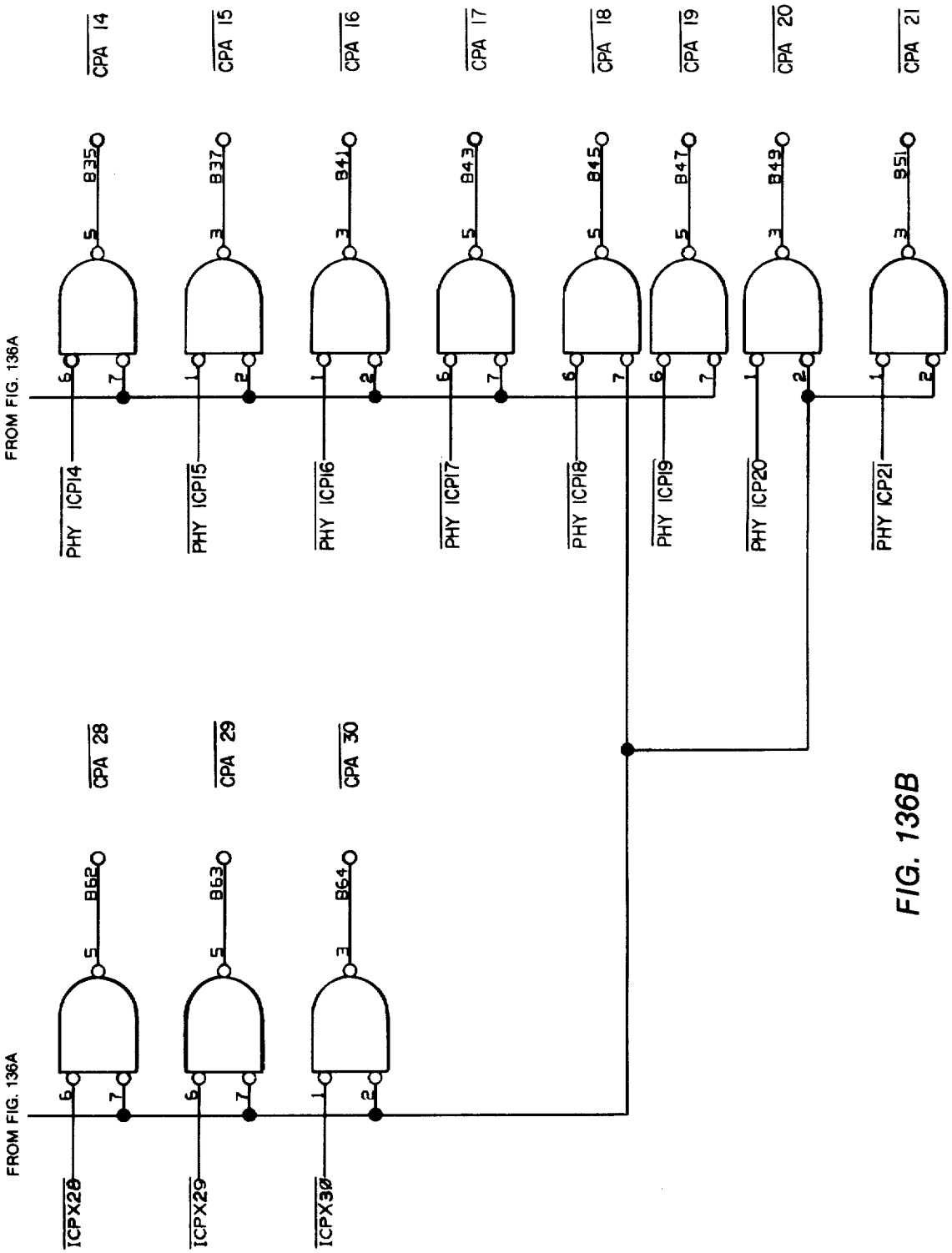

FROM FIG. 136A
PHY ICP14
PHY ICP15
PHY ICP16
PHY ICP17
PHY ICP18
PHY ICP19
PHY ICP20
PHY ICP21
B35
B37
B41
B43
B45
B47
B49
B51
CPA 14
CPA 15
CPA 16
CPA 17
CPA 18
CPA 19
CPA 20
CPA 21
FROM FIG. 136A
ICPX28
ICPX29
ICPX30
B62
B63
B64
CPA 28
CPA 29
CPA 30

FIG. 136B

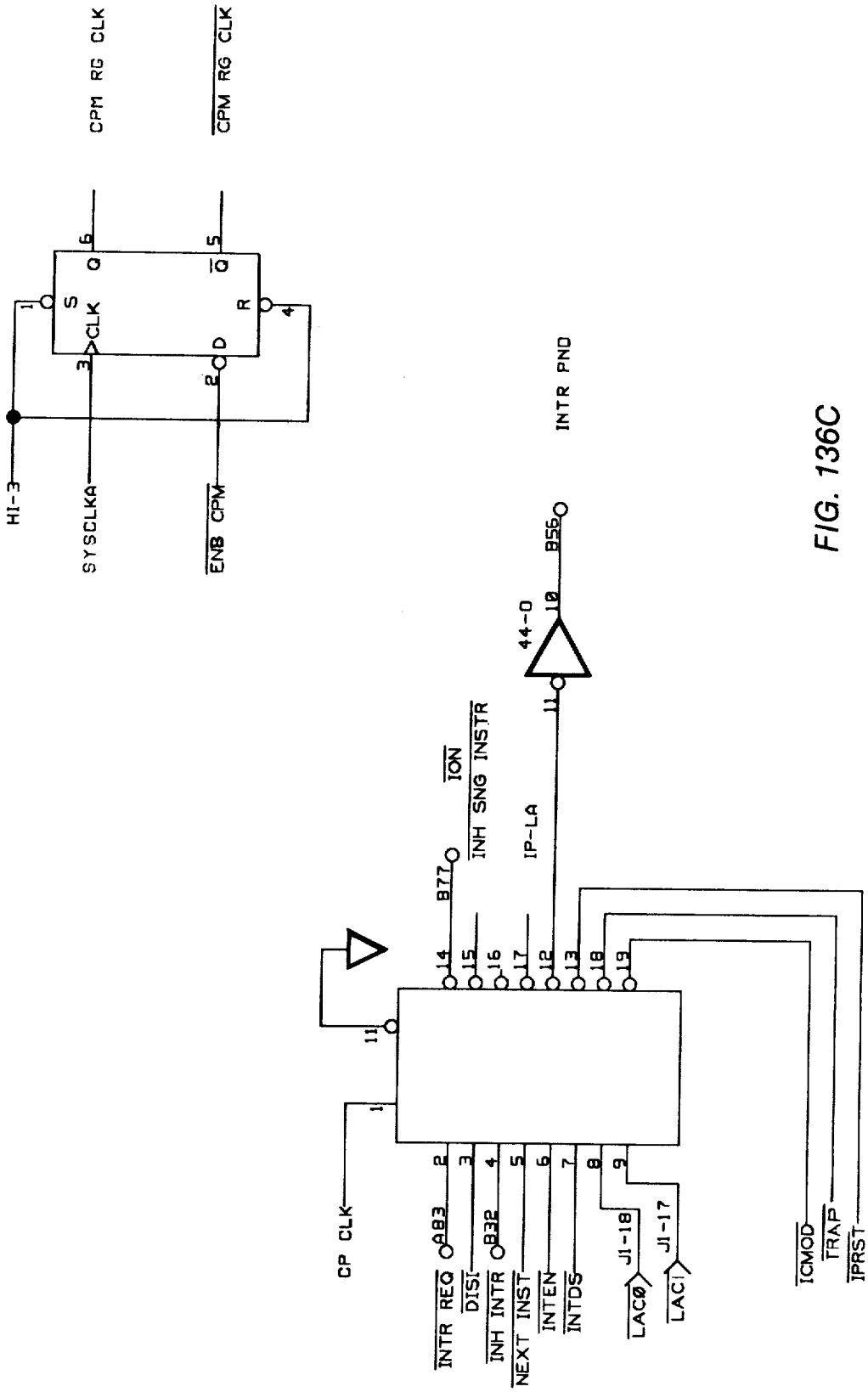

CPM RG CLK
CPM RG CLK
HI-3
SYSCLKA
ENB CPM
S
CLK
D
Q
Q
R
INTR PND
44-D
B56
ION
INH SNG INSTR
IP-LA
B77
CP CLK
INTR REQ
DISI
INH INTR
NEXT INST
INTEN
INTDS
LAC0
LAC1
J1-18
J1-17
B83
B32
ICMOD
TRAP
IPRST

FIG. 136C

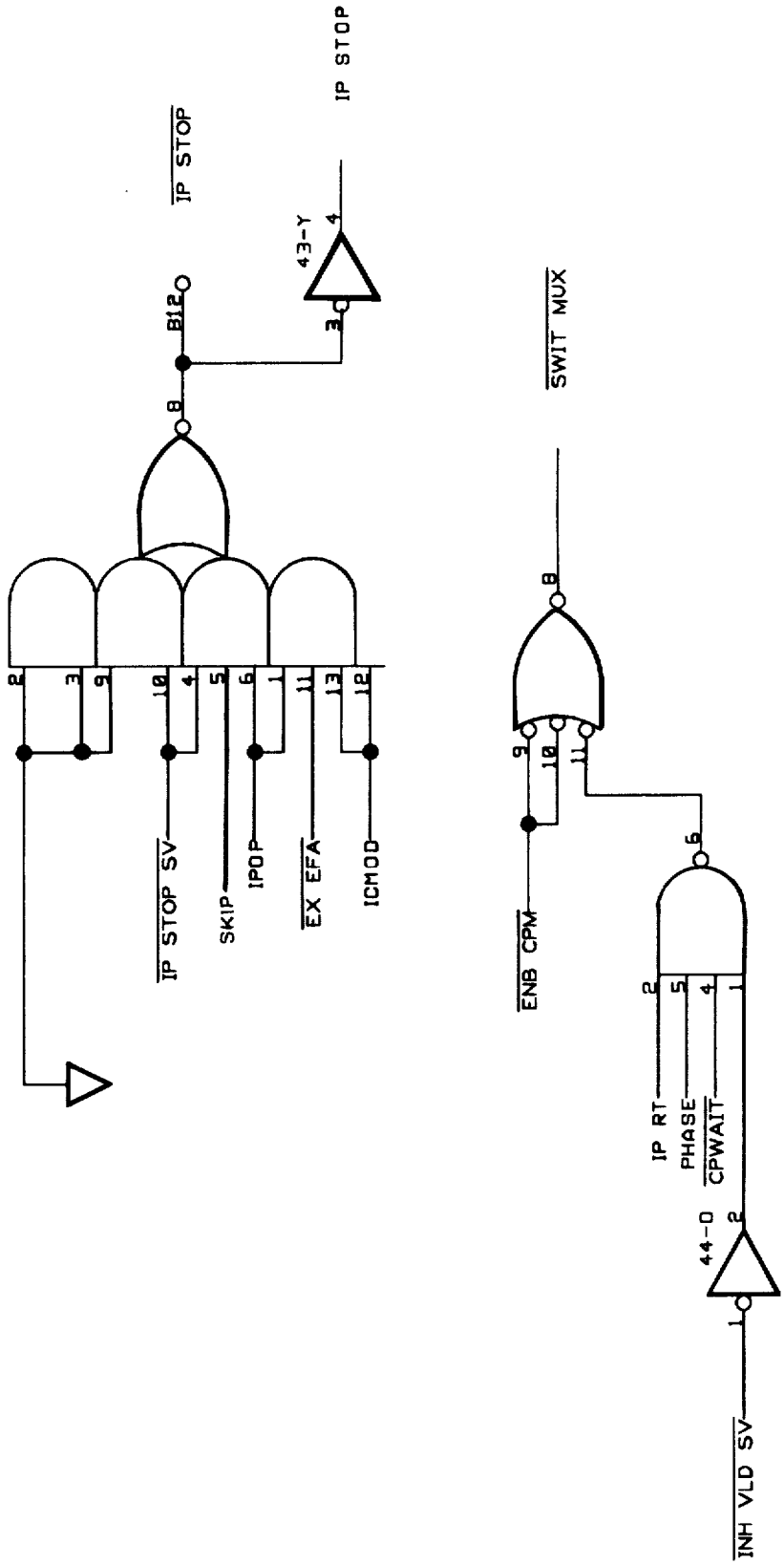
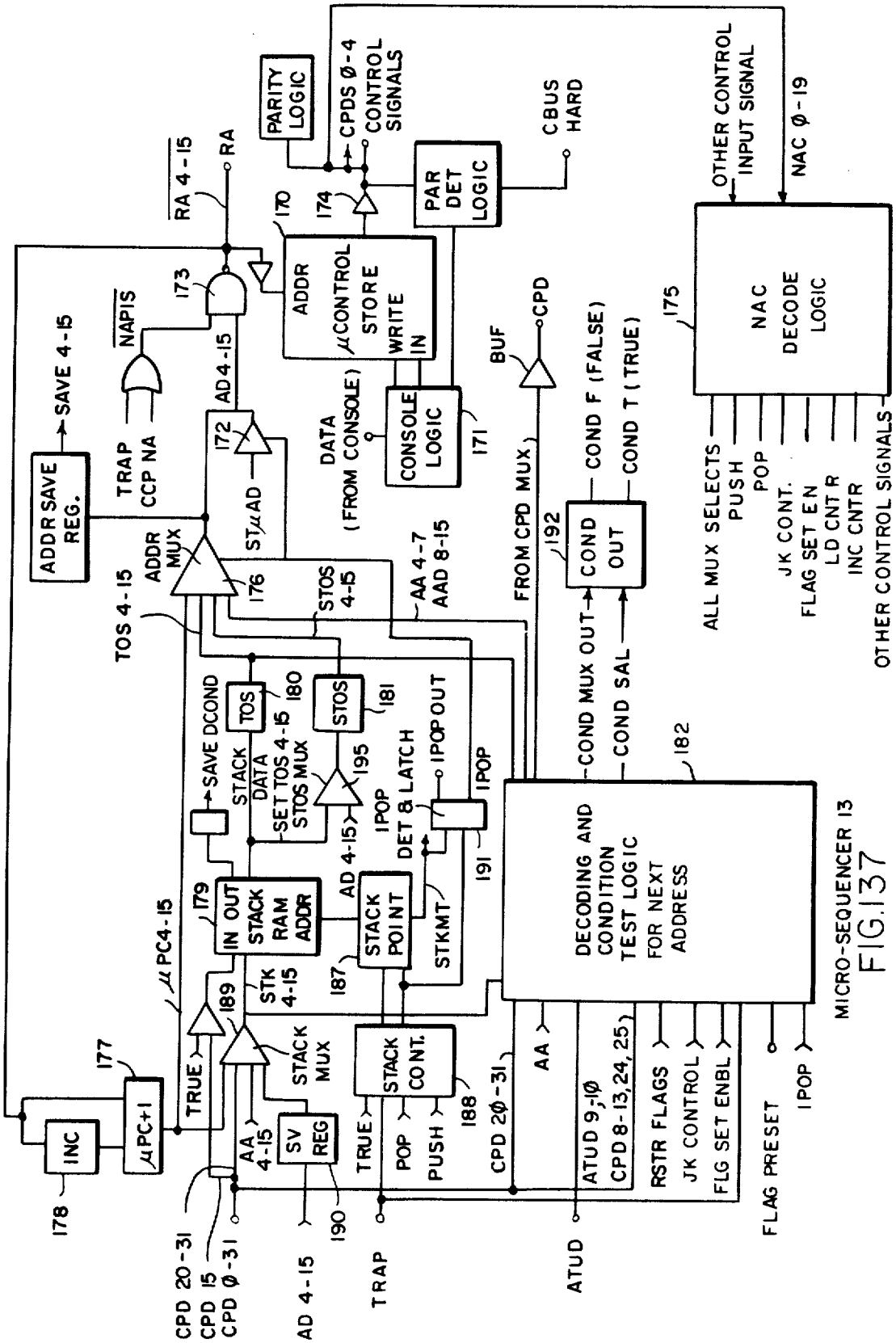

IP STOP
IP STOP
43-Y
B12
SWIT MUX
IP STOP SV
SKIP
IPOP
EX EFA
ICMOD
ENB CPM
IP RT
PHASE
CPWAIT
44-D
INH VLD SV

FIG. 136D


PARITY LOGIC
CPDS Ø-4 CONTROL SIGNALS
CBUS HARD
OTHER CONTROL INPUT SIGNAL
NAC Ø-19
RA 4-15
RA
PAR DET LOGIC
170
174
ADDR
μCONTROL STORE WRITE IN
173
NAPIS
AD 4-15
SAVE 4-15
BUF
CPD
175
NAC DECODE LOGIC
CONSOLE LOGIC
171
DATA (FROM CONSOLE)
COND F (FALSE)
COND T (TRUE)
172
TRAP
CCP NA
ADDR SAVE REG.
STμAD
ADDR MUX
176
STOS 4-15
AA 4-7
AAD 8-15
FROM CPD MUX
192
COND OUT
ALL MUX SELECTS
PUSH
POP
JK CONT.
FLAG SET EN
LD CNTR
INC CNTR
OTHER CONTROL SIGNALS
TOS 4-15
SAVE DCOND
TOS
180
STOS
181
STACK DATA
SET TOS 4-15
STOS MUX
195
IPOP OUT
IPOP
DET & LATCH
COND MUX OUT
COND SAL
182
DECODING AND CONDITION TEST LOGIC FOR NEXT ADDRESS
MICRO-SEQUENCER 13
FIG.137
μPC4-15
179
IN OUT
STACK RAM ADDR
AD 4-15
STACK POINT
STKMT
191
STK 4-15
187
189
STACK MUX
177
TRUE
STACK CONT.
188
AA
CPD 2Ø-31
ATUD 9-1Ø
CPD 8-13,24,25
RSTR FLAGS
JK CONTROL
FLG SET ENBL
FLAG PRESET
IPOP
INC
178
μPC+1
AA 4-15
SV REG
190
TRUE
POP
PUSH
CPD 20-31
CPD 15
CPD Ø-31
AD 4-15
TRAP
ATUD

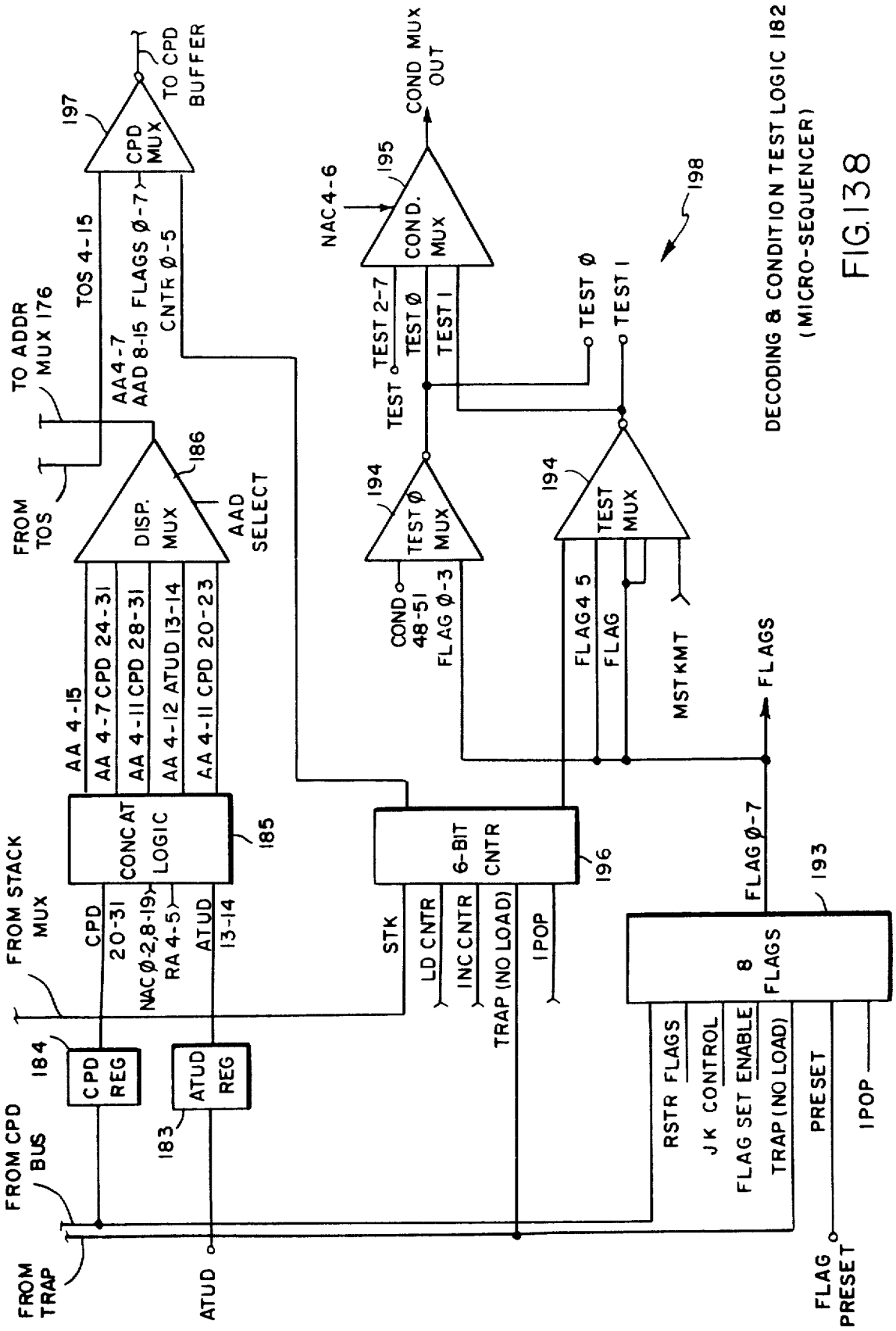
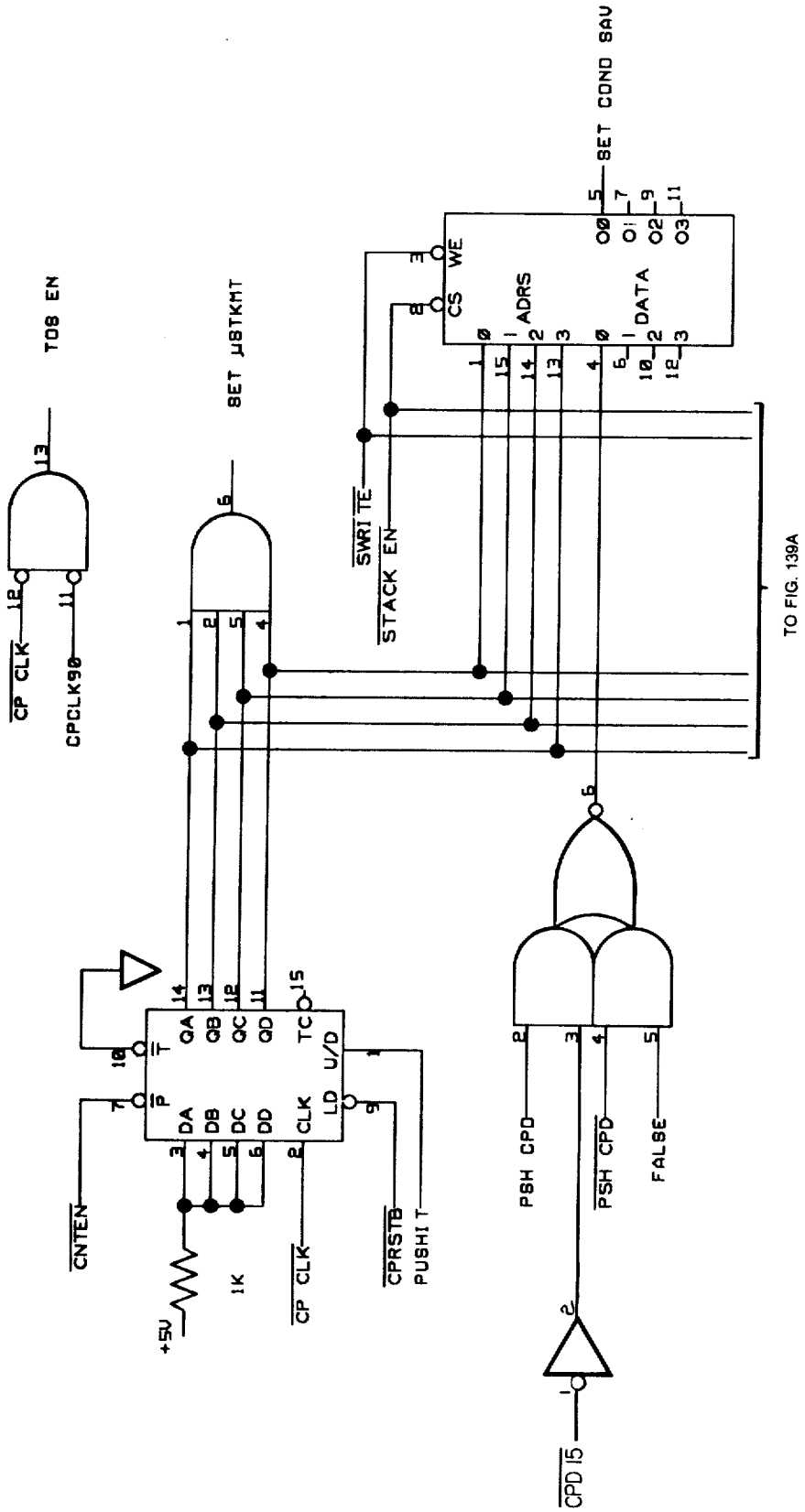

DECODING & CONDITION TEST LOGIC 182
(MICRO-SEQUENCER)
198
COND MUX OUT
195
COND. MUX
NAC 4-6
TEST 2-7
TEST Ø
TEST 1
TEST
TEST Ø
TEST 1
194
TEST Ø MUX
COND 48-51
FLAG Ø-3
194
TEST MUX
FLAG 4 5
FLAG
MST KMT
FLAGS
FLAG Ø-7
193
8 FLAGS
RSTR FLAGS
JK CONTROL
FLAG SET ENABLE
TRAP (NO LOAD)
PRESET
IPOP
FLAG PRESET
196
6-BIT CNTR
STK
LD CNTR
INC CNTR
TRAP (NO LOAD)
IPOP
185
CONCAT LOGIC
AA 4-15
AA 4-7 CPD 24-31
AA 4-11 CPD 28-31
AA 4-12 ATUD 13-14
AA 4-11 CPD 20-23
CPD 20-31
NAC Ø-2,8-19
RA 4-5
ATUD 13-14
FROM STACK MUX
184
CPD REG
183
ATUD REG
FROM CPD BUS
FROM TRAP
ATUD
186
DISP. MUX
AAD SELECT
FROM TOS
TO ADDR MUX 176
TOS 4-15
AA4-7
AAD 8-15
FLAGS Ø-7
CNTR Ø-5
197
CPD MUX
TO CPD BUFFER

FIG. 138



STACK MUX, STACK RAM, STACK POINTER AND TDS U

FIG. 139

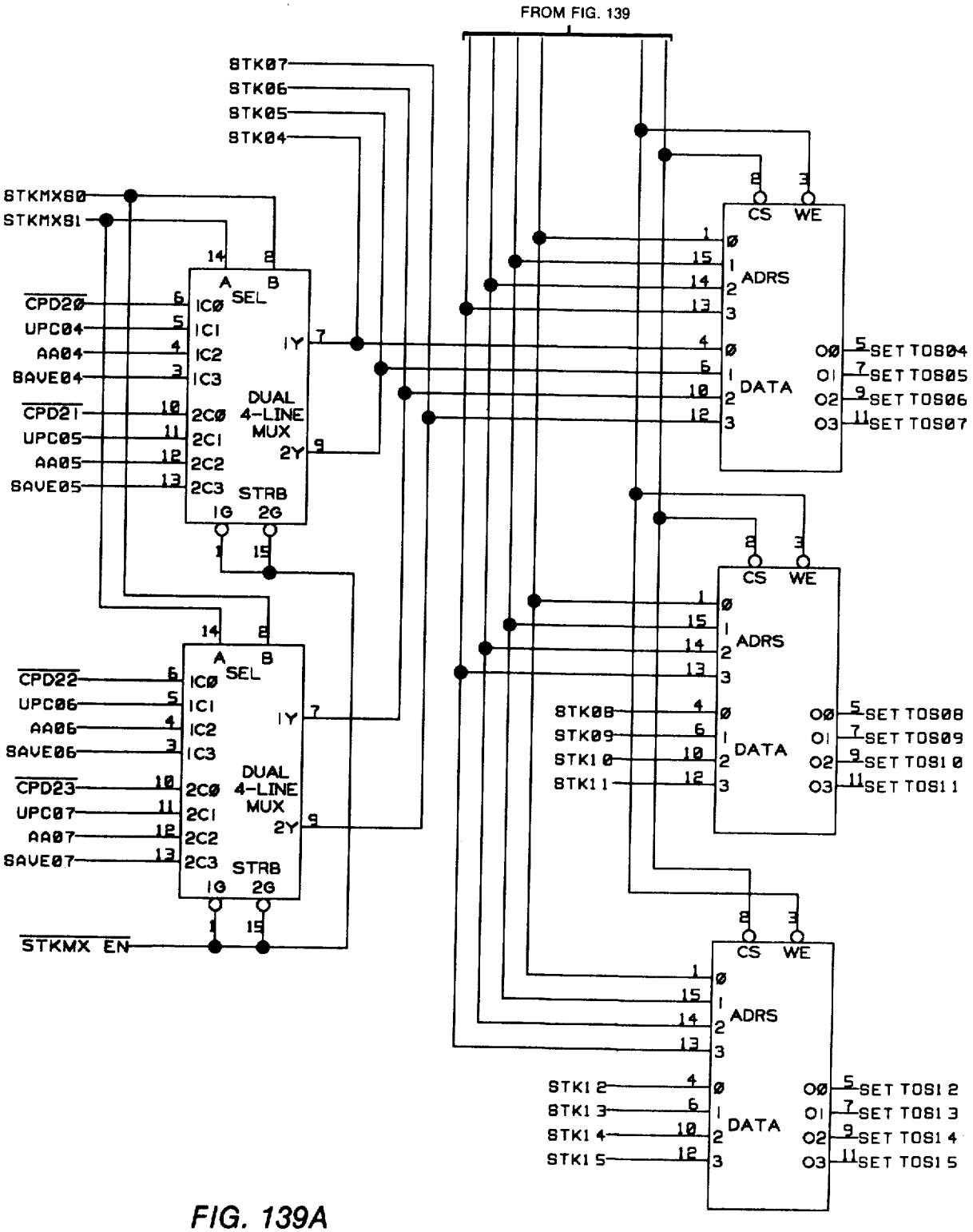

FROM FIG. 139
STK07
STK06
STK05
STK04
STKMXS0
STKMXS1
CPD20
UPC04
AA04
SAVE04
CPD21
UPC05
AA05
SAVE05
CPD22
UPC06
AA06
SAVE06
CPD23
UPC07
AA07
SAVE07
STKMX EN
DUAL 4-LINE MUX
SEL
STRB
CS
WE
ADRS
DATA
SET TOS04
SET TOS05
SET TOS06
SET TOS07
STK08
STK09
STK10
STK11
SET TOS08
SET TOS09
SET TOS10
SET TOS11
STK12
STK13
STK14
STK15
SET TOS12
SET TOS13
SET TOS14
SET TOS15

FIG. 139A

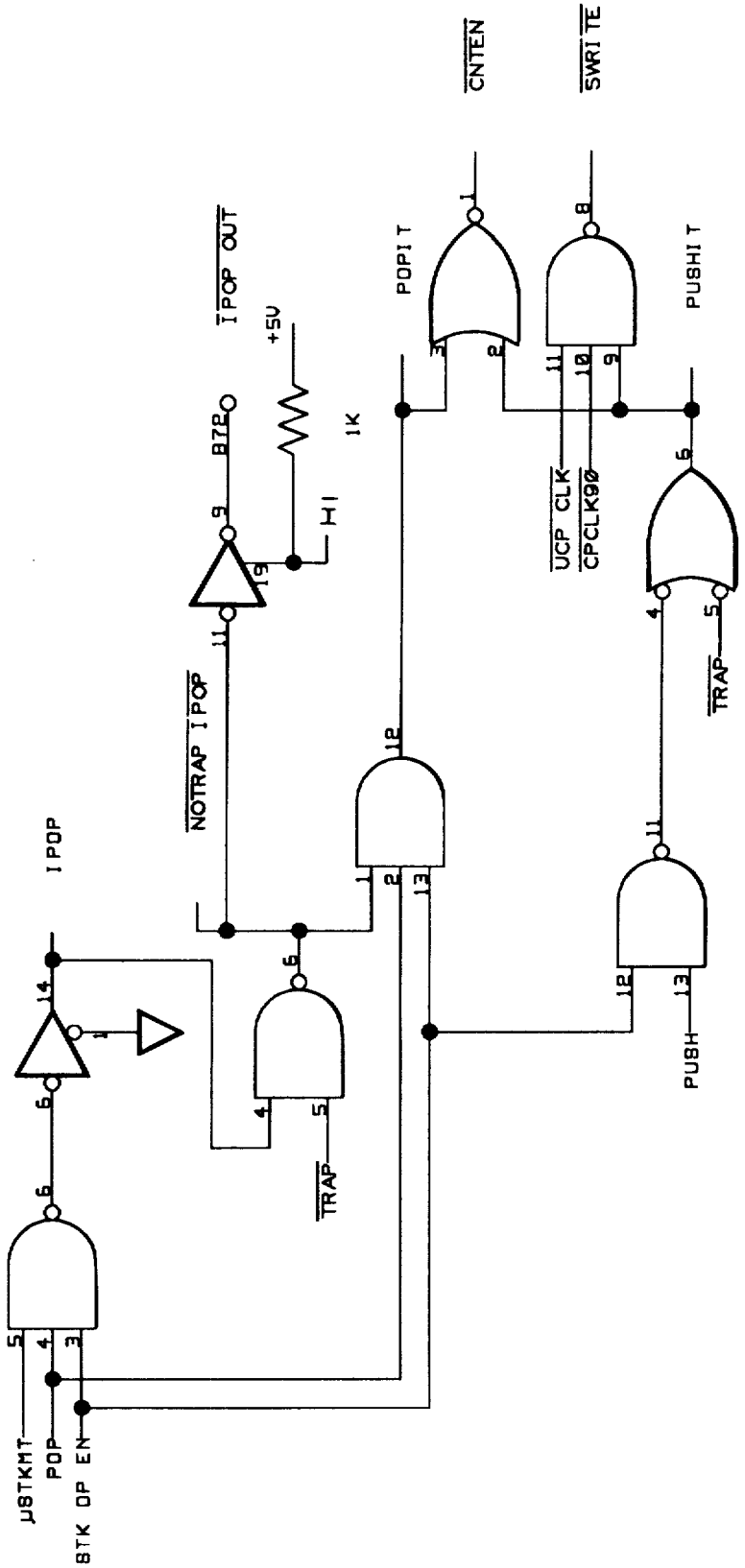

CNTEN
SWRITE
POPIT
PUSHIT
IPOP OUT
+5V
1K
BZB
HI
UCP CLK
CPCLK90
NOTRAP IPOP
IPOP
TRAP
PUSH
μSTKMT
POP
STK OP EN

FIG. 139B

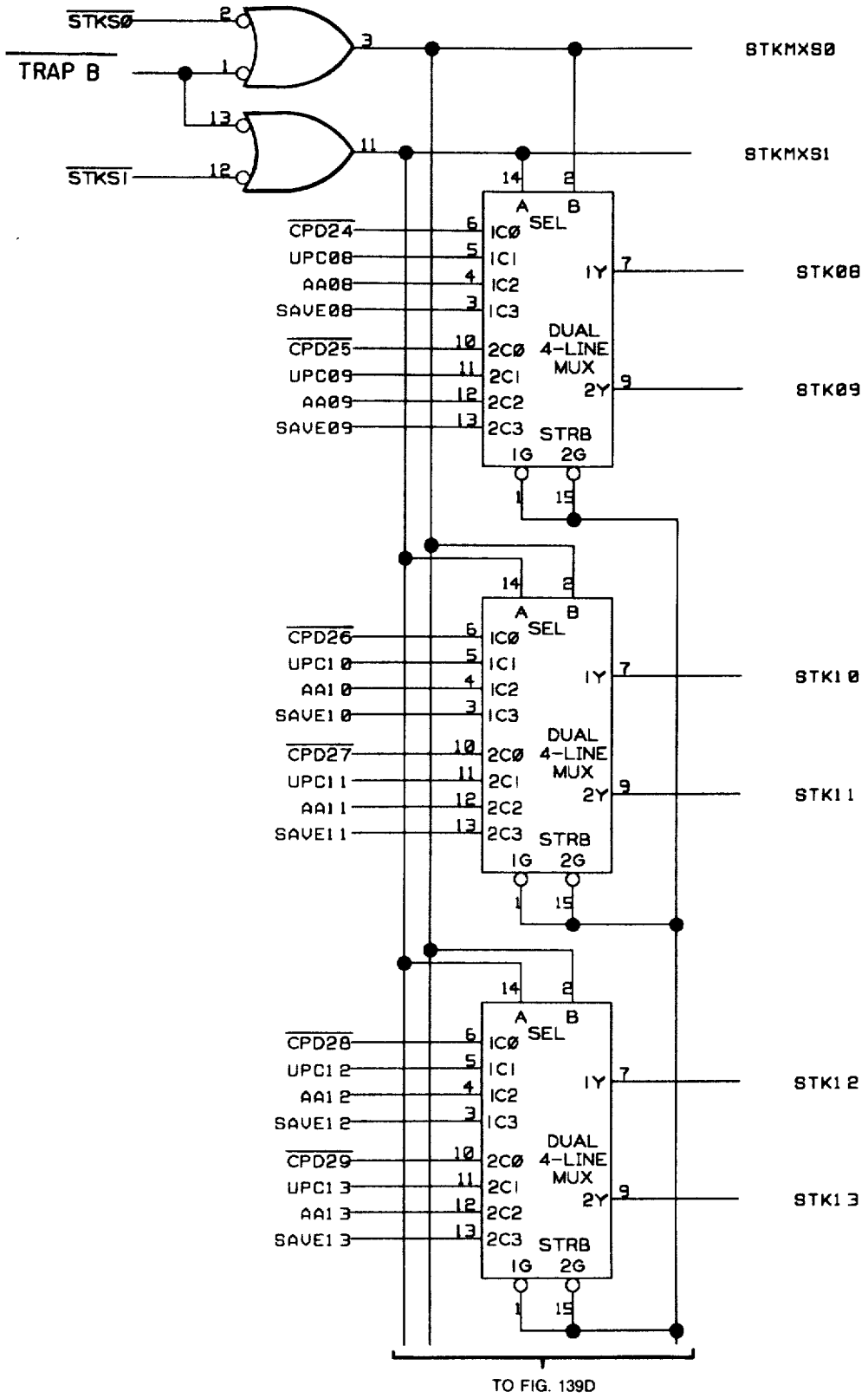

STKS0
TRAP B
STKS1
STKMXS0
STKMXS1
CPD24
UPC08
AA08
SAVE08
CPD25
UPC09
AA09
SAVE09
STK08
STK09
CPD26
UPC10
AA10
SAVE10
CPD27
UPC11
AA11
SAVE11
STK10
STK11
CPD28
UPC12
AA12
SAVE12
CPD29
UPC13
AA13
SAVE13
STK12
STK13
A
B
SEL
1C0
1C1
1C2
1C3
1Y
DUAL
4-LINE
MUX
2C0
2C1
2C2
2C3
2Y
STRB
1G
2G
TO FIG. 139D

FIG. 139C

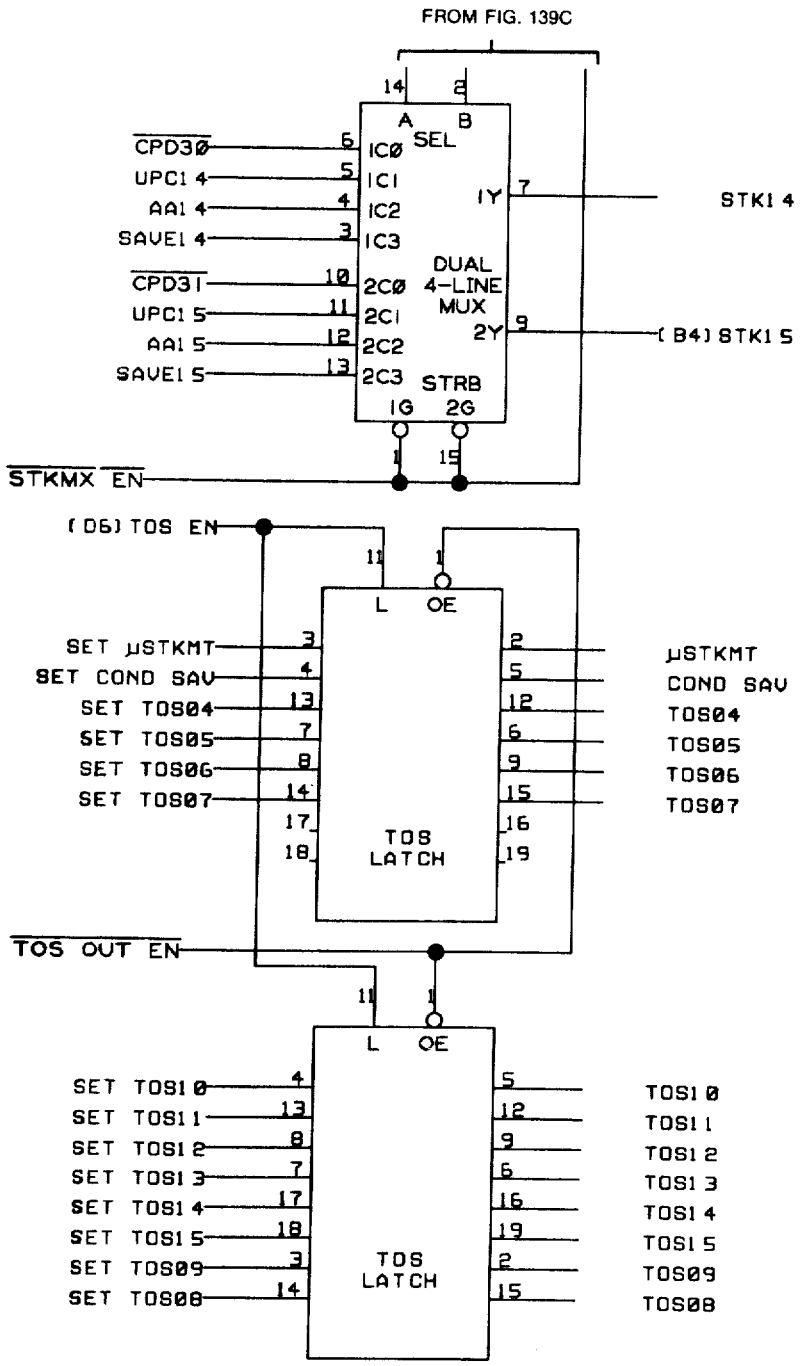
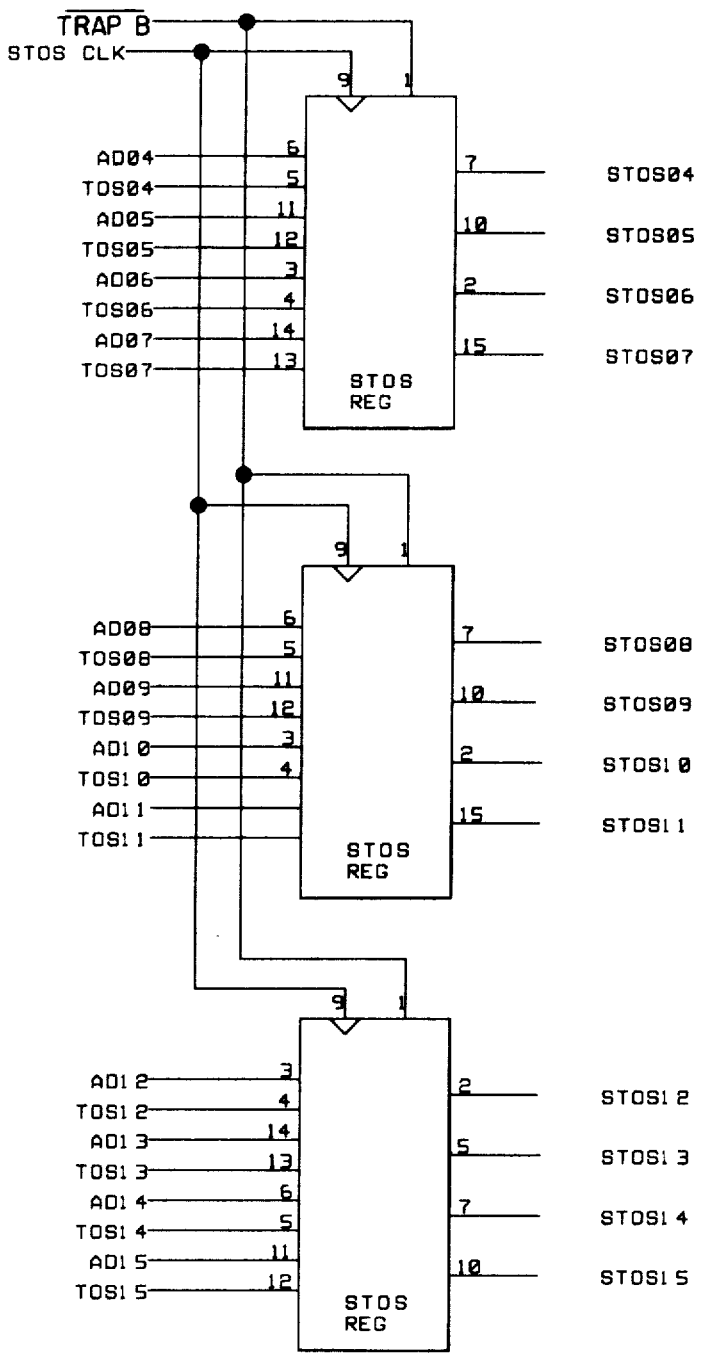
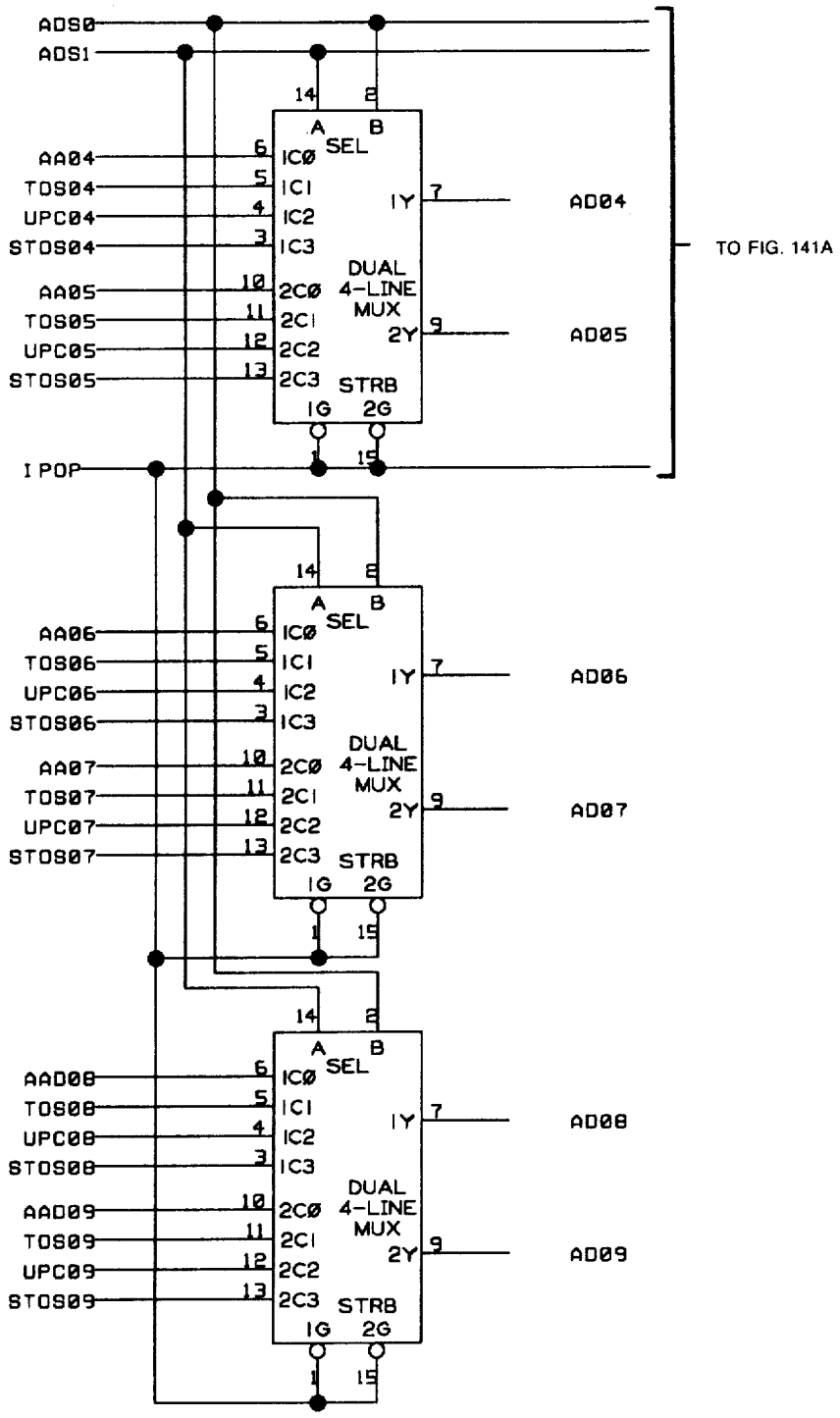

FROM FIG. 139C
A
B
SEL
CPD3Ø
UPC1 4
AA1 4
SAVE1 4
CPD31
UPC1 5
AA1 5
SAVE1 5
1CØ
1C1
1C2
1C3
2CØ
2C1
2C2
2C3
1Y
2Y
DUAL
4-LINE
MUX
STRB
1G
2G
STK1 4
(B4) STK1 5
STKMX EN
(D6) TOS EN
L
OE
SET μSTKMT
SET COND SAV
SET TOSØ4
SET TOSØ5
SET TOSØ6
SET TOSØ7
TOS
LATCH
μSTKMT
COND SAV
TOSØ4
TOSØ5
TOSØ6
TOSØ7
TOS OUT EN
SET TOS1 Ø
SET TOS1 1
SET TOS1 2
SET TOS1 3
SET TOS1 4
SET TOS1 5
SET TOSØ9
SET TOSØ8
TOS1 Ø
TOS1 1
TOS1 2
TOS1 3
TOS1 4
TOS1 5
TOSØ9
TOSØ8

FIG. 139D



STOS UNIT 181

FIG. 140



ADDRESS MUX 176

FIG. 141

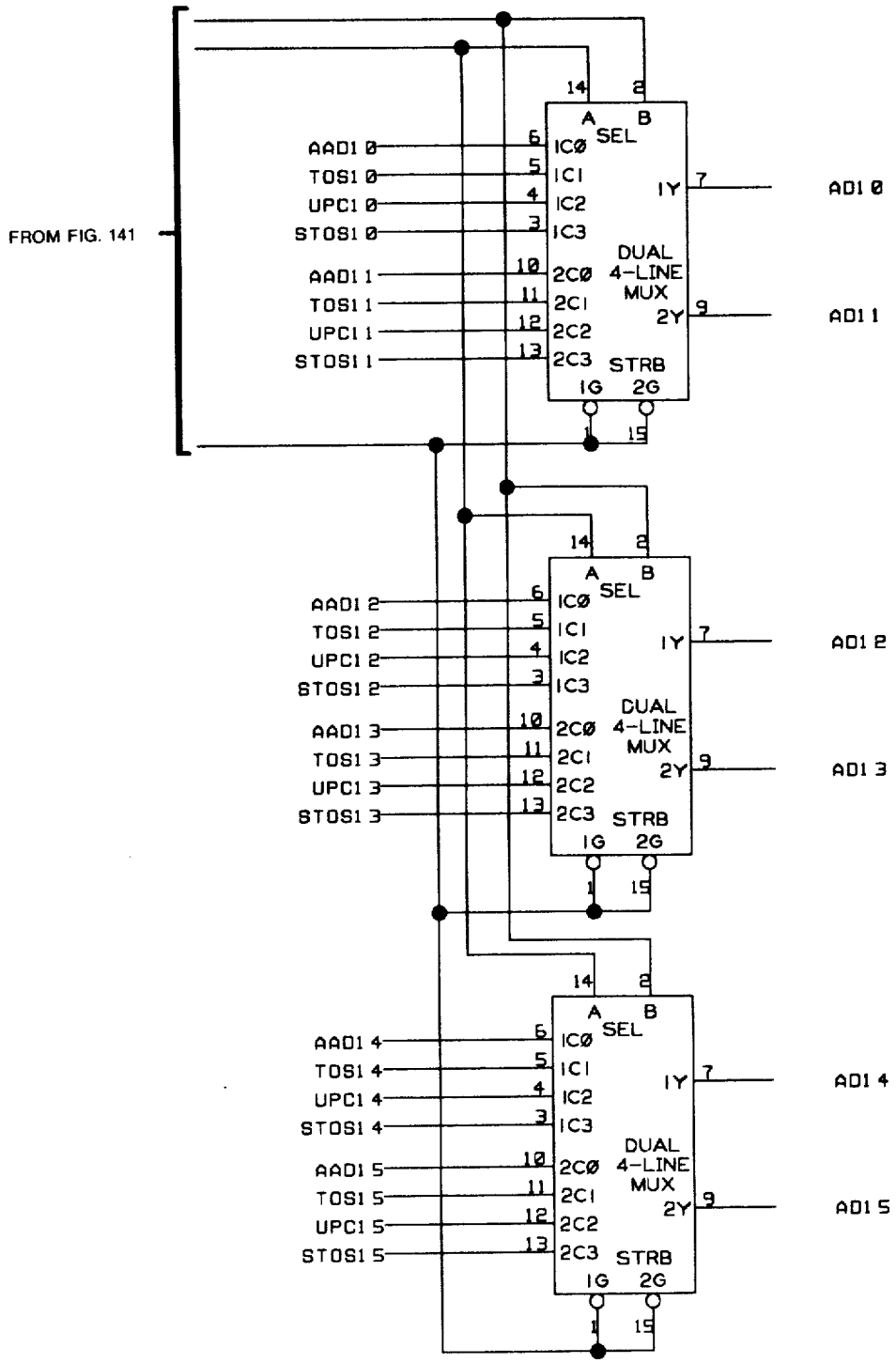

FROM FIG. 141
AAD1 0
TOS1 0
UPC1 0
STOS1 0
AAD1 1
TOS1 1
UPC1 1
STOS1 1
14
2
A
B
SEL
6
1C0
5
1C1
4
1C2
3
1C3
1Y
7
AD1 0
DUAL
4-LINE
MUX
10
2C0
11
2C1
12
2C2
13
2C3
2Y
9
AD1 1
STRB
1G
2G
1
15
AAD1 2
TOS1 2
UPC1 2
STOS1 2
AAD1 3
TOS1 3
UPC1 3
STOS1 3
AD1 2
AD1 3
AAD1 4
TOS1 4
UPC1 4
STOS1 4
AAD1 5
TOS1 5
UPC1 5
STOS1 5
AD1 4
AD1 5

FIG. 141A

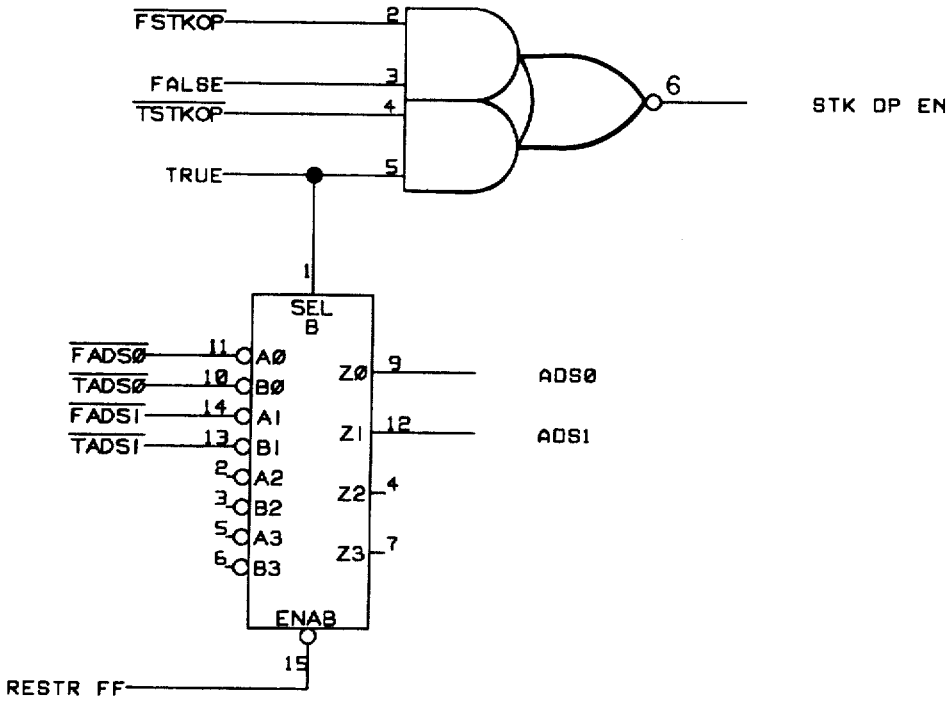
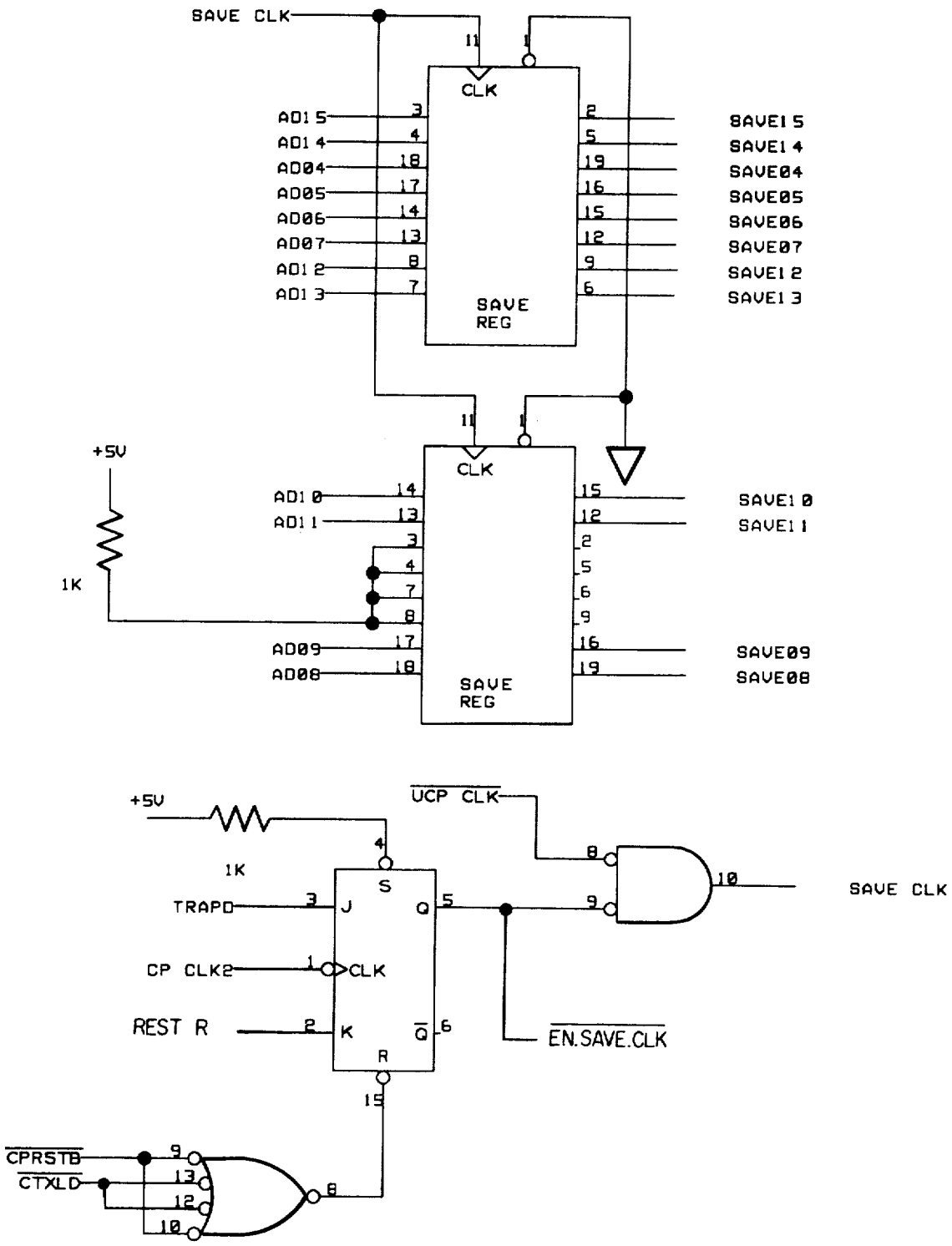
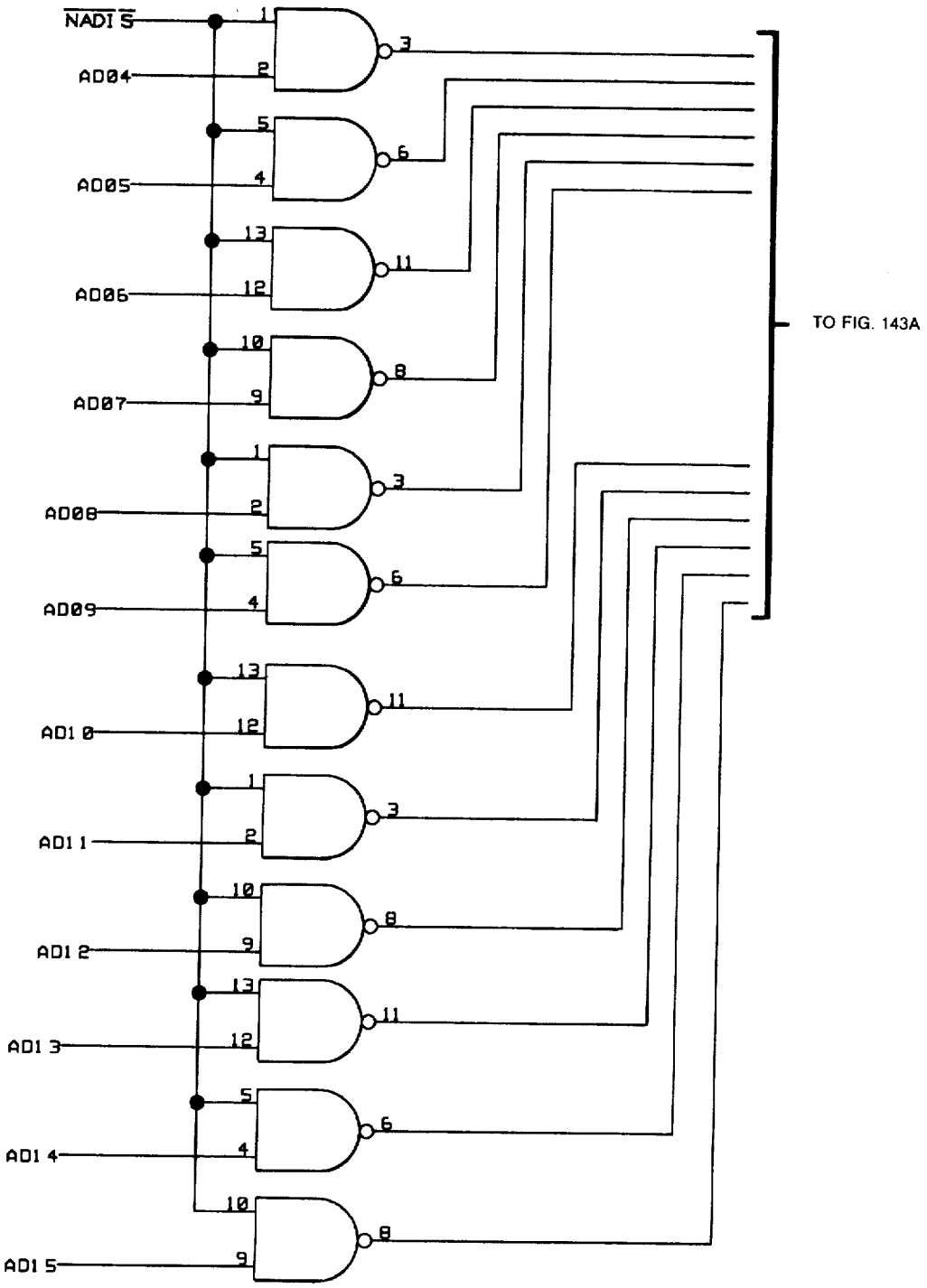

FSTKOP
FALSE
TSTKOP
TRUE
STK DP EN
SEL
B
FADSØ
TADSØ
FADSI
TADSI
AØ
BØ
AI
BI
A2
B2
A3
B3
ZØ
ZI
Z2
Z3
ADSØ
ADSI
ENAB
RESTR FF

FIG. 141B



ADDR SAVE REGISTER

FIG. 142



ADDRESS LOGIC TO μCONTROL STORE

FIG. 143

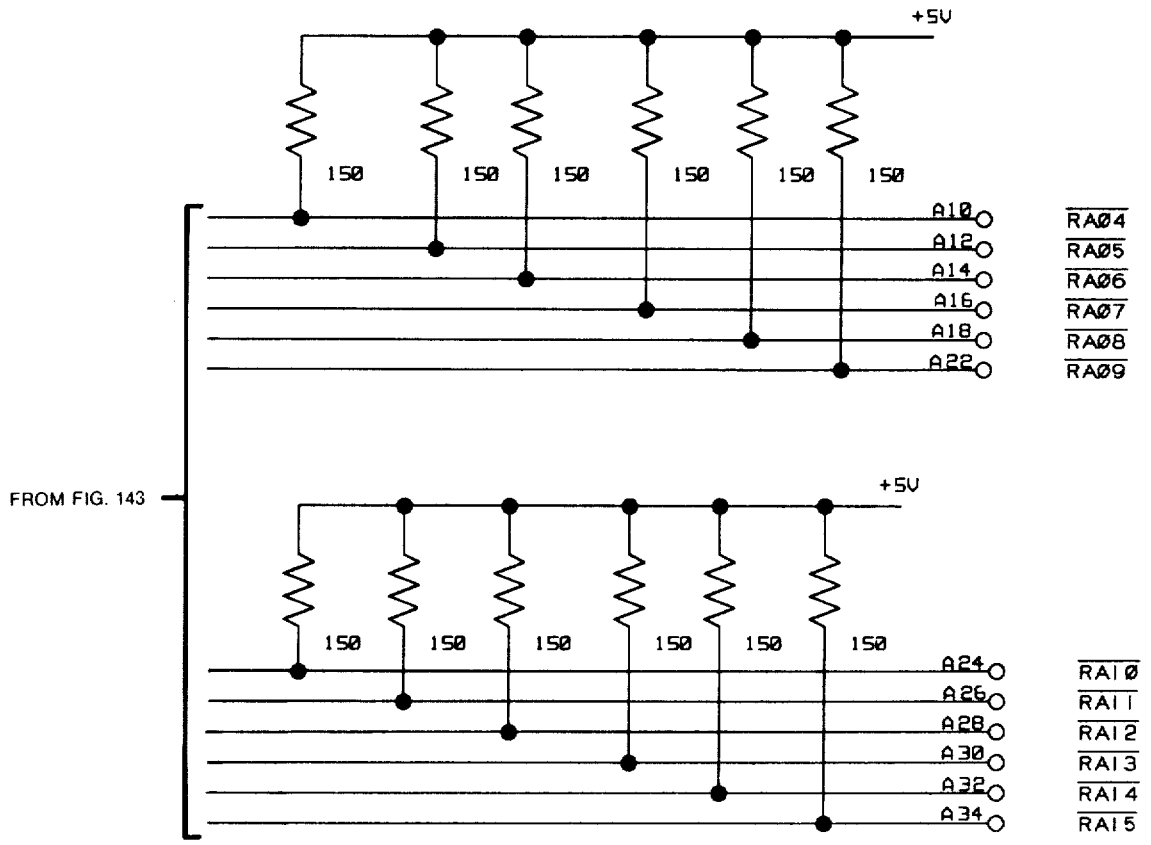
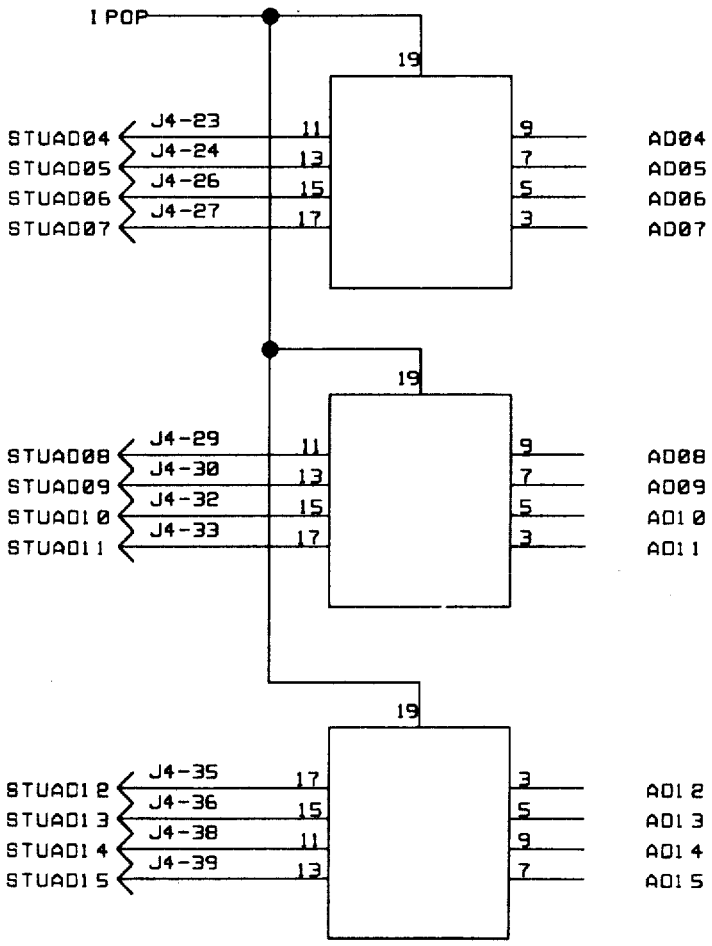
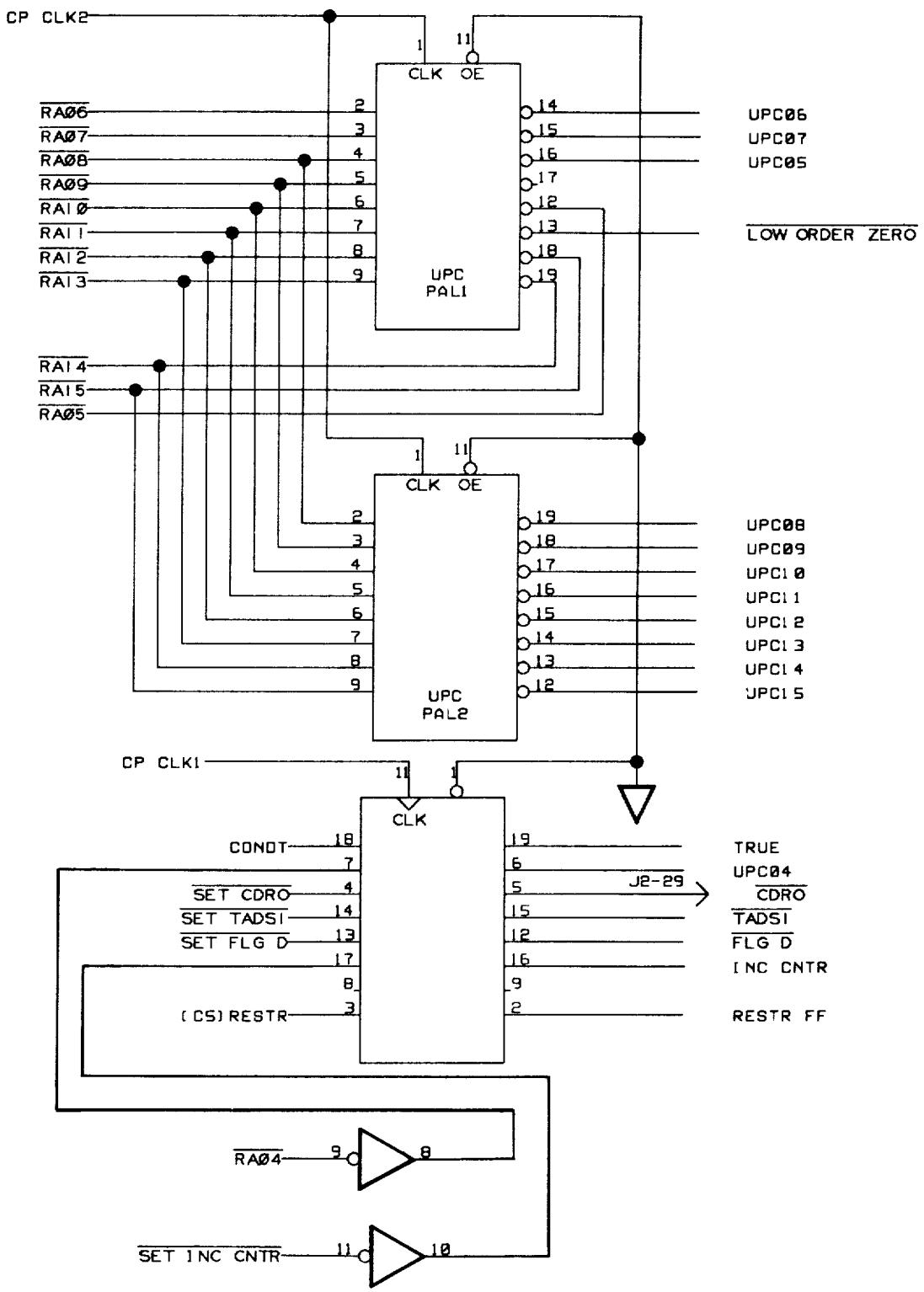

+5V
150
150
150
150
150
150
A10
A12
A14
A16
A18
A22
RA04
RA05
RA06
RA07
RA08
RA09
FROM FIG. 143
+5V
150
150
150
150
150
150
A24
A26
A28
A30
A32
A34
RAI0
RAI1
RAI2
RAI3
RAI4
RAI5

FIG. 143A



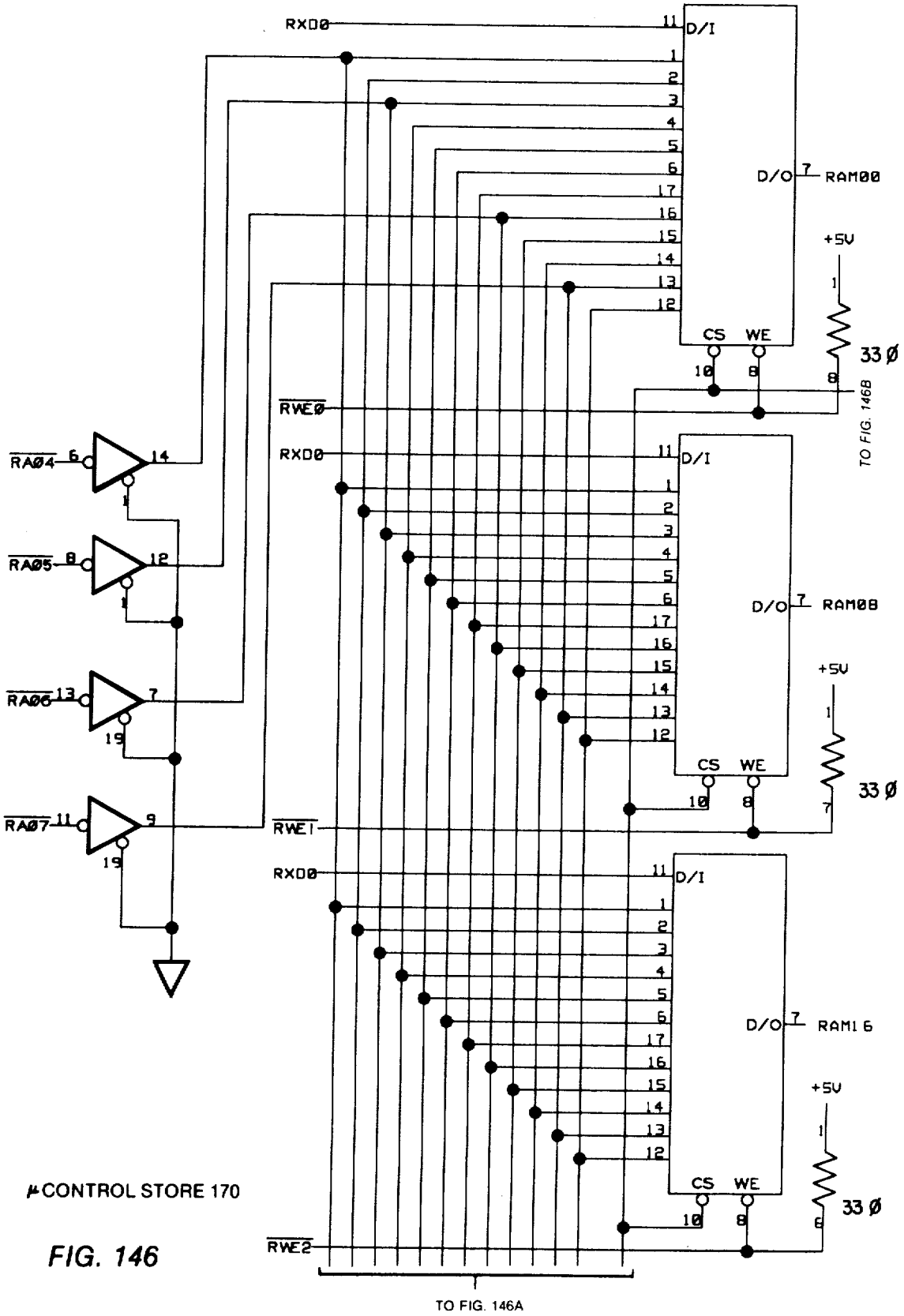
ST μAD DRIVER

FIG. 144



(MPC + 1) UNIT 177 AND INCREMENT UNIT 178

FIG. 145



μCONTROL STORE 170

FIG. 146

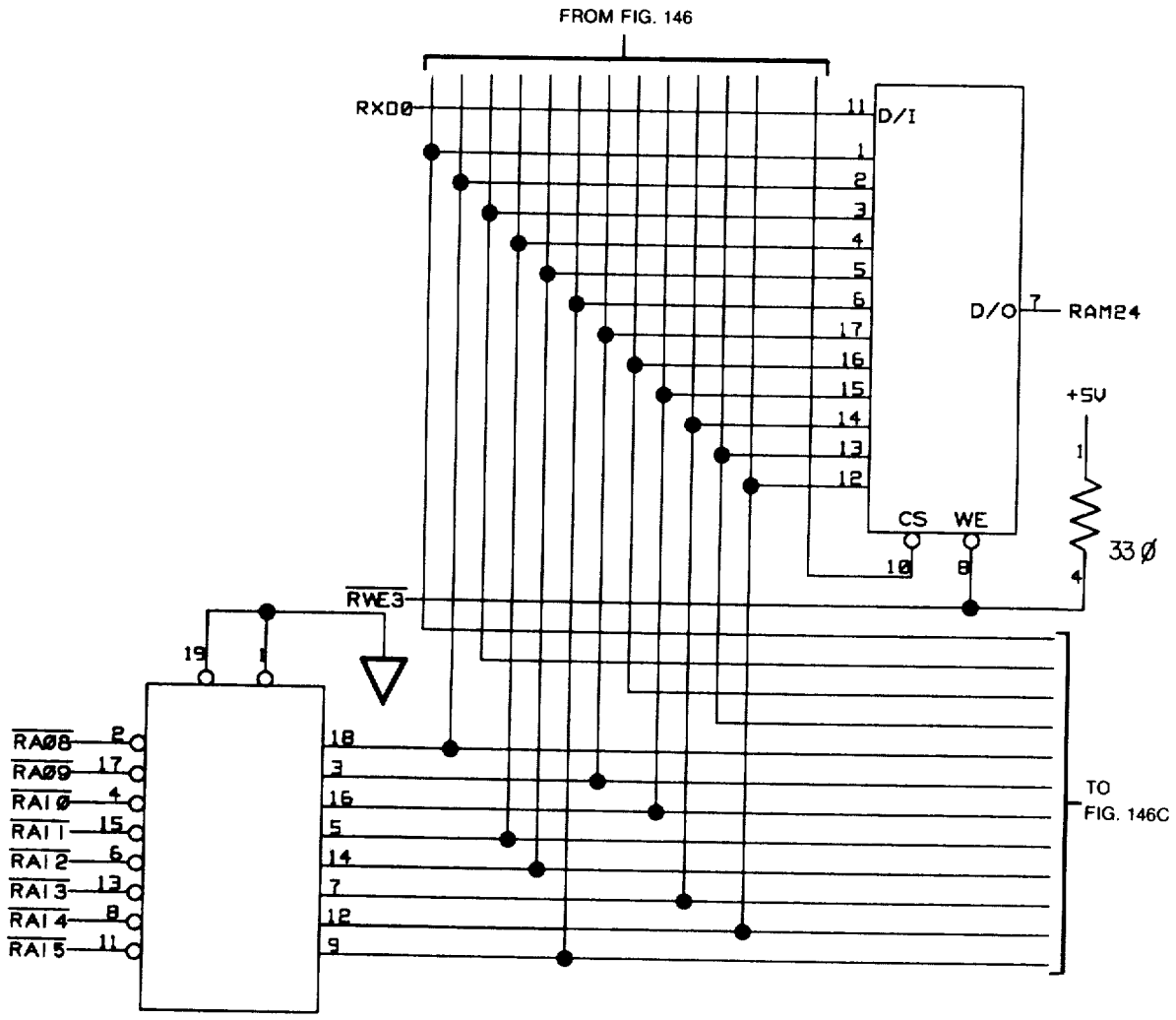

FROM FIG. 146
RX00
D/I
D/O
RAM24
+5V
33 Ø
CS
WE
RWE3
RA08
RA09
RAI0
RAI1
RAI2
RAI3
RAI4
RAI5
TO
FIG. 146C

FIG. 146A

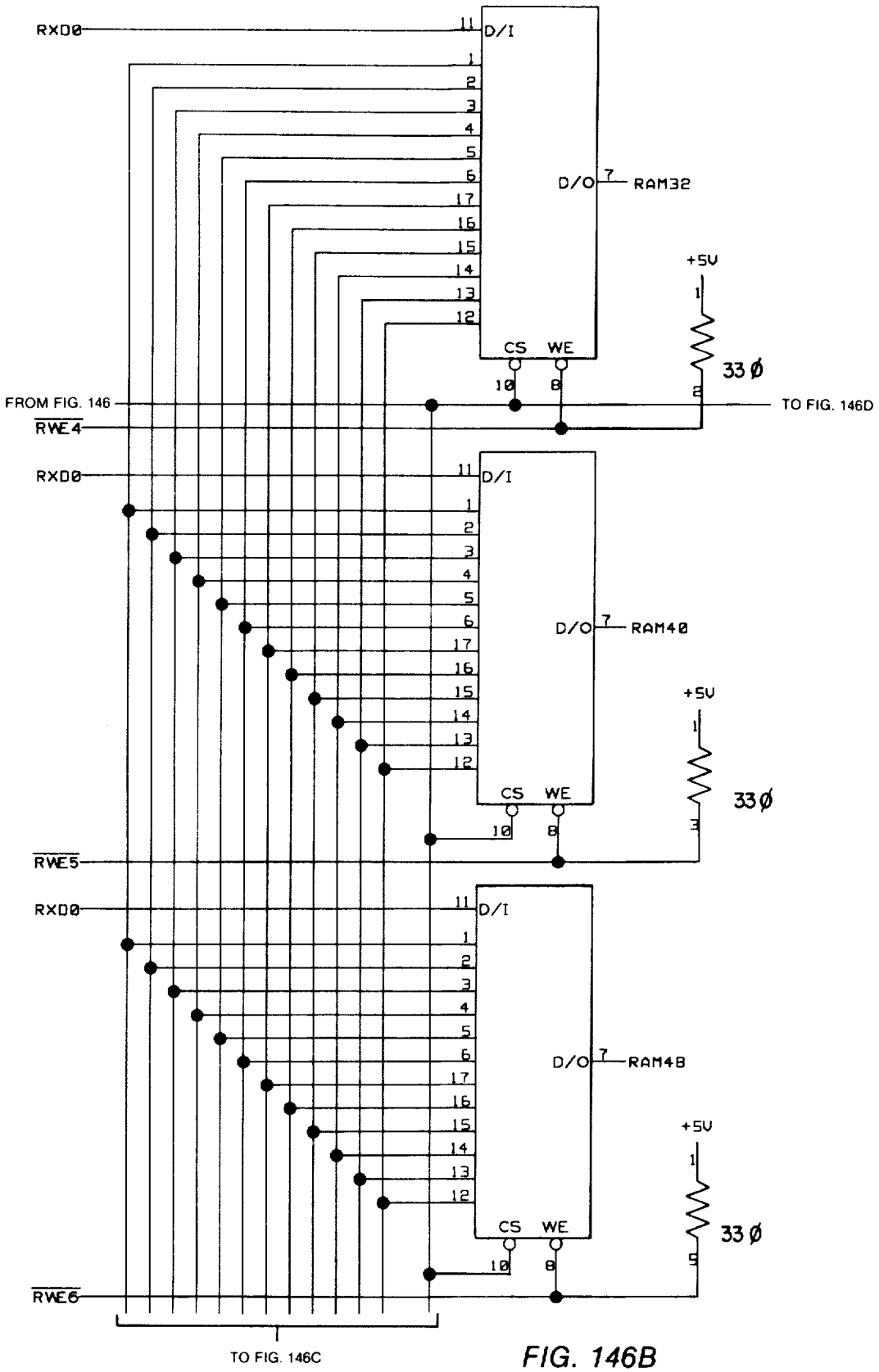

RXD0
D/I
D/O
RAM32
+5V
33Ø
CS
WE
FROM FIG. 146
RWE4
TO FIG. 146D
RXD0
D/I
D/O
RAM40
+5V
33Ø
CS
WE
RWE5
RXD0
D/I
D/O
RAM48
+5V
33Ø
CS
WE
RWE6
TO FIG. 146C

FIG. 146B

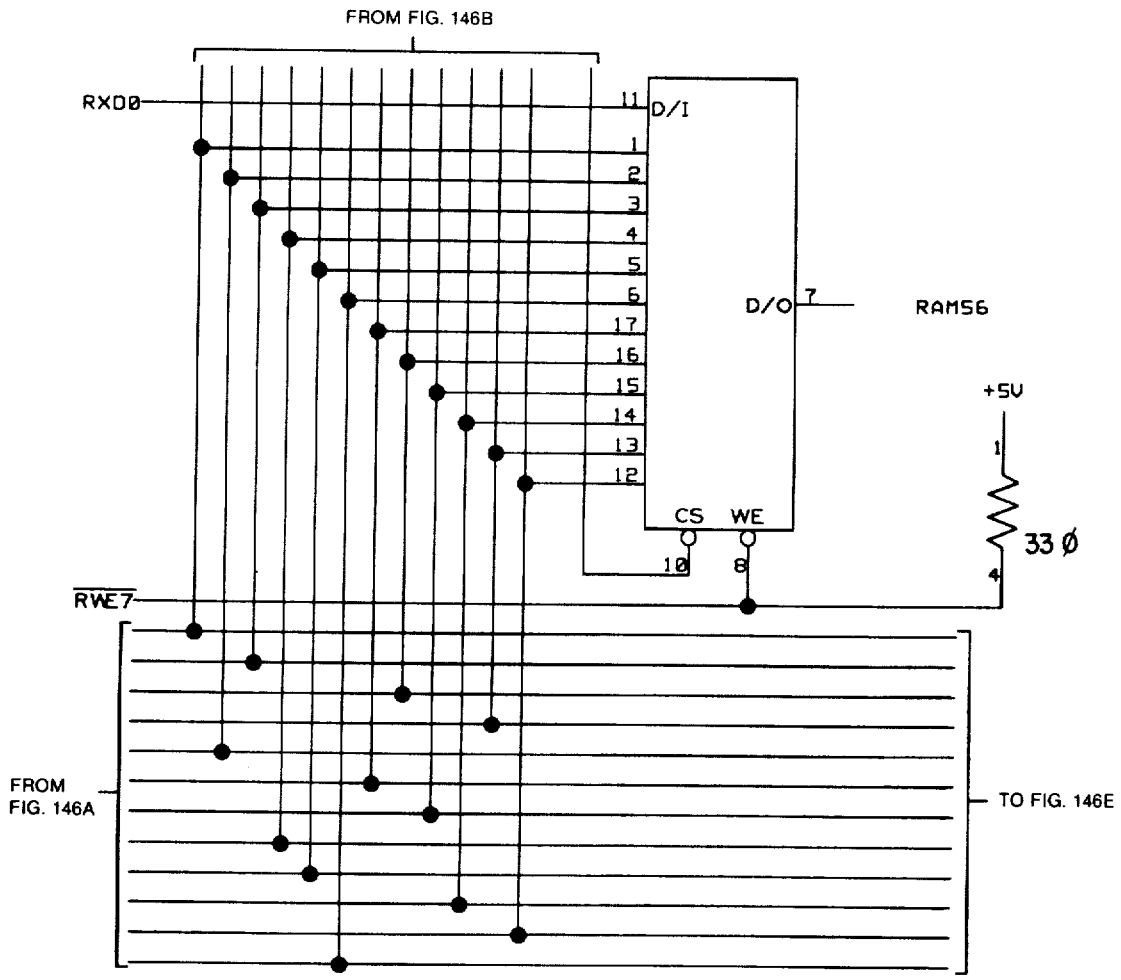

FROM FIG. 146B
RXD0
11
D/I
1
2
3
4
5
6
17
16
15
14
13
12
D/O
7
RAM56
+5V
1
33 Ø
4
CS
WE
10
8
RWE7
FROM
FIG. 146A
TO FIG. 146E

FIG. 146C

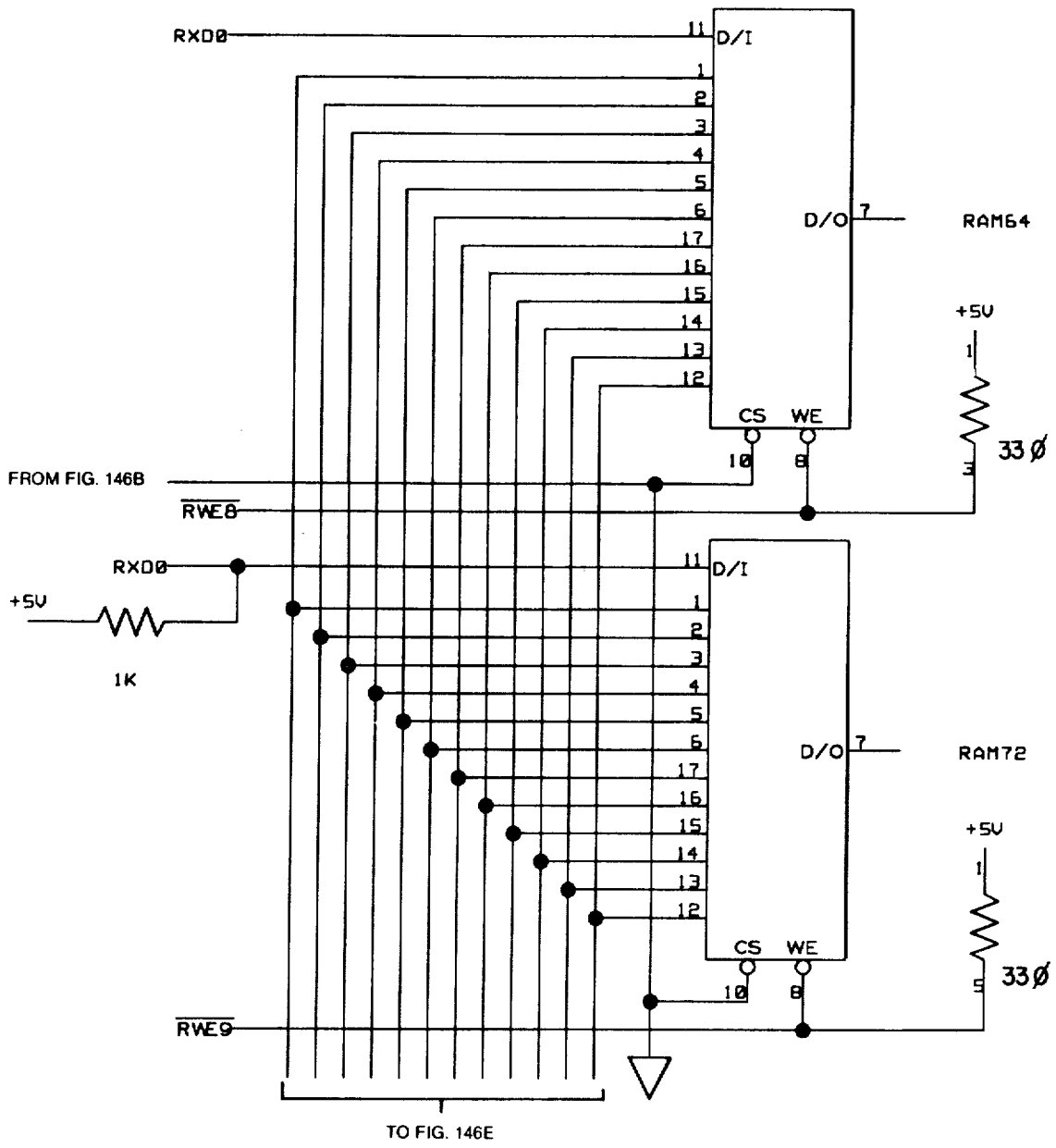

RXD0
D/I
D/O
RAM64
+5V
33Ø
CS
WE
FROM FIG. 146B
RWE8
RXD0
+5V
1K
D/I
D/O
RAM72
+5V
33Ø
CS
WE
RWE9
TO FIG. 146E

FIG. 146D

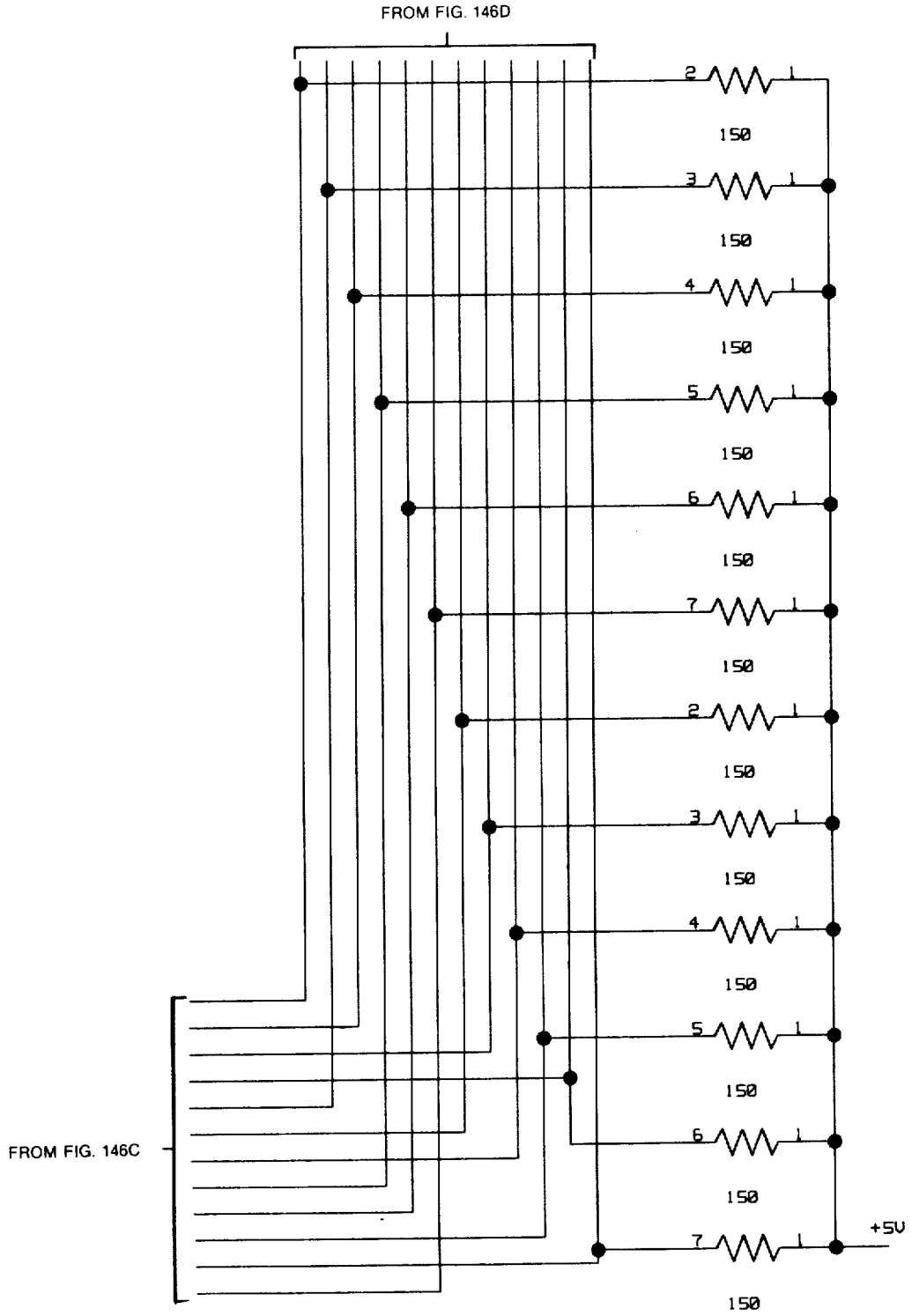


FIG. 146E

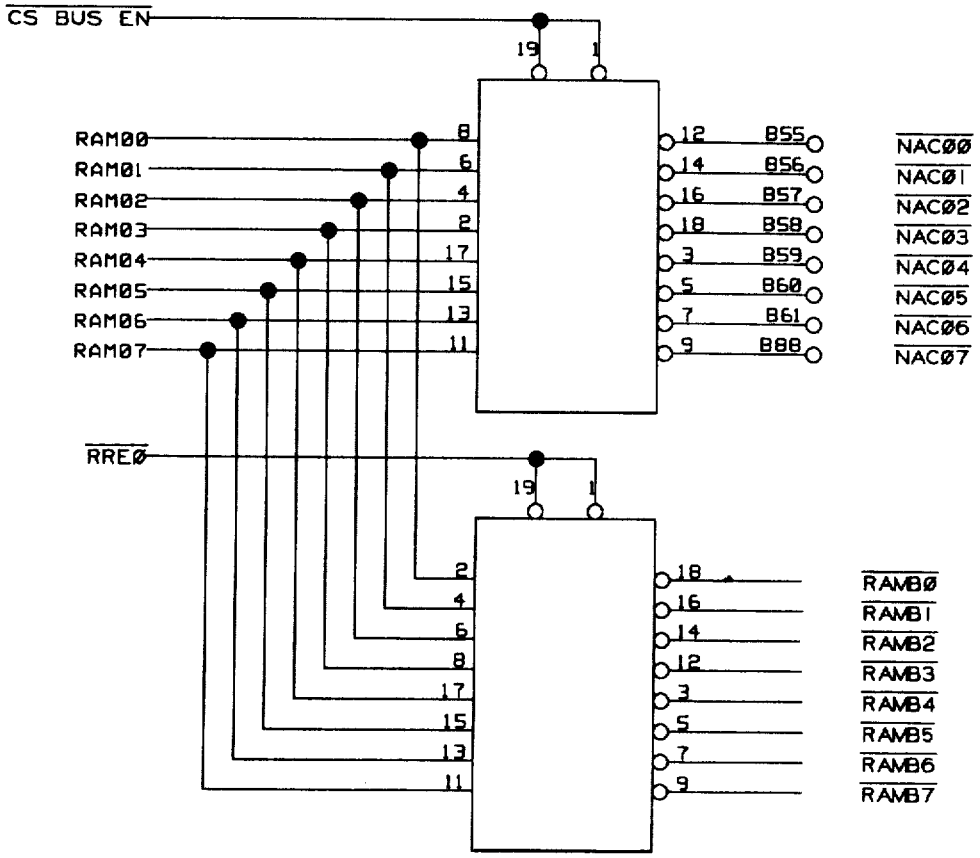
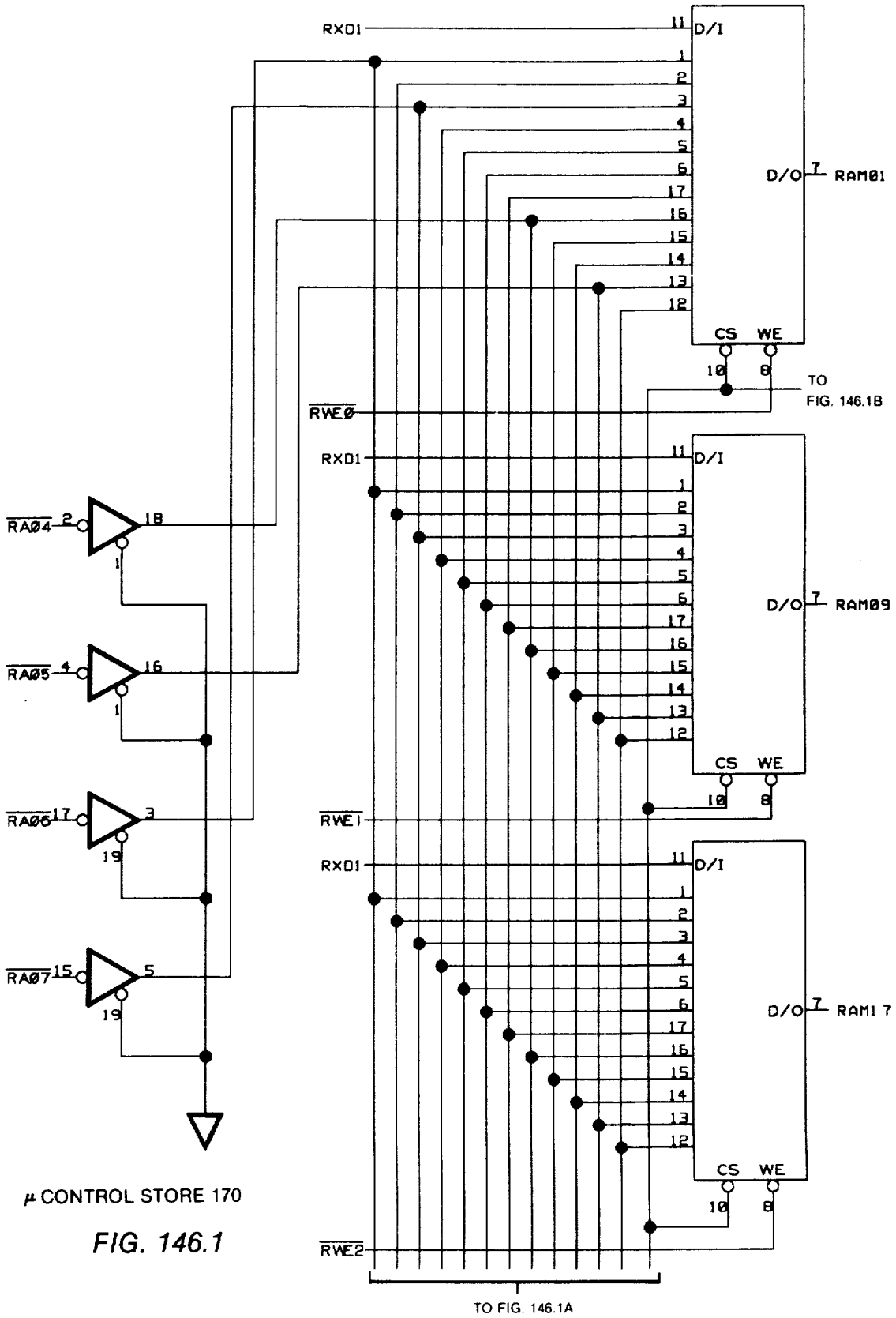

CS BUS EN
19
1
RAM00
RAM01
RAM02
RAM03
RAM04
RAM05
RAM06
RAM07
8
6
4
2
17
15
13
11
12
14
16
18
3
5
7
9
B55
B56
B57
B58
B59
B60
B61
B88
NAC00
NAC01
NAC02
NAC03
NAC04
NAC05
NAC06
NAC07
RRE0
19
1
2
4
6
8
17
15
13
11
18
16
14
12
3
5
7
9
RAMB0
RAMB1
RAMB2
RAMB3
RAMB4
RAMB5
RAMB6
RAMB7

FIG. 146F



μ CONTROL STORE 170

FIG. 146.1

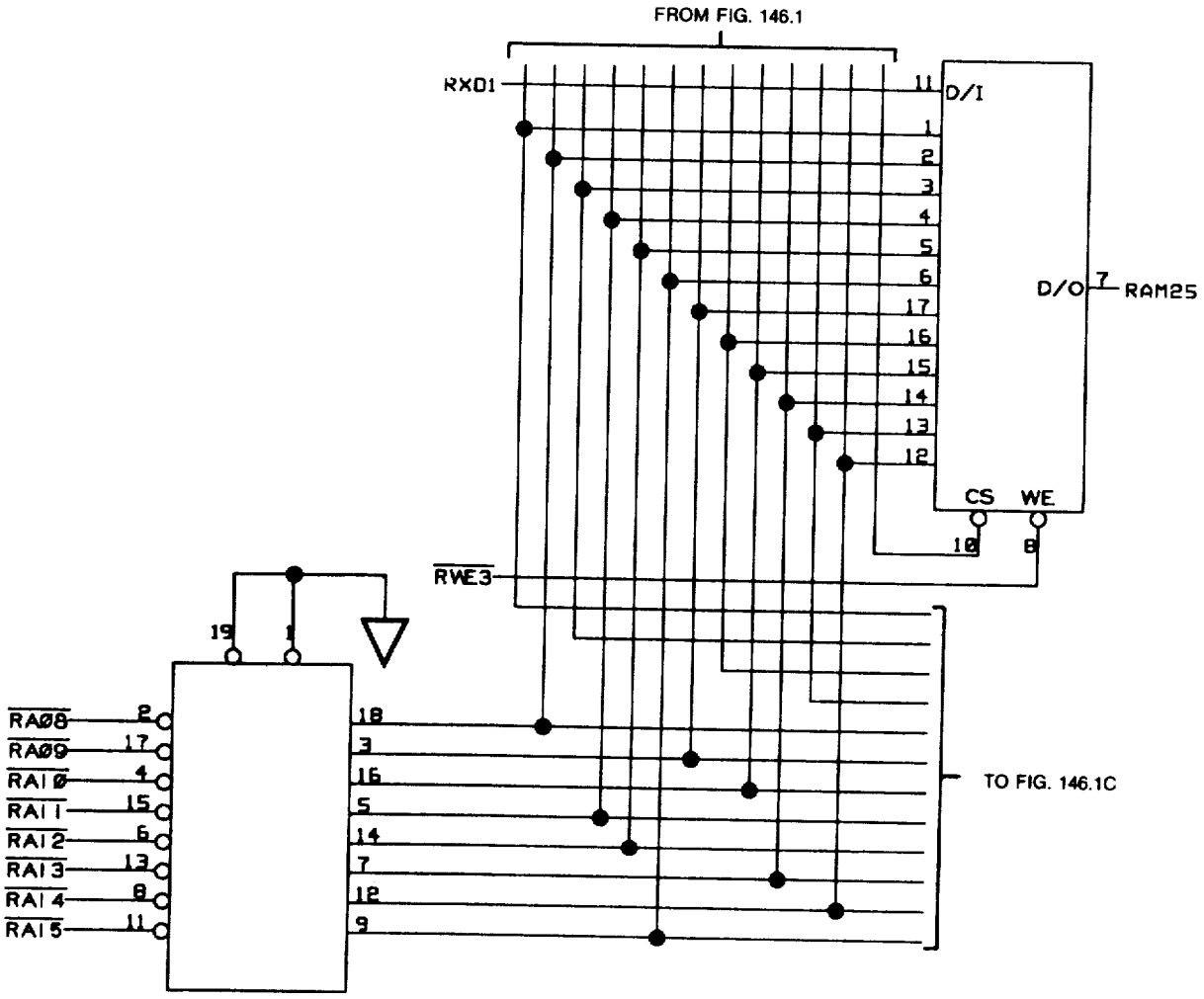

FROM FIG. 146.1
RXD1
D/I
D/O
RAM25
CS
WE
RWE3
RAØ8
RAØ9
RAI Ø
RAI 1
RAI 2
RAI 3
RAI 4
RAI 5
TO FIG. 146.1C

FIG. 146.1A

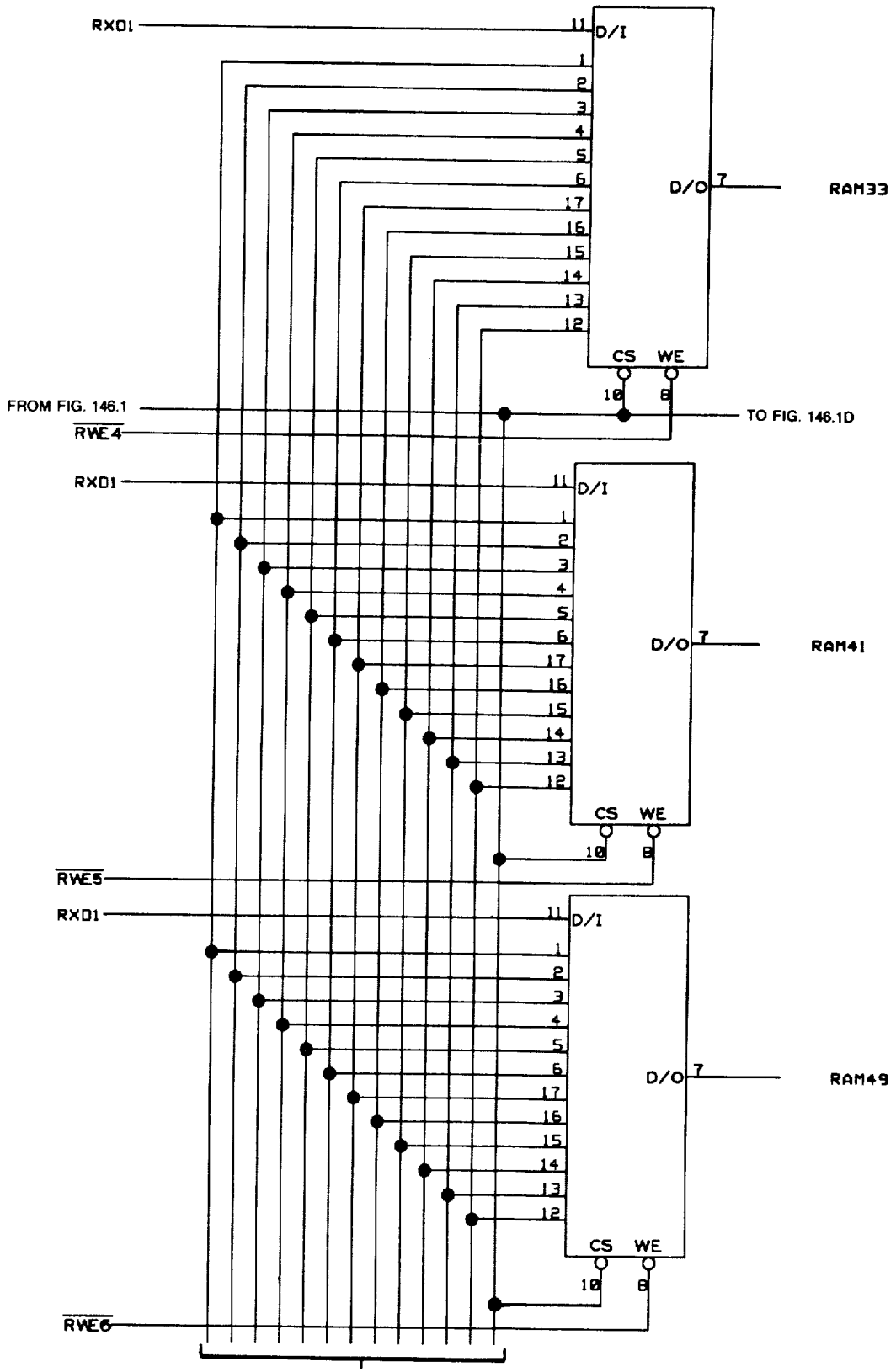

RXD1
D/I
D/O
RAM33
CS
WE
FROM FIG. 146.1
TO FIG. 146.1D
RWE4
RXD1
D/I
D/O
RAM41
CS
WE
RWE5
RXD1
D/I
D/O
RAM49
CS
WE
RWE6
TO FIG. 146.1C

FIG. 146.1B

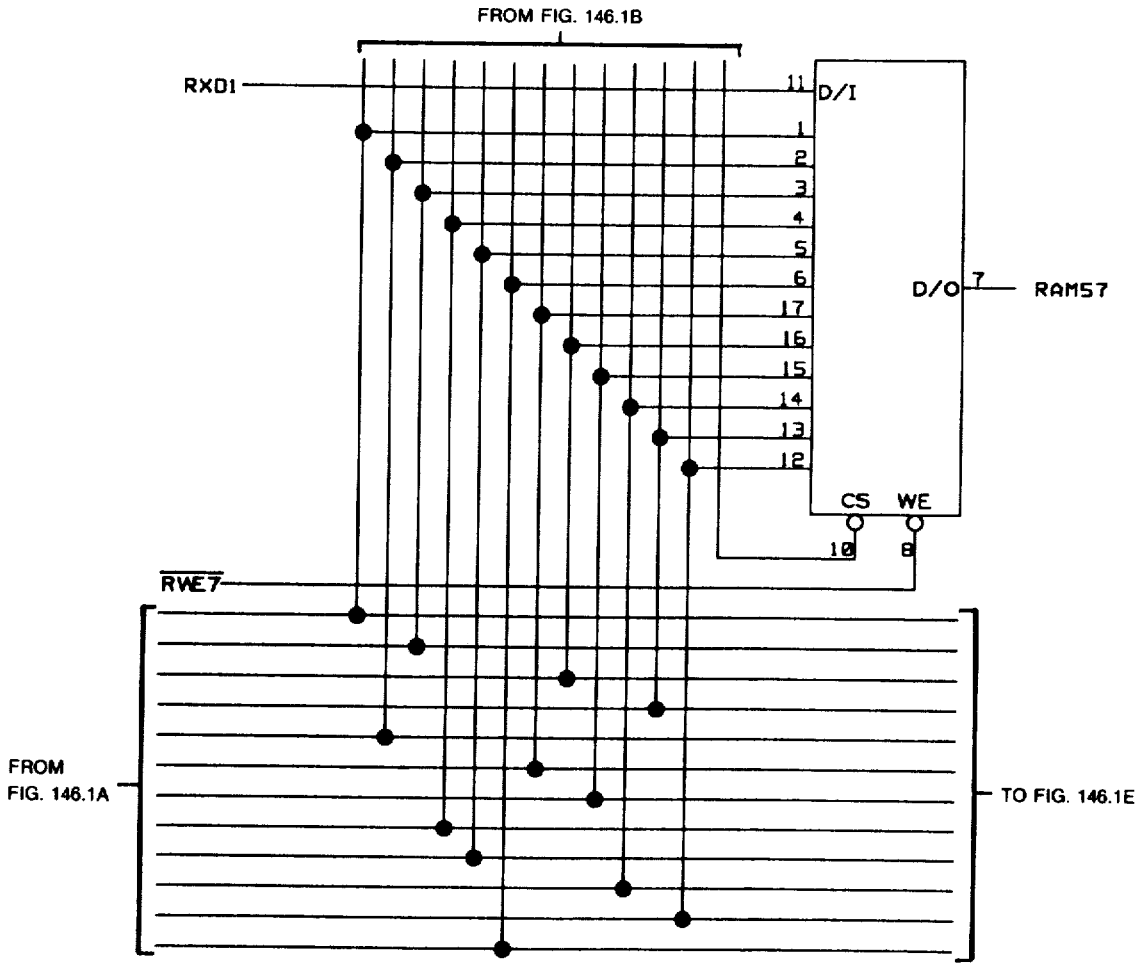

FROM FIG. 146.1B
RXD1
11
D/I
1
2
3
4
5
6
17
16
15
14
13
12
D/O
7
RAM57
CS
WE
10
8
RWE7
FROM
FIG. 146.1A
TO FIG. 146.1E

FIG. 146.1C

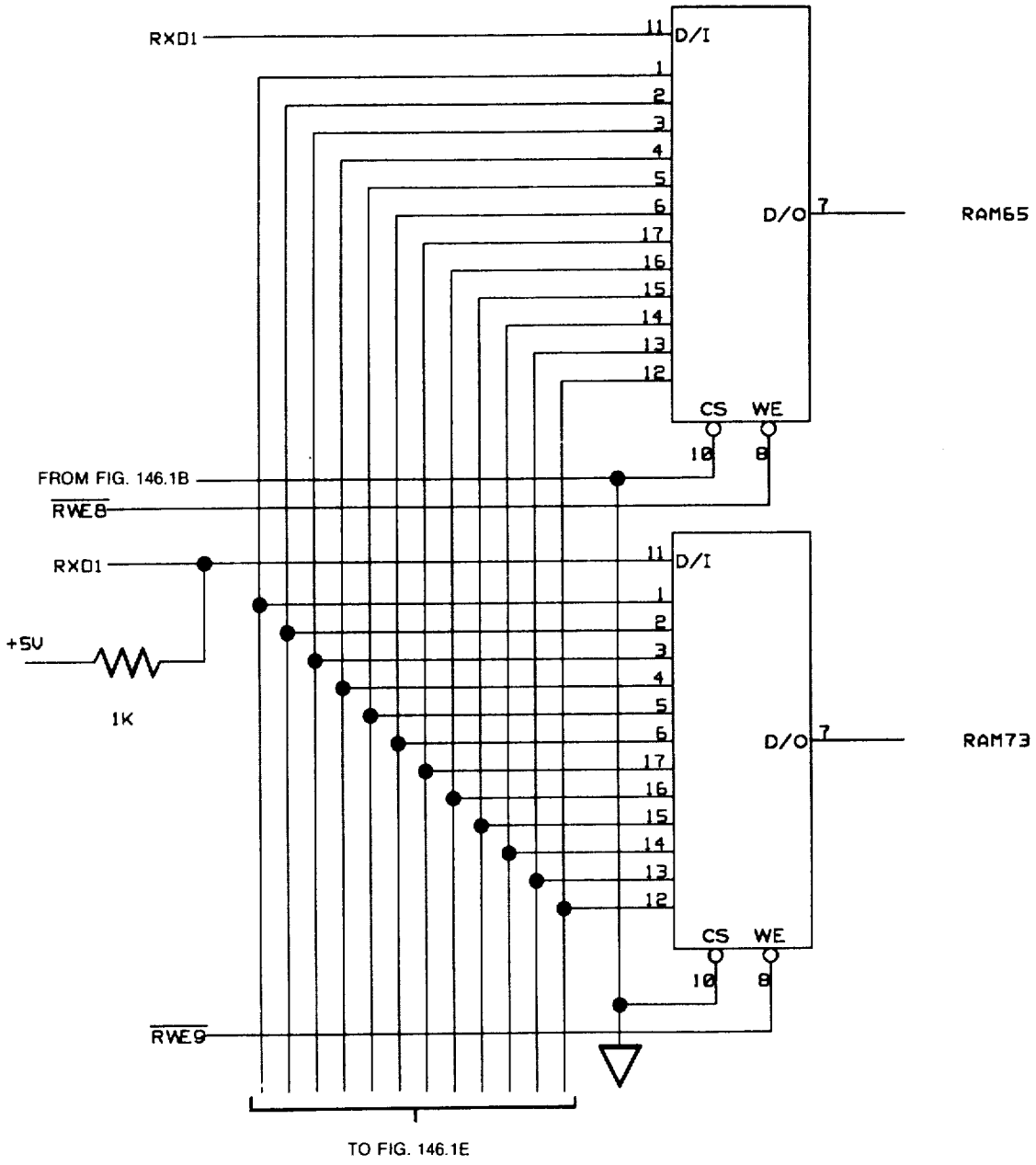

RXD1
D/I
D/O
RAM65
CS
WE
FROM FIG. 146.1B
RWE8
RXD1
+5V
1K
D/I
D/O
RAM73
CS
WE
RWE9
TO FIG. 146.1E

FIG. 146.1D

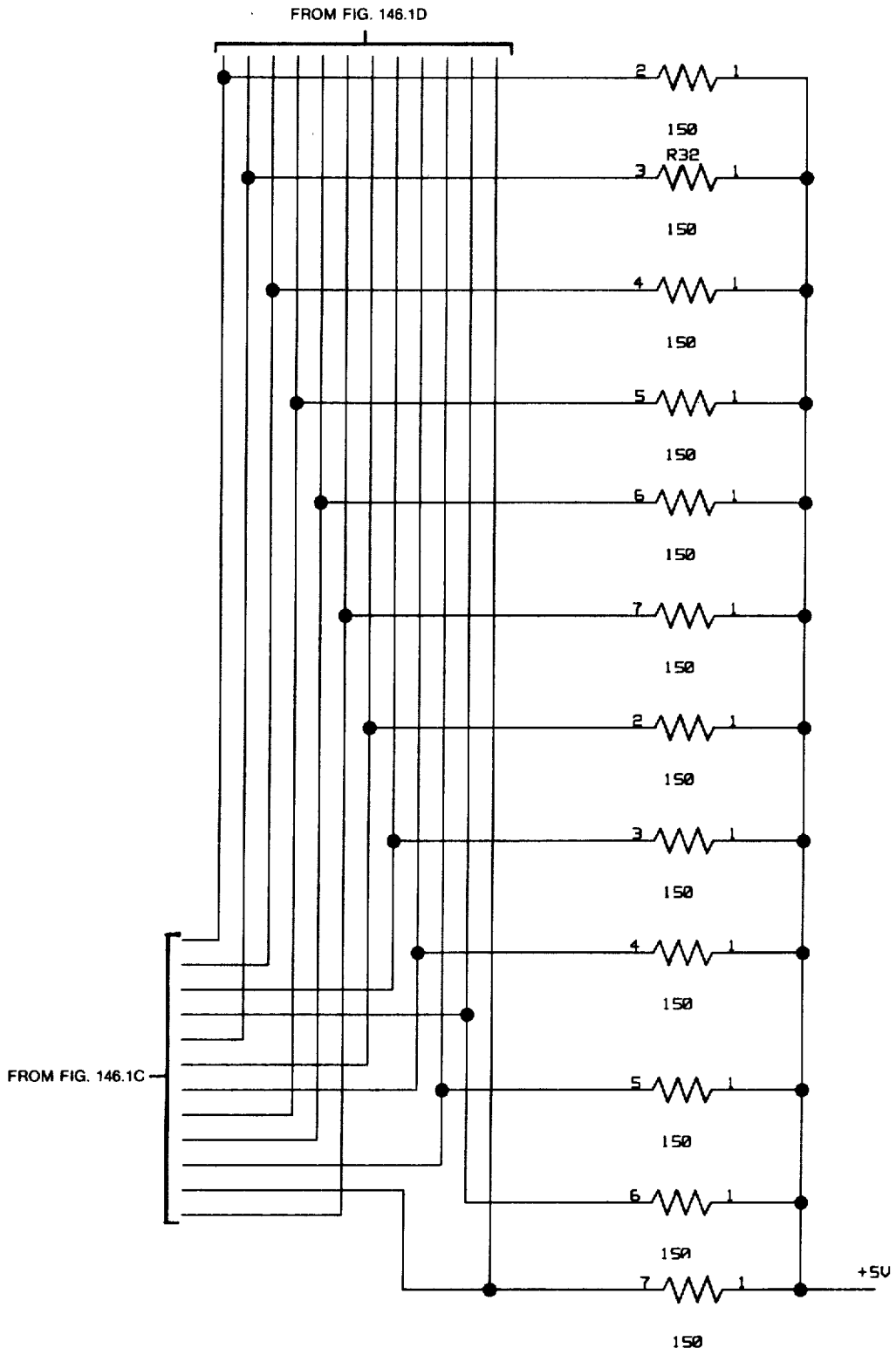


FIG. 146.1E

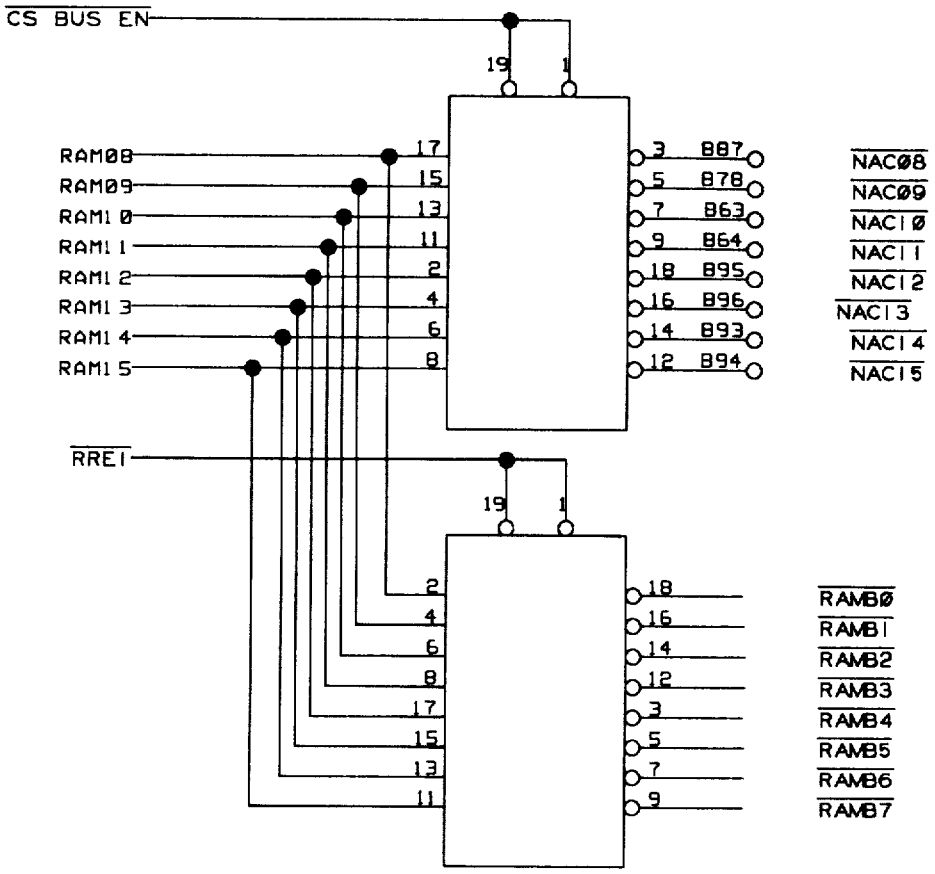
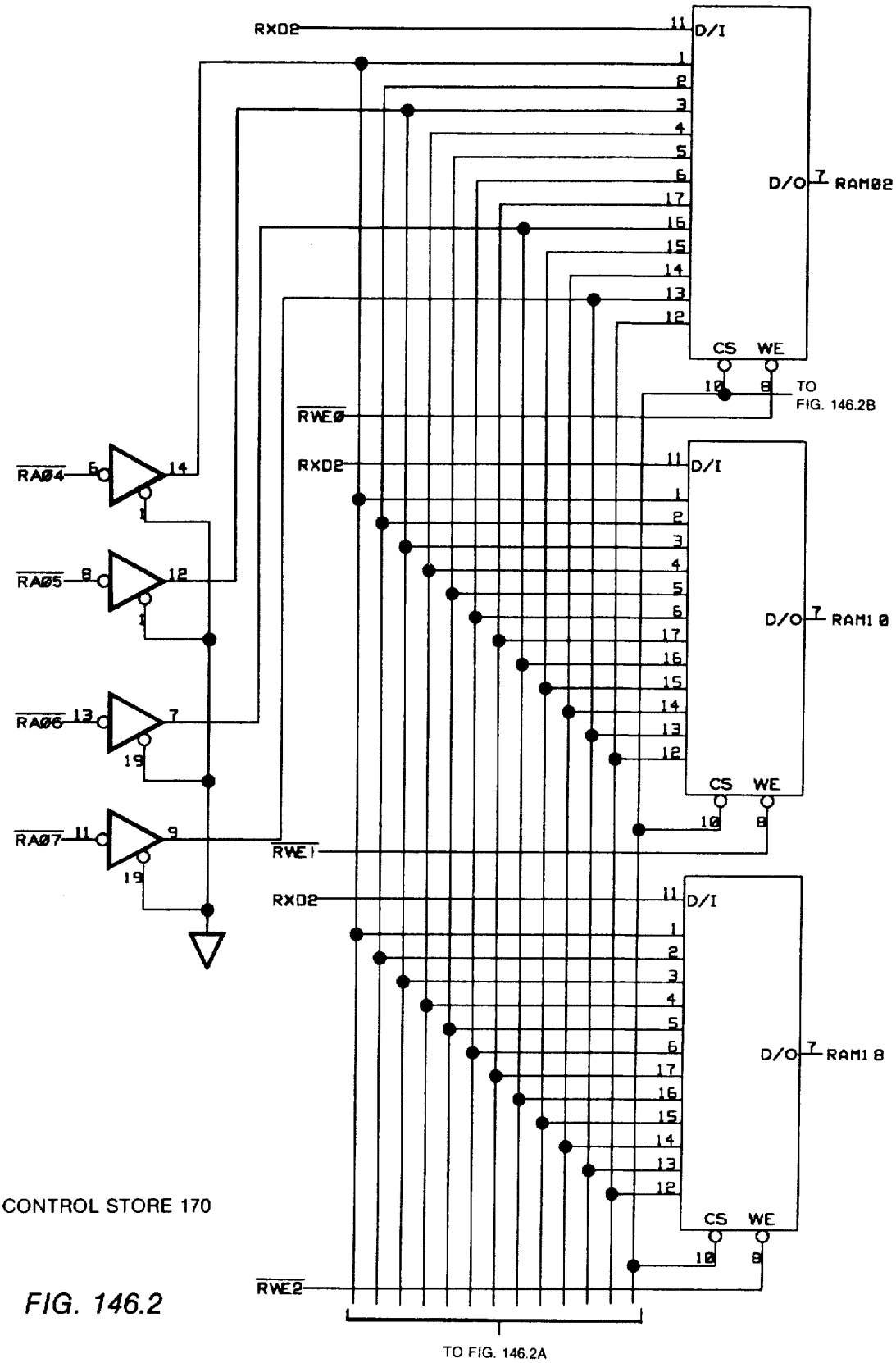

CS BUS EN
19
1
RAM08
RAM09
RAM10
RAM11
RAM12
RAM13
RAM14
RAM15
17
15
13
11
2
4
6
8
3 B87
5 B78
7 B63
9 B64
18 B95
16 B96
14 B93
12 B94
NAC08
NAC09
NAC10
NAC11
NAC12
NAC13
NAC14
NAC15
RRE1
19
1
2
4
6
8
17
15
13
11
18
16
14
12
3
5
7
9
RAMB0
RAMB1
RAMB2
RAMB3
RAMB4
RAMB5
RAMB6
RAMB7

FIG. 146.1F



μCONTROL STORE 170

FIG. 146.2

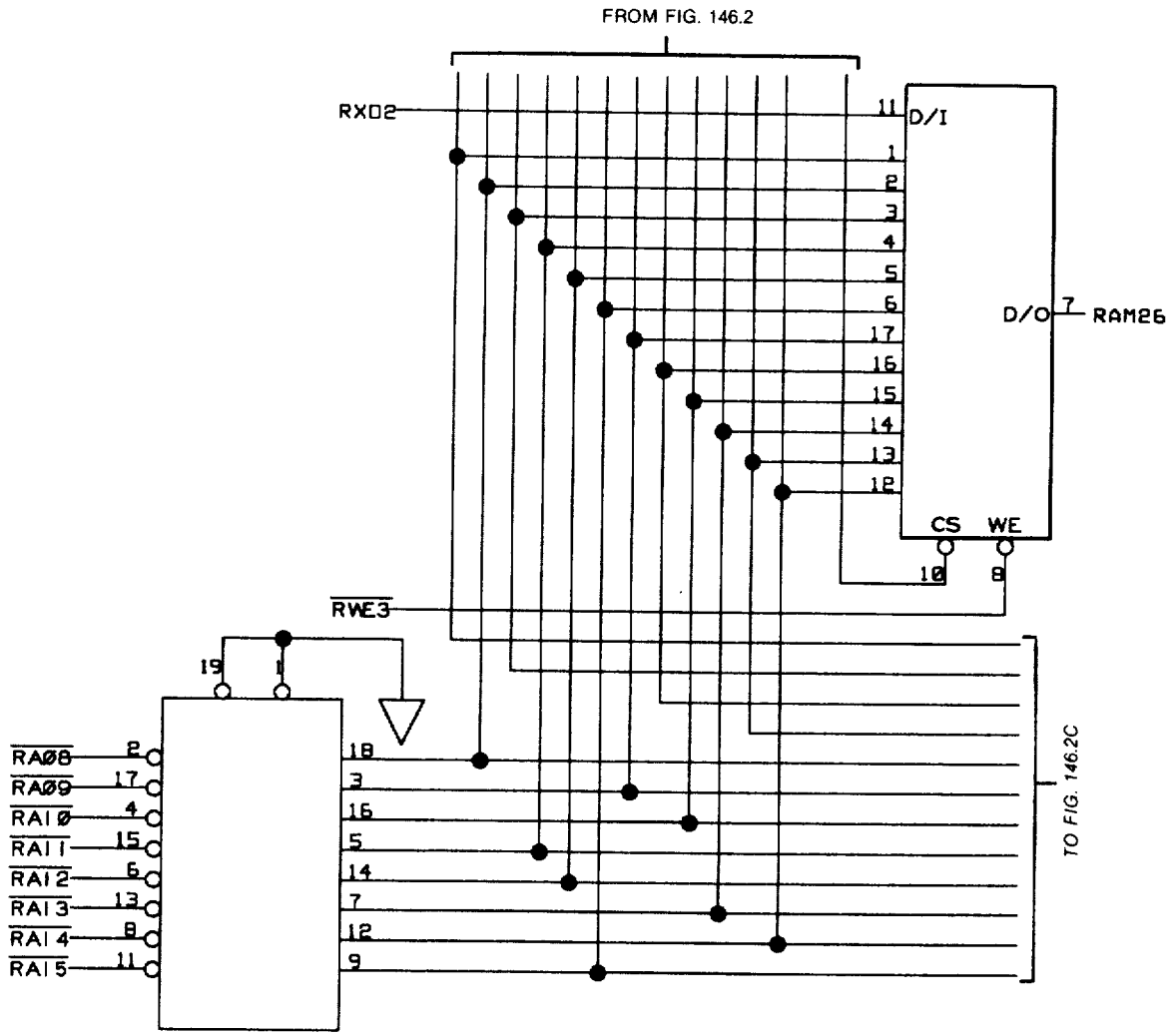

FROM FIG. 146.2
RX02
D/I
D/O
RAM26
CS
WE
RWE3
RAØ8
RAØ9
RAIØ
RAI1
RAI2
RAI3
RAI4
RAI5
TO FIG. 146.2C

FIG. 146.2A

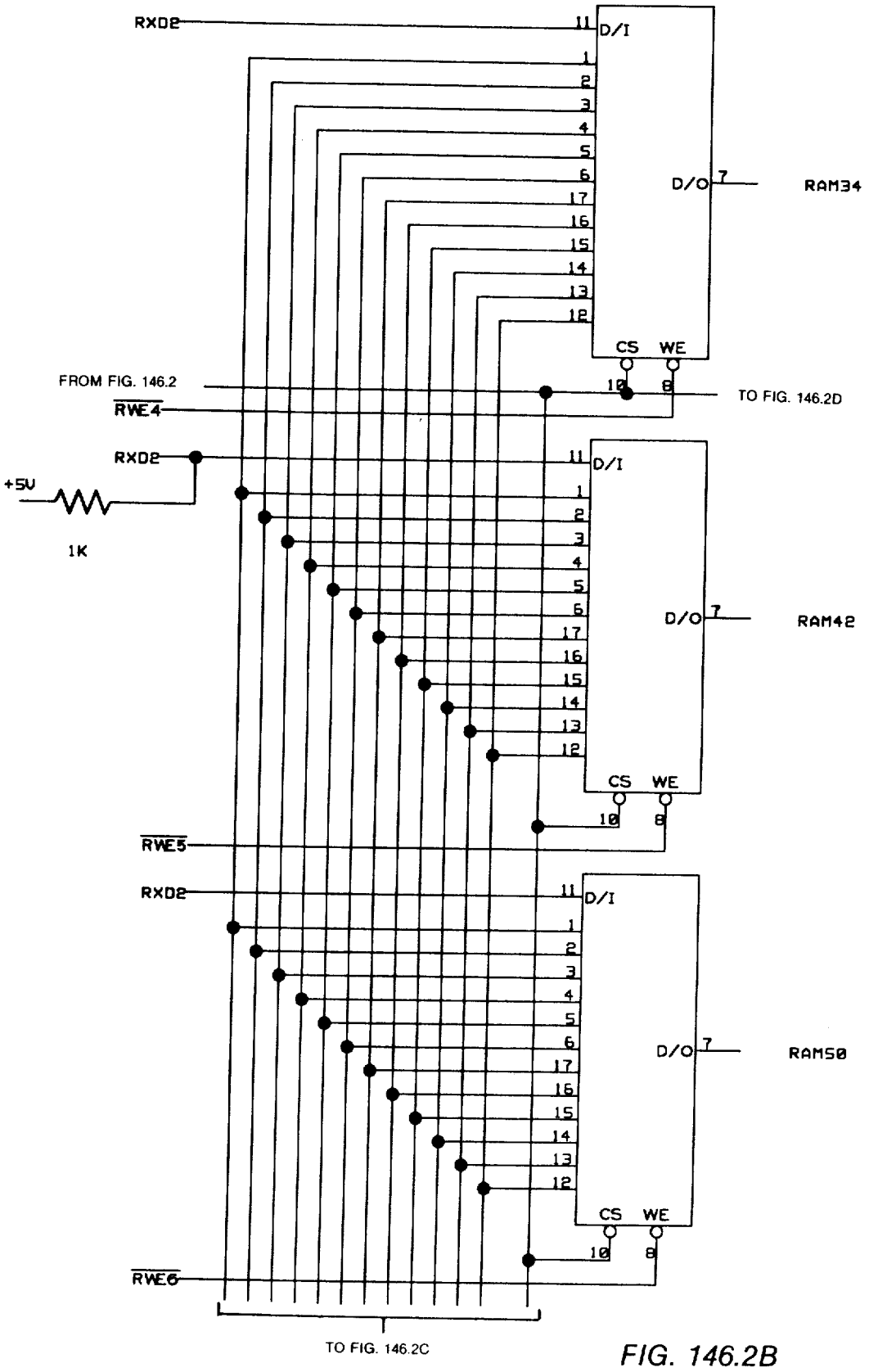

RXD2
D/I
D/O
RAM34
CS
WE
FROM FIG. 146.2
RWE4
TO FIG. 146.2D
RXD2
+5V
1K
D/I
D/O
RAM42
CS
WE
RWE5
RXD2
D/I
D/O
RAM50
CS
WE
RWE6
TO FIG. 146.2C

FIG. 146.2B

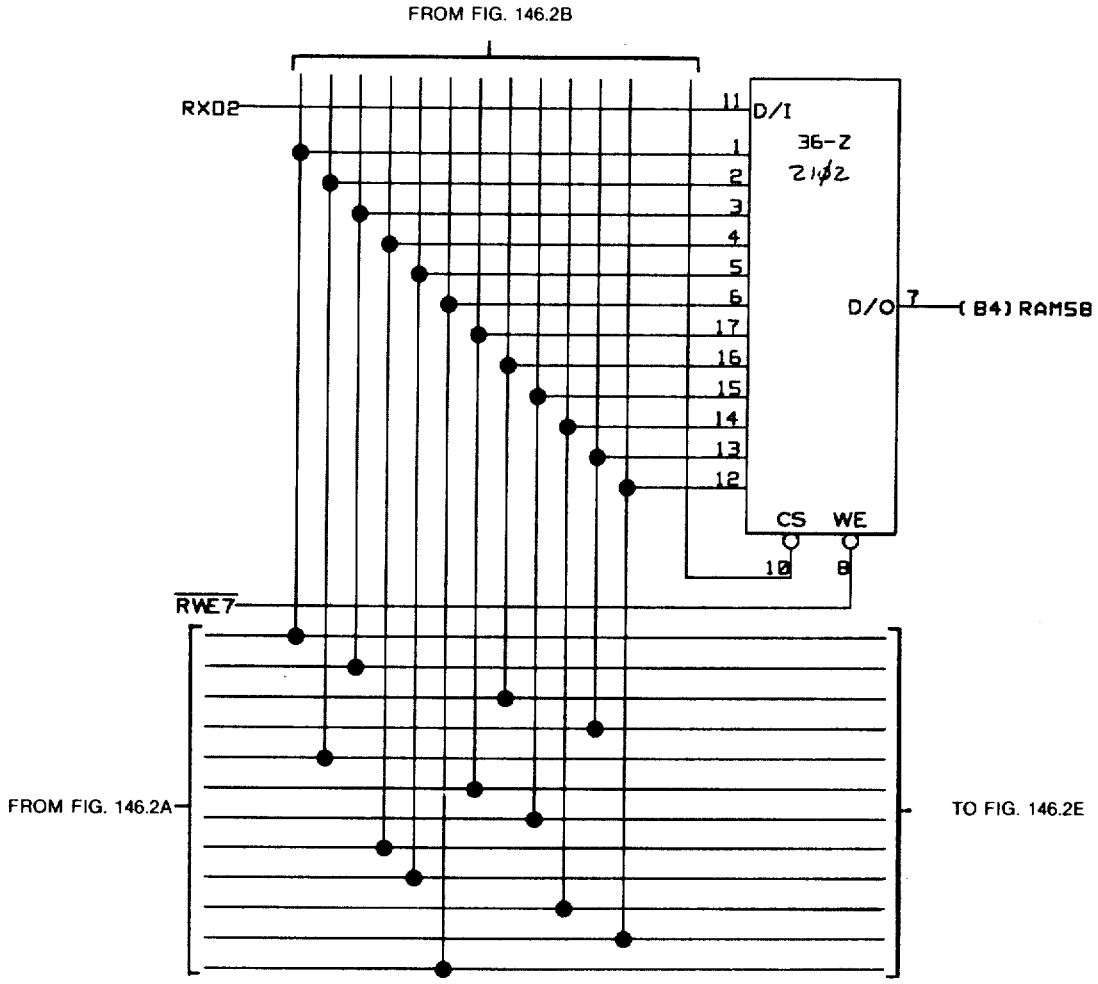

FROM FIG. 146.2B
RXD2
11
D/I
1
36-2
2
21Ø2
3
4
5
6
D/O
7
(B4) RAMSB
17
16
15
14
13
12
CS
WE
10
8
RWE7
FROM FIG. 146.2A
TO FIG. 146.2E

FIG. 146.2C

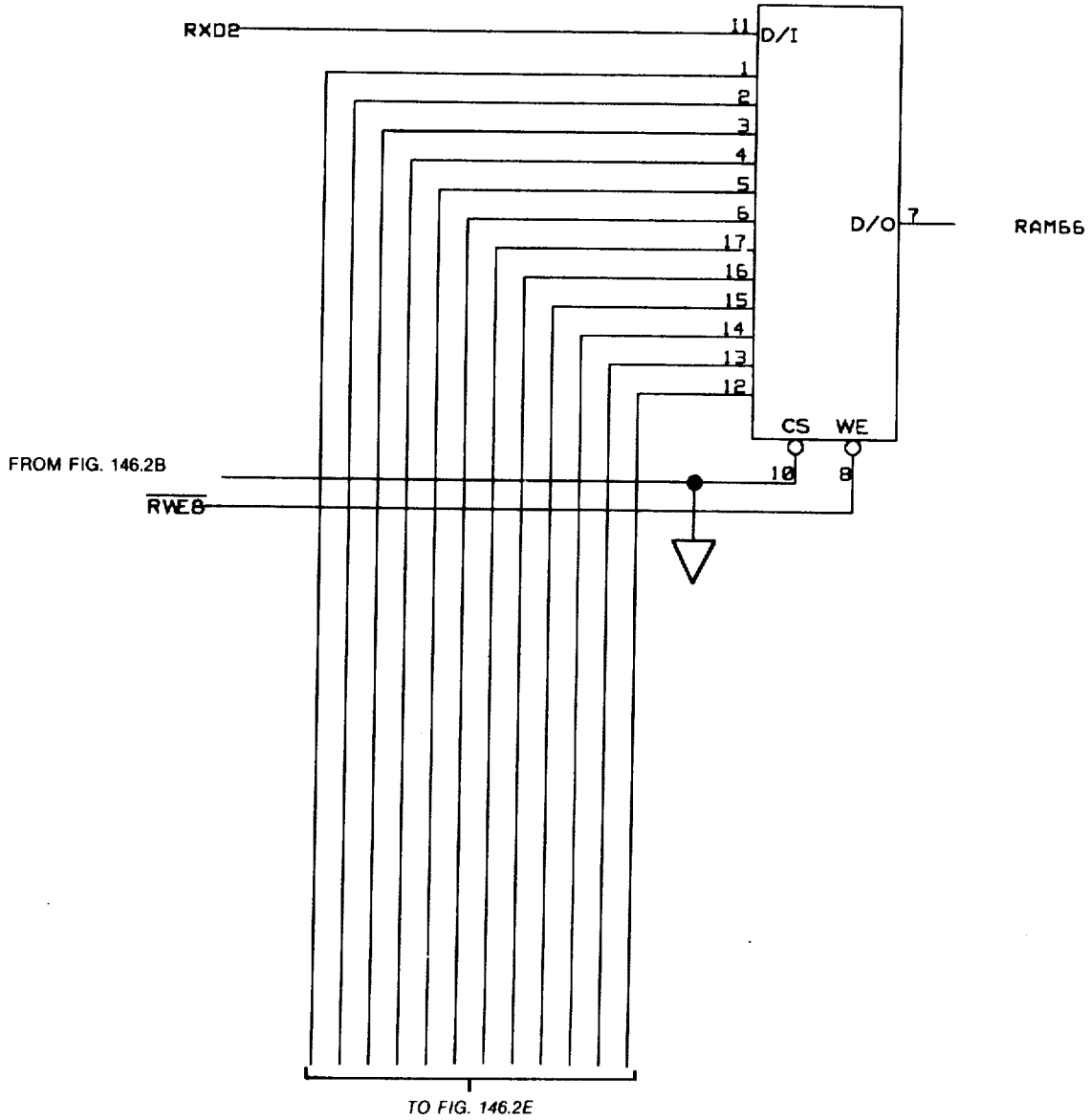

RXD2
11
D/I
1
2
3
4
5
6
17
16
15
14
13
12
D/O
7
RAM66
CS
WE
FROM FIG. 146.2B
10
8
RWE8
TO FIG. 146.2E

FIG. 146.2D

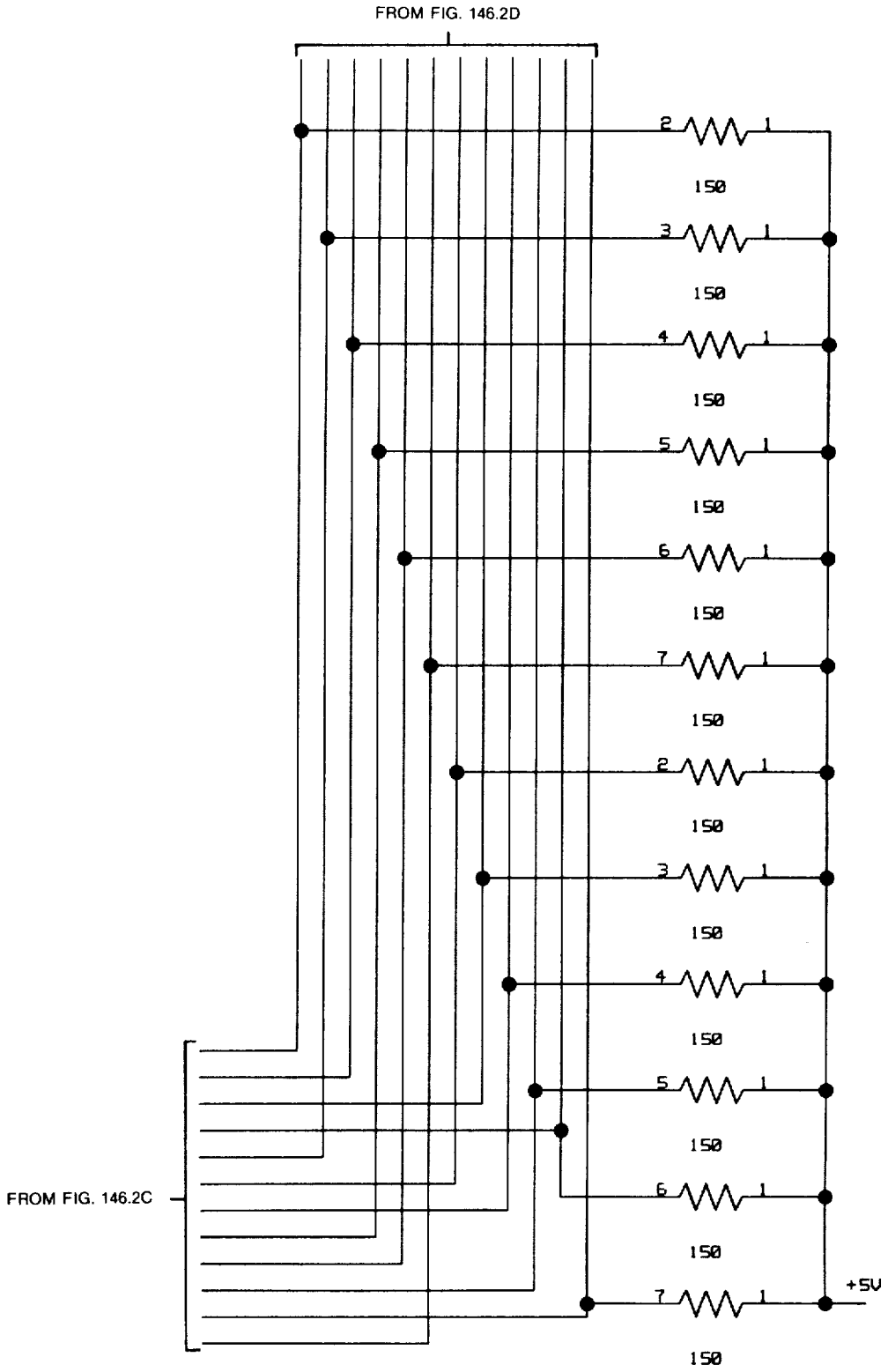


FIG. 146.2E

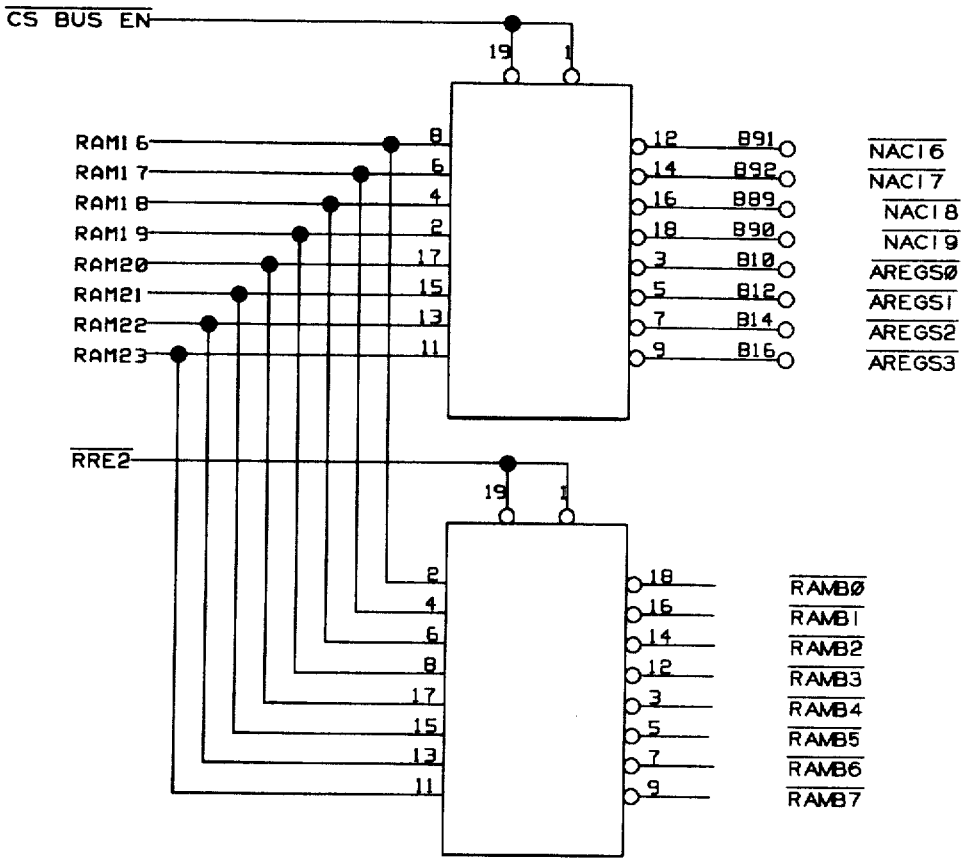

CS BUS EN
19
1
RAM16
RAM17
RAM18
RAM19
RAM20
RAM21
RAM22
RAM23
8
6
4
2
17
15
13
11
12
14
16
18
3
5
7
9
B91
B92
B89
B90
B10
B12
B14
B16
NAC16
NAC17
NAC18
NAC19
AREGS0
AREGS1
AREGS2
AREGS3
RRE2
19
1
2
4
6
8
17
15
13
11
18
16
14
12
3
5
7
9
RAMB0
RAMB1
RAMB2
RAMB3
RAMB4
RAMB5
RAMB6
RAMB7

FIG. 146.2F

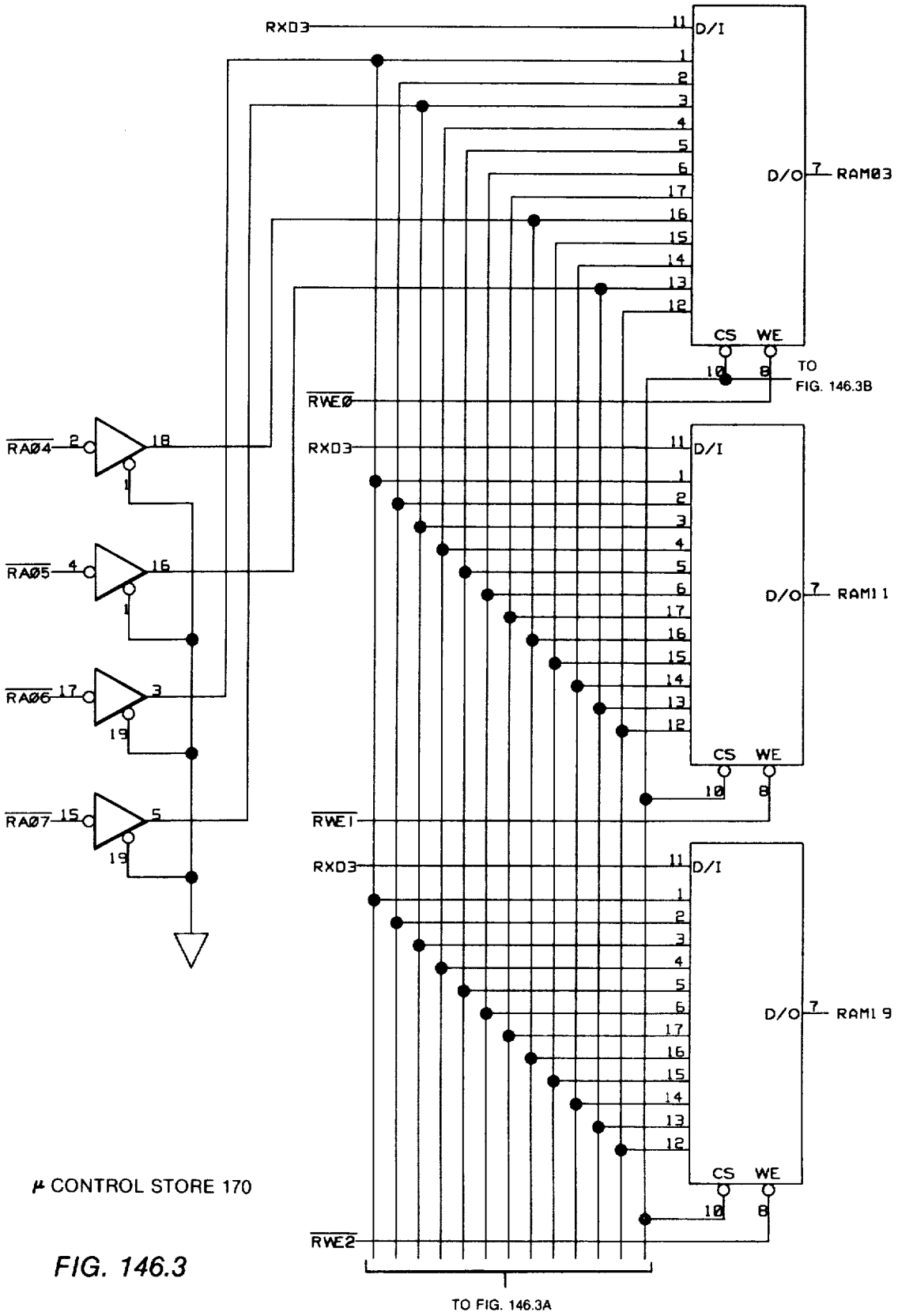

RXD3
D/I
D/O
RAMØ3
CS
WE
TO FIG. 146.3B
RWEØ
RAØ4
RAØ5
RAØ6
RAØ7
RAM11
RWE1
RAM19
RWE2
μ CONTROL STORE 170
TO FIG. 146.3A

FIG. 146.3

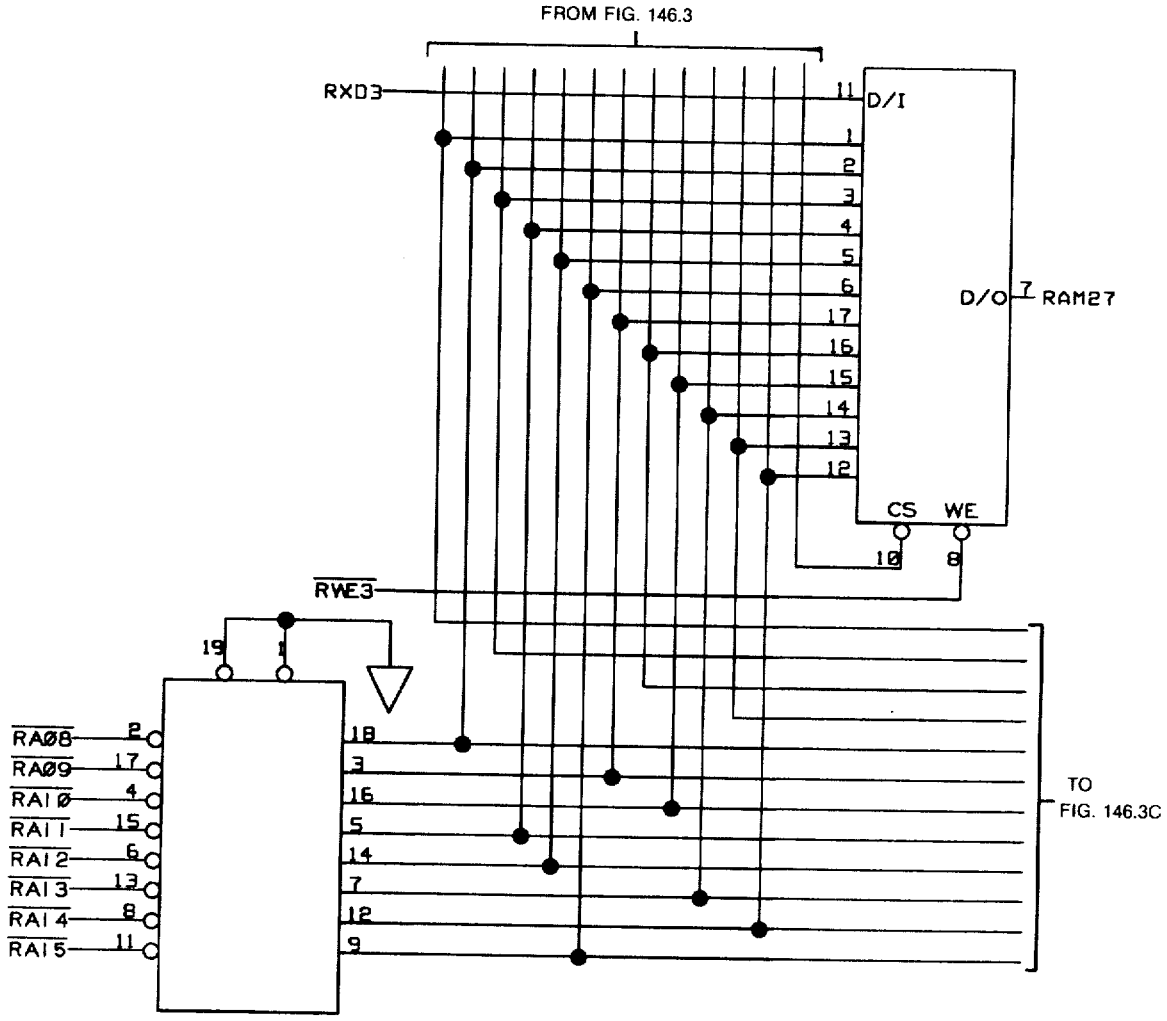

FROM FIG. 146.3
RXD3
D/I
D/O
RAM27
CS
WE
RWE3
RA08
RA09
RAI0
RAI1
RAI2
RAI3
RAI4
RAI5
TO
FIG. 146.3C

FIG. 146.3A

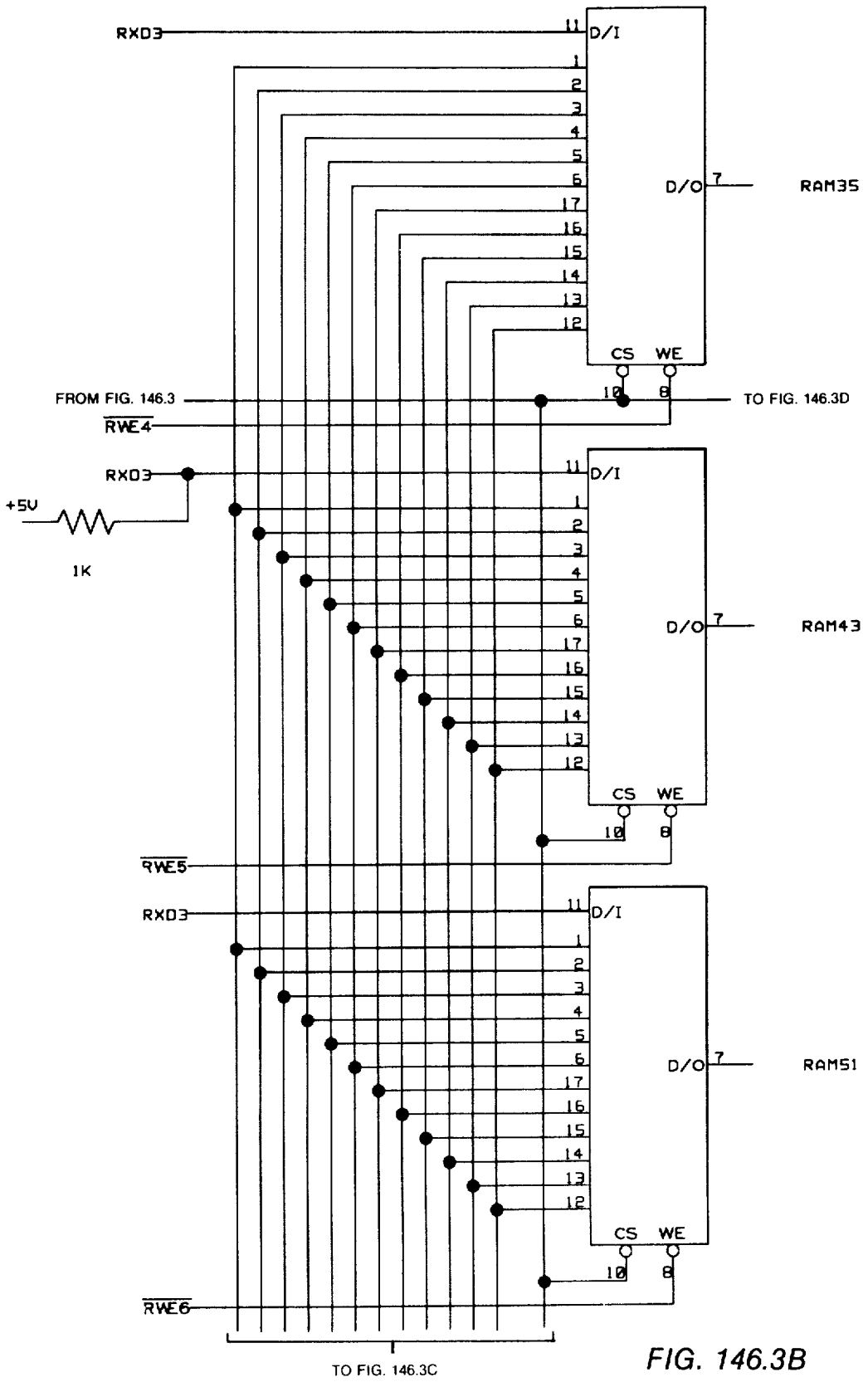

RXD3
D/I
D/O
RAM35
CS
WE
FROM FIG. 146.3
TO FIG. 146.3D
RWE4
RXD3
+5V
1K
D/I
D/O
RAM43
CS
WE
RWE5
RXD3
D/I
D/O
RAM51
CS
WE
RWE6
TO FIG. 146.3C

FIG. 146.3B

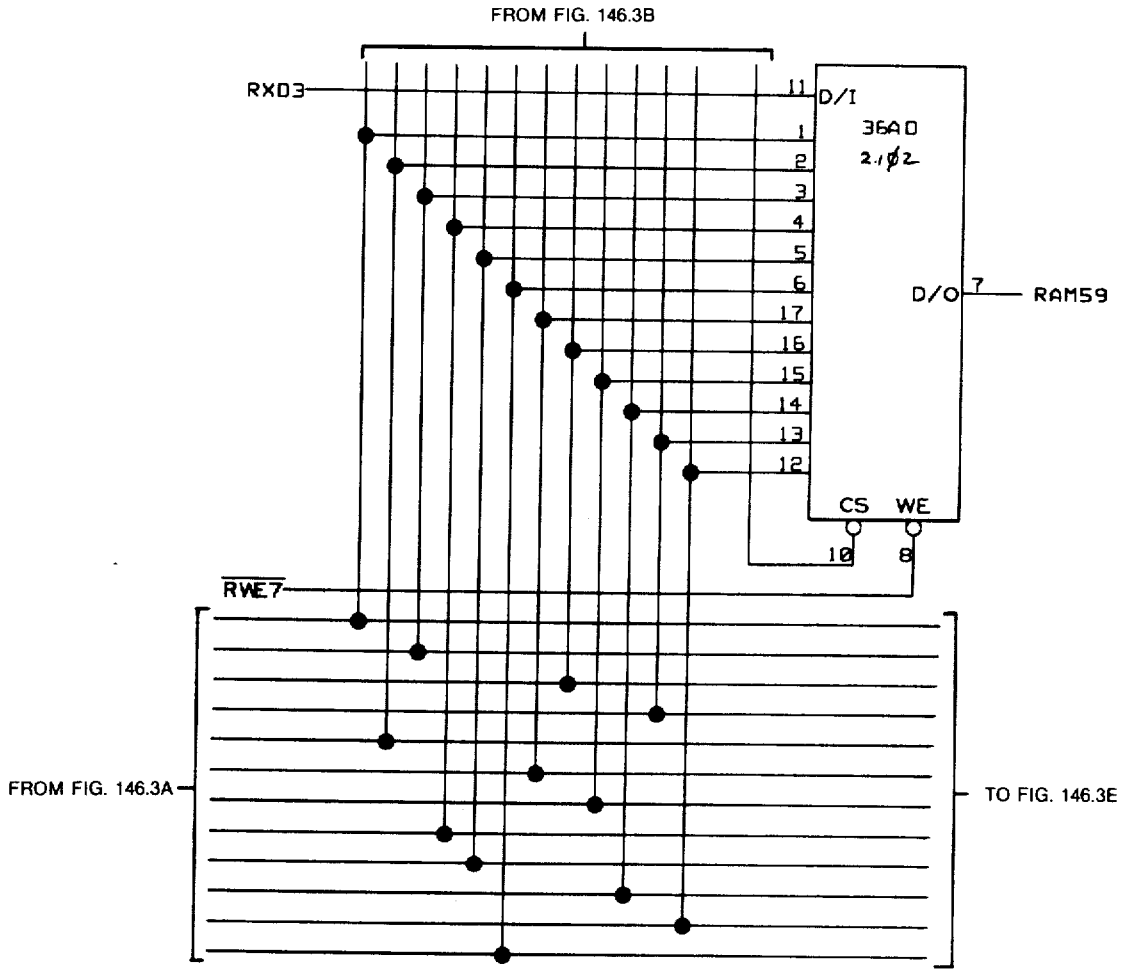

FROM FIG. 146.3B
RXD3
11
D/I
1
36A0
2.1Ø2
2
3
4
5
6
17
16
15
14
13
12
D/O
7
RAM59
CS
WE
10
8
RWE7
FROM FIG. 146.3A
TO FIG. 146.3E

FIG. 146.3C

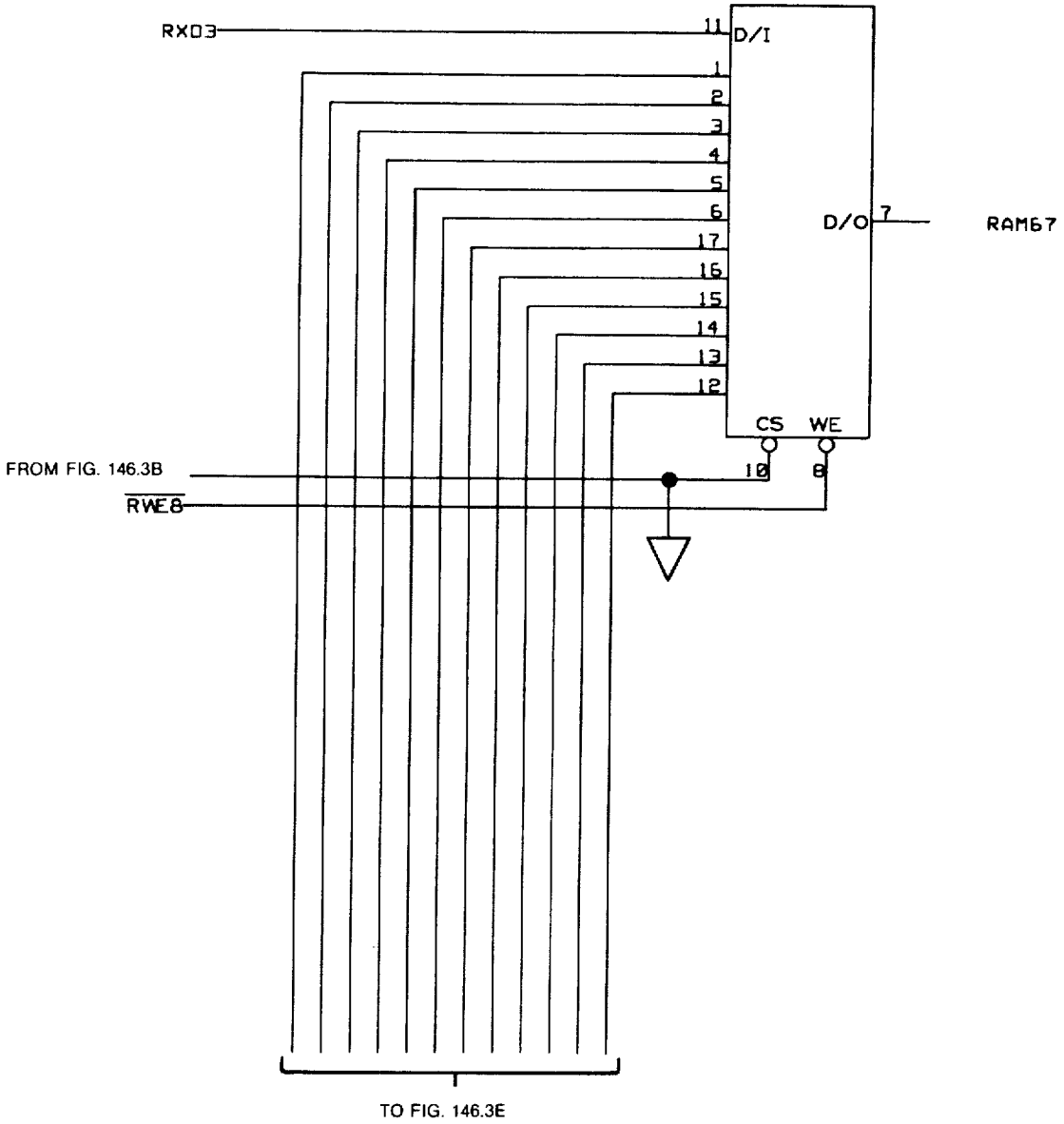

RX03
11
D/I
1
2
3
4
5
6
17
16
15
14
13
12
D/O
7
RAM67
CS
WE
10
8
FROM FIG. 146.3B
RWE8
TO FIG. 146.3E

FIG. 146.3D

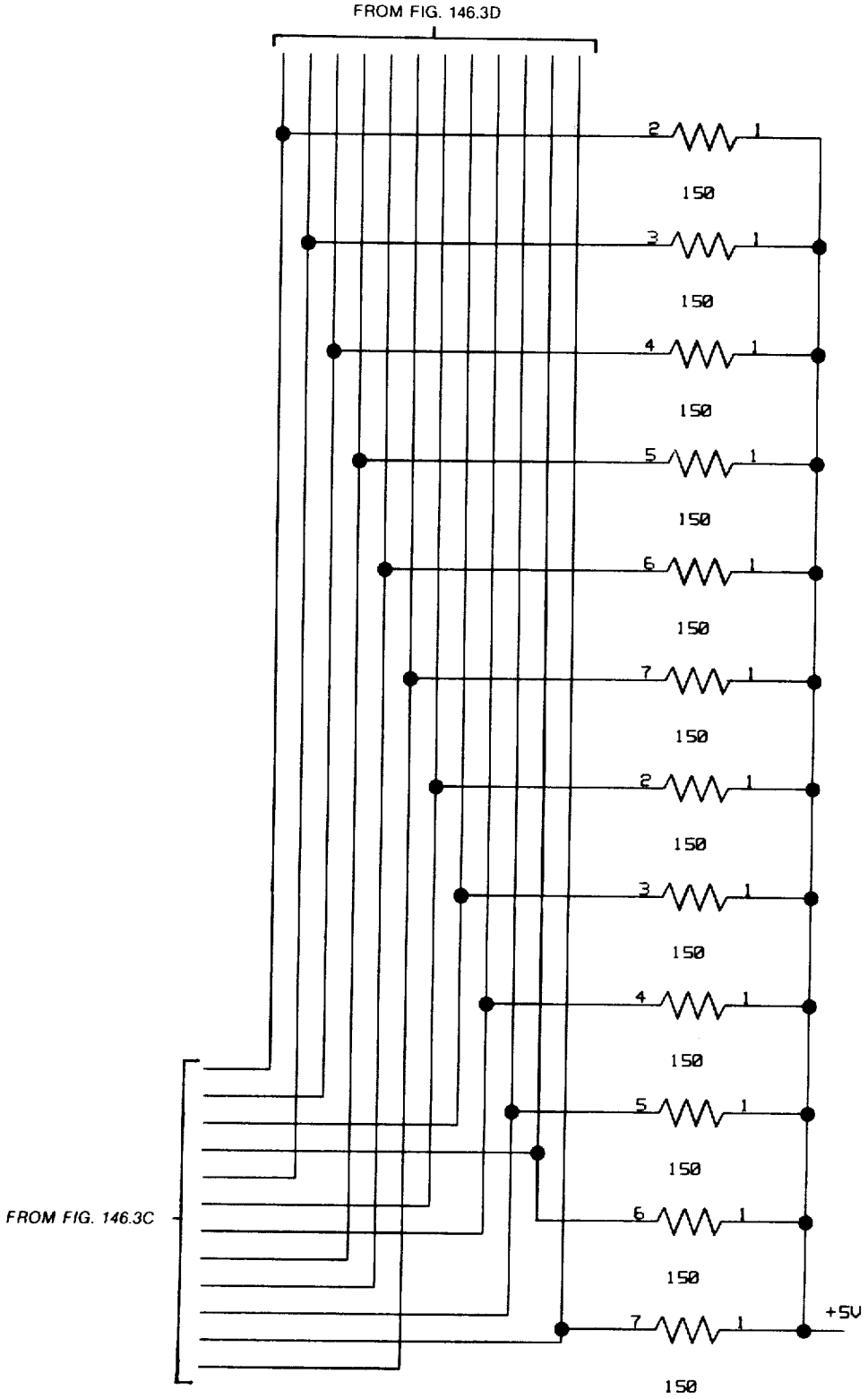


FIG. 146.3E

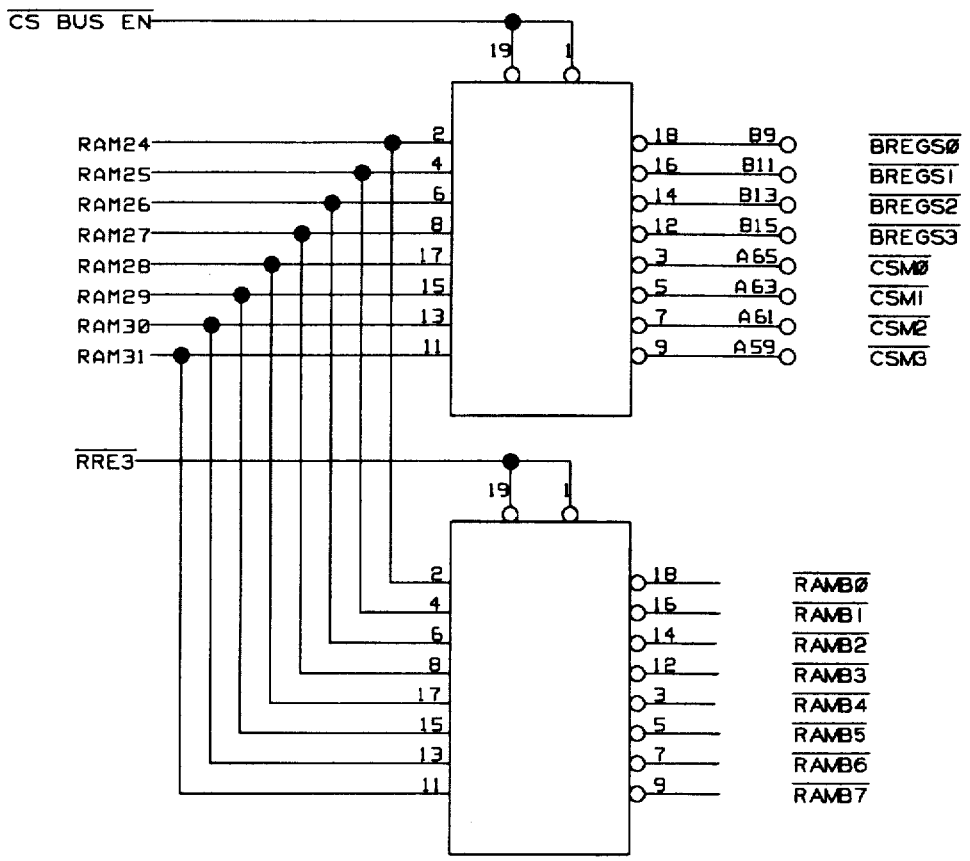
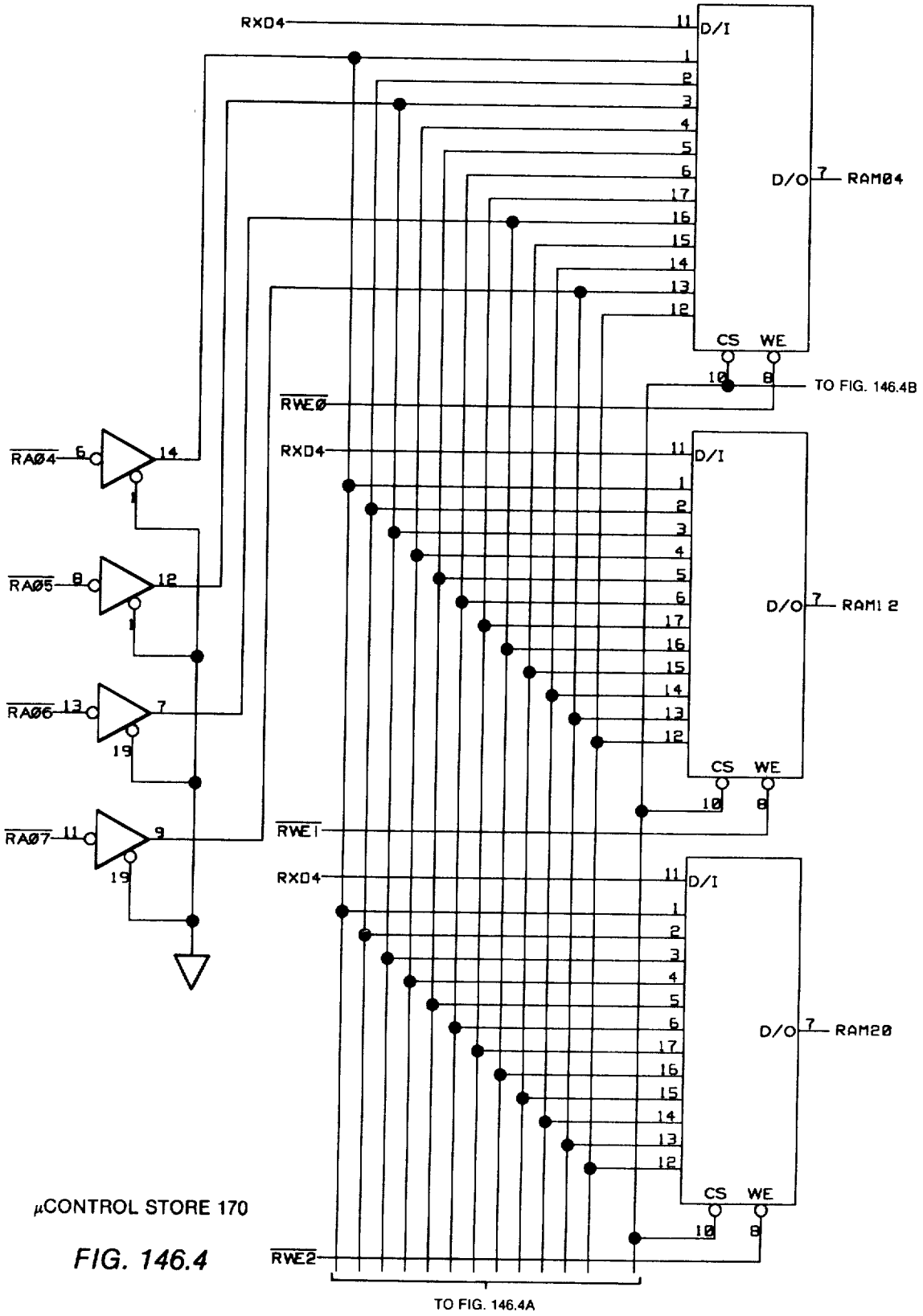

CS BUS EN
19
1
RAM24
RAM25
RAM26
RAM27
RAM28
RAM29
RAM30
RAM31
2
4
6
8
17
15
13
11
18
16
14
12
3
5
7
9
B9
B11
B13
B15
A65
A63
A61
A59
BREGS0
BREGS1
BREGS2
BREGS3
CSM0
CSM1
CSM2
CSM3
RRE3
19
1
2
4
6
8
17
15
13
11
18
16
14
12
3
5
7
9
RAMB0
RAMB1
RAMB2
RAMB3
RAMB4
RAMB5
RAMB6
RAMB7

FIG. 146.3F



μCONTROL STORE 170

FIG. 146.4

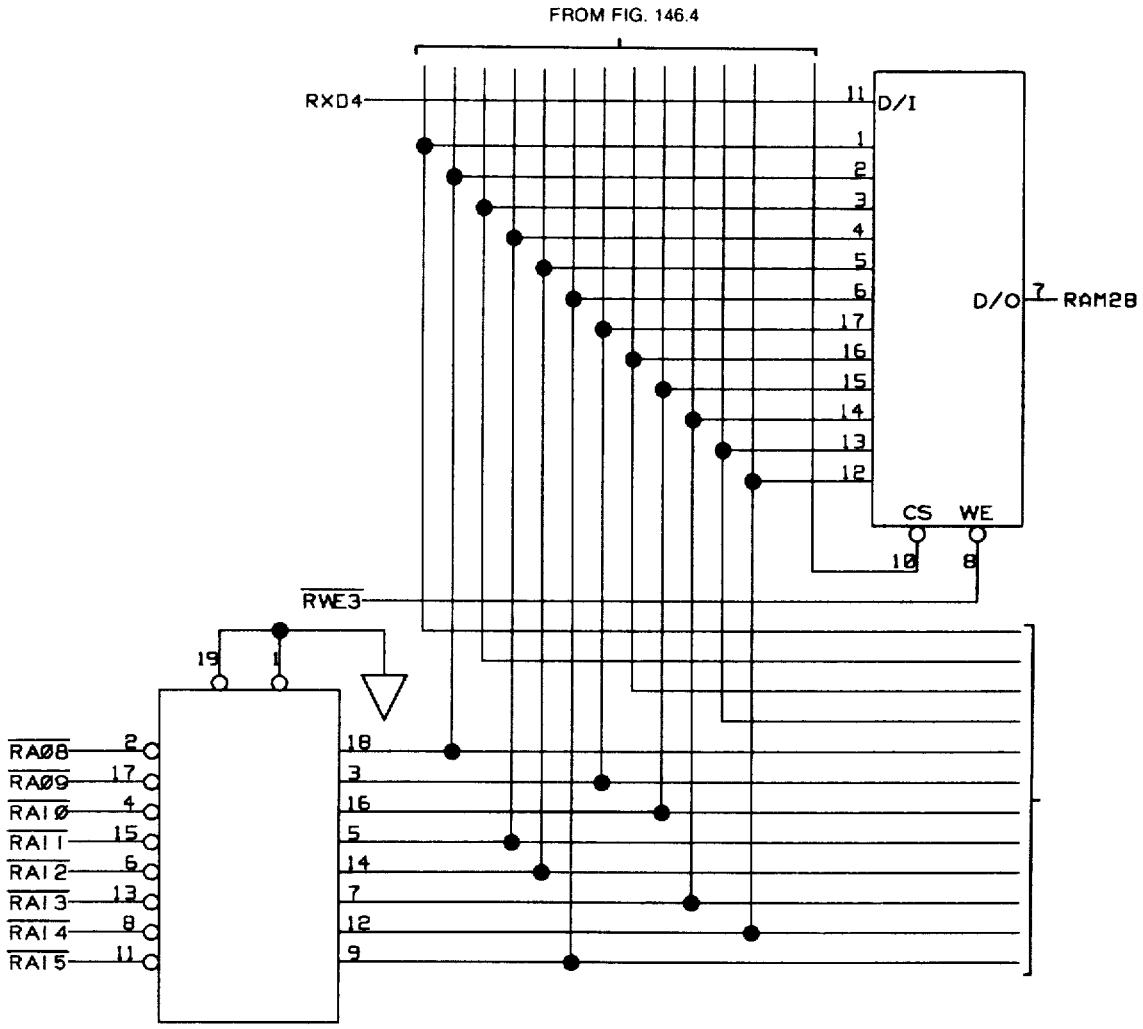

FROM FIG. 146.4
RXD4
D/I
D/O
RAM2B
CS
WE
RWE3
RAØ8
RAØ9
RAIØ
RAI1
RAI2
RAI3
RAI4
RAI5

FIG. 146.4A

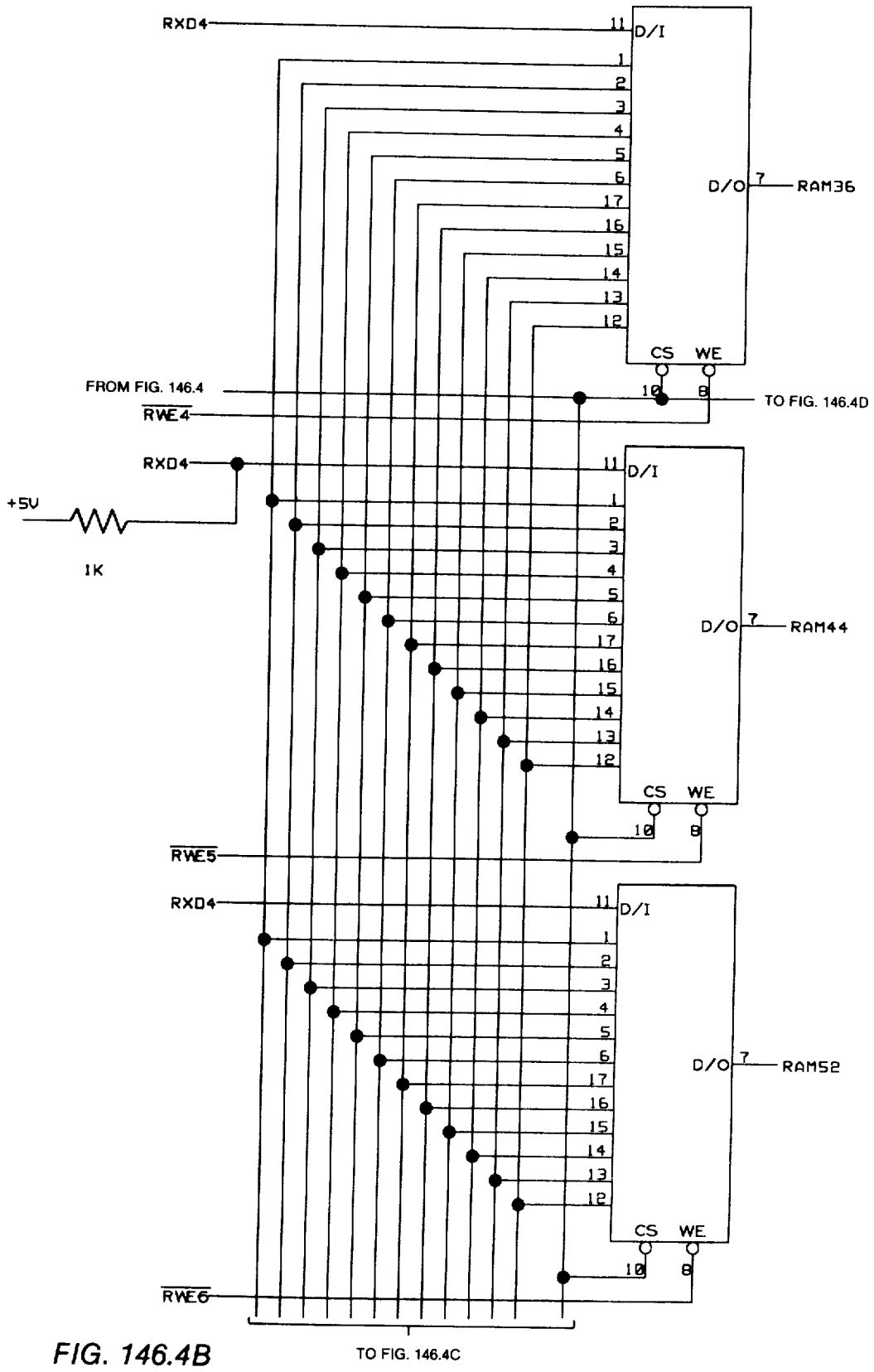

RXD4
D/I
D/O
RAM36
CS
WE
FROM FIG. 146.4
RWE4
TO FIG. 146.4D
RXD4
+5V
1K
D/I
D/O
RAM44
CS
WE
RWE5
RXD4
D/I
D/O
RAM52
CS
WE
RWE6
TO FIG. 146.4C

FIG. 146.4B

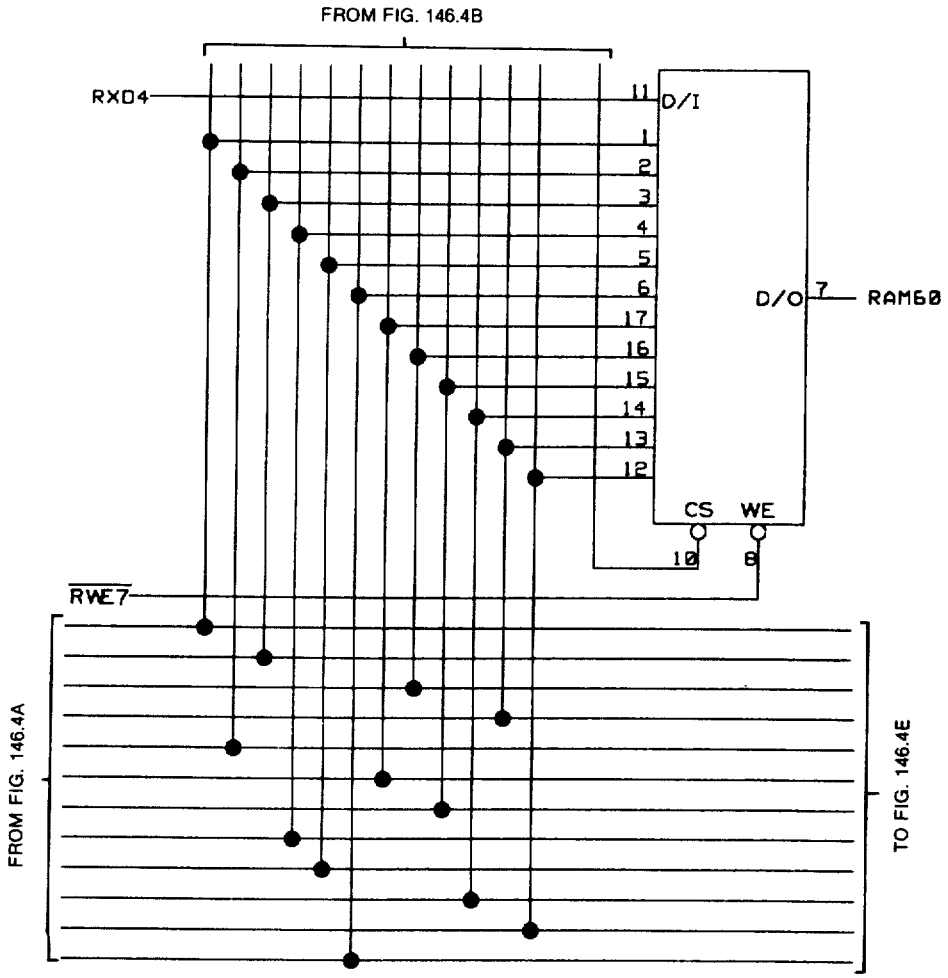

FROM FIG. 146.4B
RXD4
11
D/I
1
2
3
4
5
6
17
16
15
14
13
12
D/O
7
RAM60
CS
WE
10
8
RWE7
FROM FIG. 146.4A
TO FIG. 146.4E

FIG. 146.4C

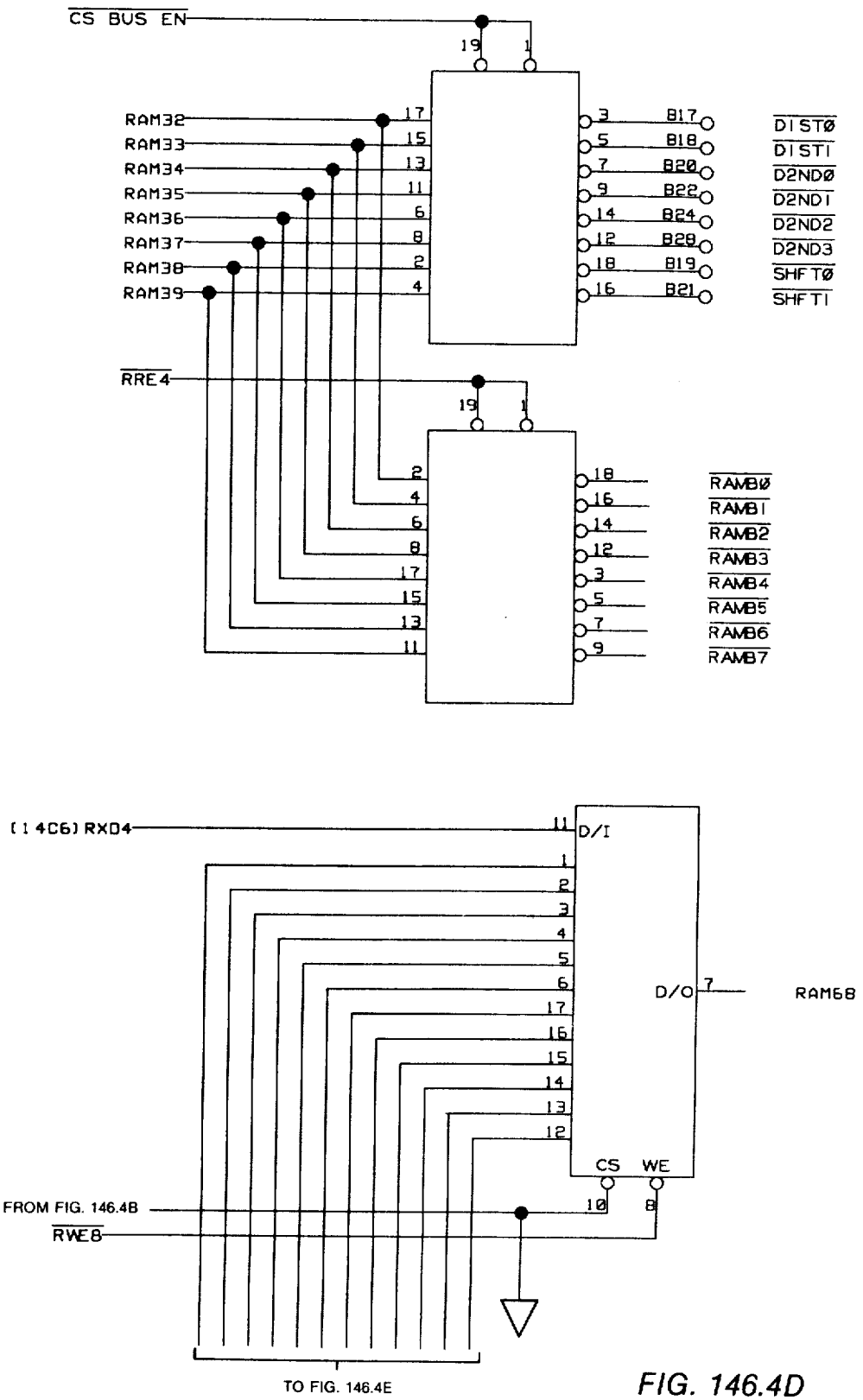

CS BUS EN
RAM32
RAM33
RAM34
RAM35
RAM36
RAM37
RAM38
RAM39
RRE4
B17
B18
B20
B22
B24
B28
B19
B21
D1STØ
D1ST1
D2NDØ
D2ND1
D2ND2
D2ND3
SHFTØ
SHFT1
RAMBØ
RAMB1
RAMB2
RAMB3
RAMB4
RAMB5
RAMB6
RAMB7
(1 4C6) RXD4
D/I
D/O
RAM68
CS
WE
FROM FIG. 146.4B
RWE8
TO FIG. 146.4E

FIG. 146.4D

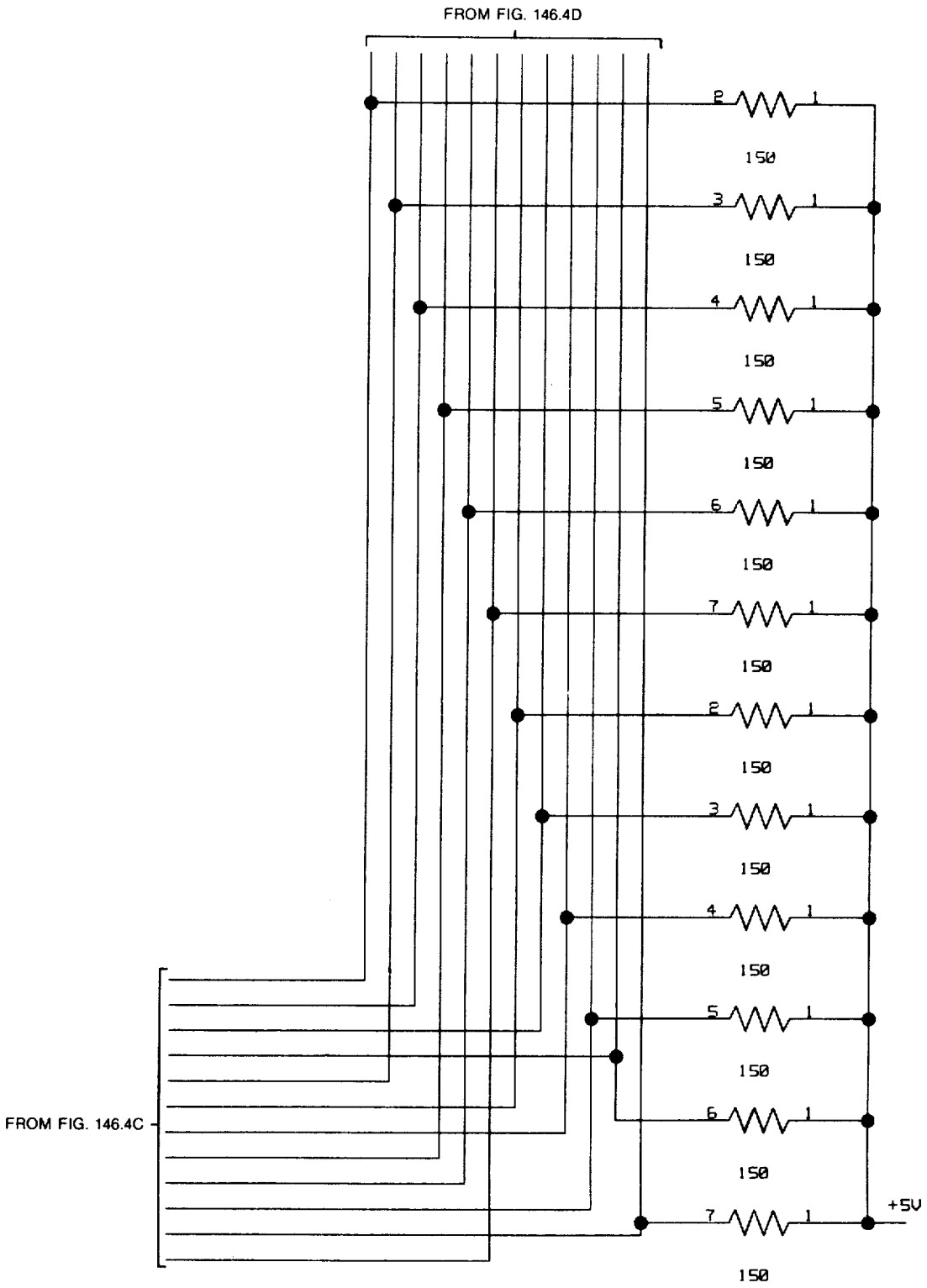
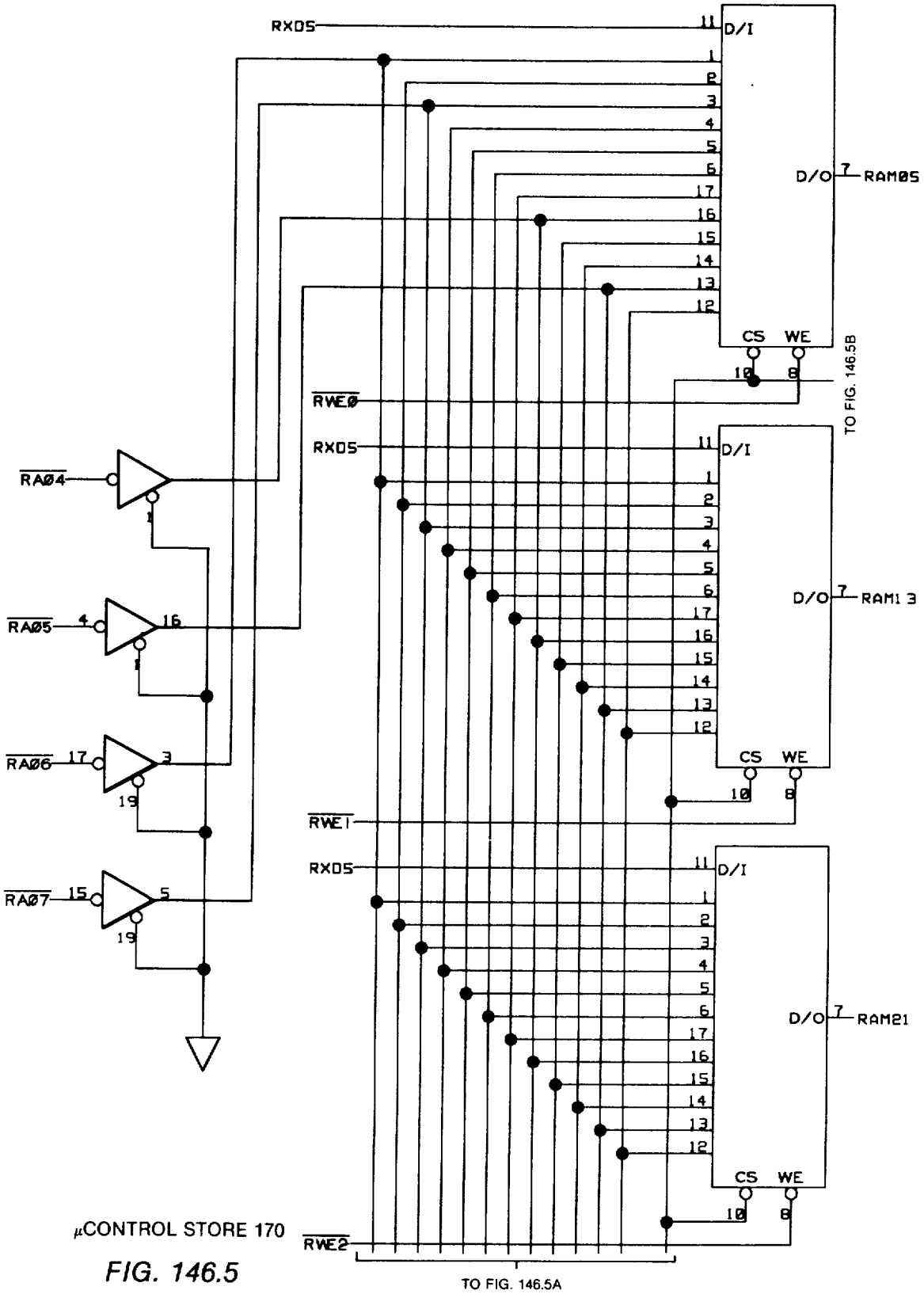


FIG. 146.4E



μCONTROL STORE 170

FIG. 146.5

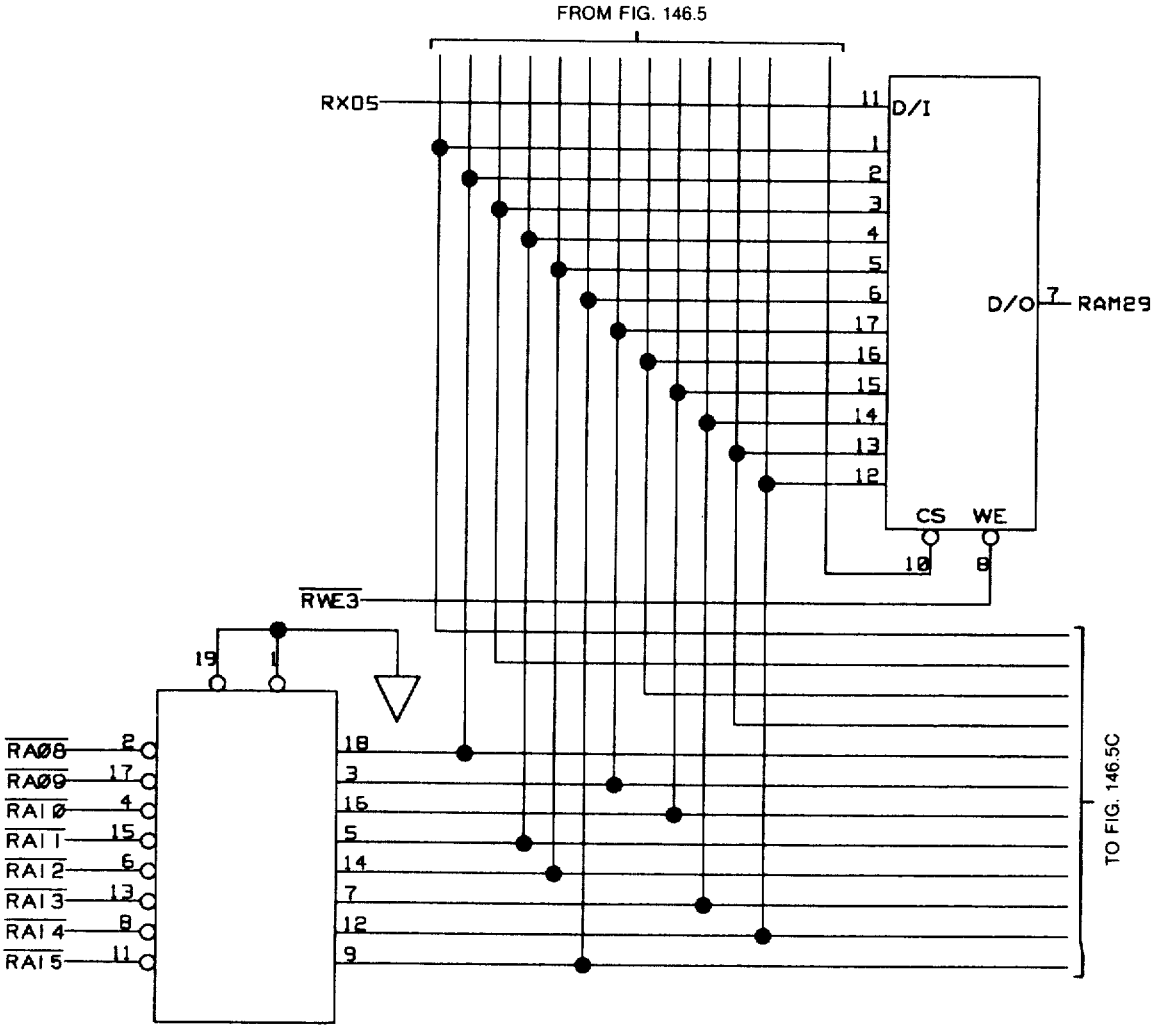

FROM FIG. 146.5
RX05
D/I
D/O
RAM29
CS
WE
RWE3
RA08
RA09
RAI0
RAI1
RAI2
RAI3
RAI4
RAI5
TO FIG. 146.5C

FIG. 146.5A

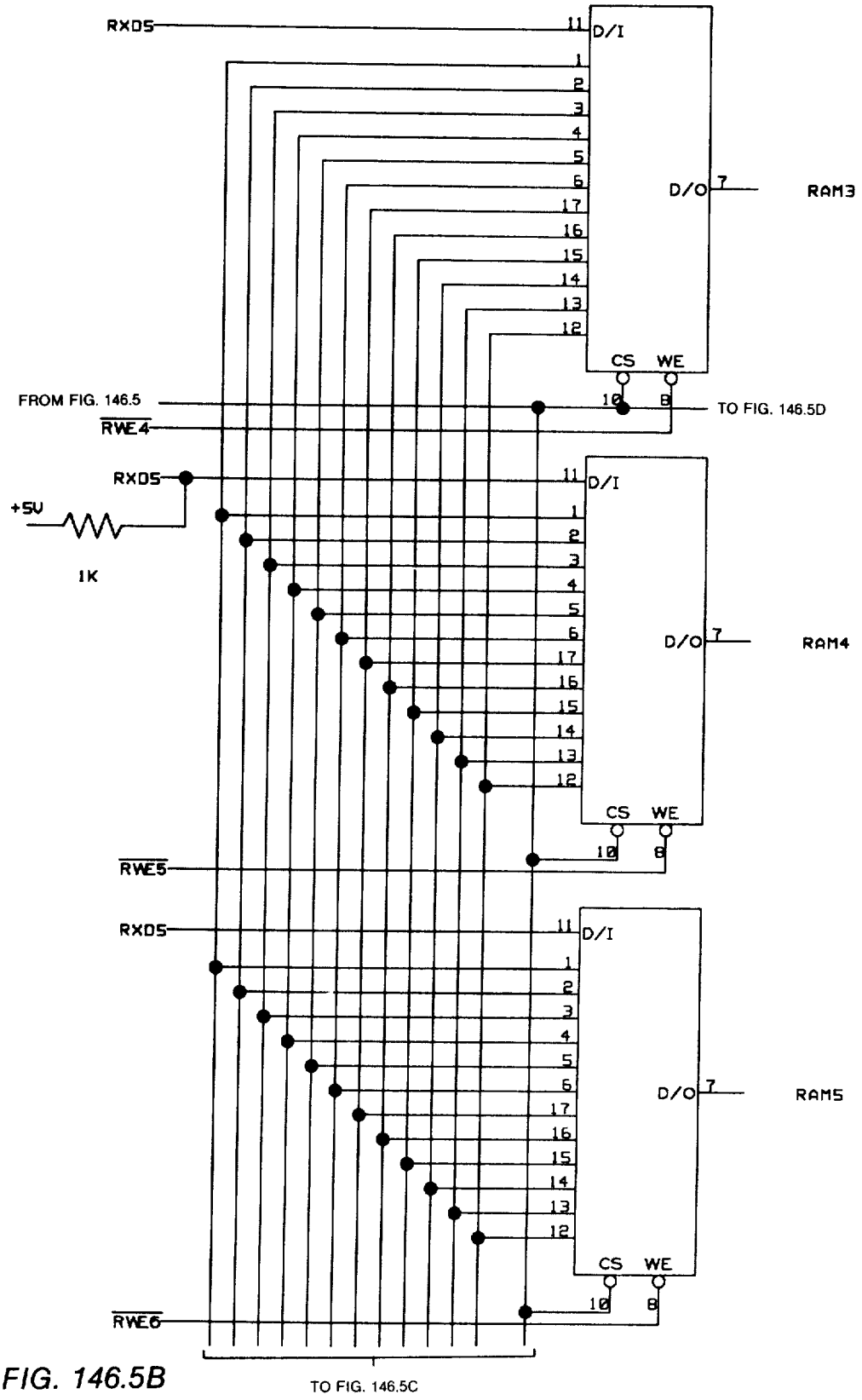

RXD5
D/I
D/O
RAM3
CS
WE
FROM FIG. 146.5
TO FIG. 146.5D
RWE4
RXD5
+5V
1K
D/I
D/O
RAM4
CS
WE
RWE5
RXD5
D/I
D/O
RAM5
CS
WE
RWE6
TO FIG. 146.5C

FIG. 146.5B

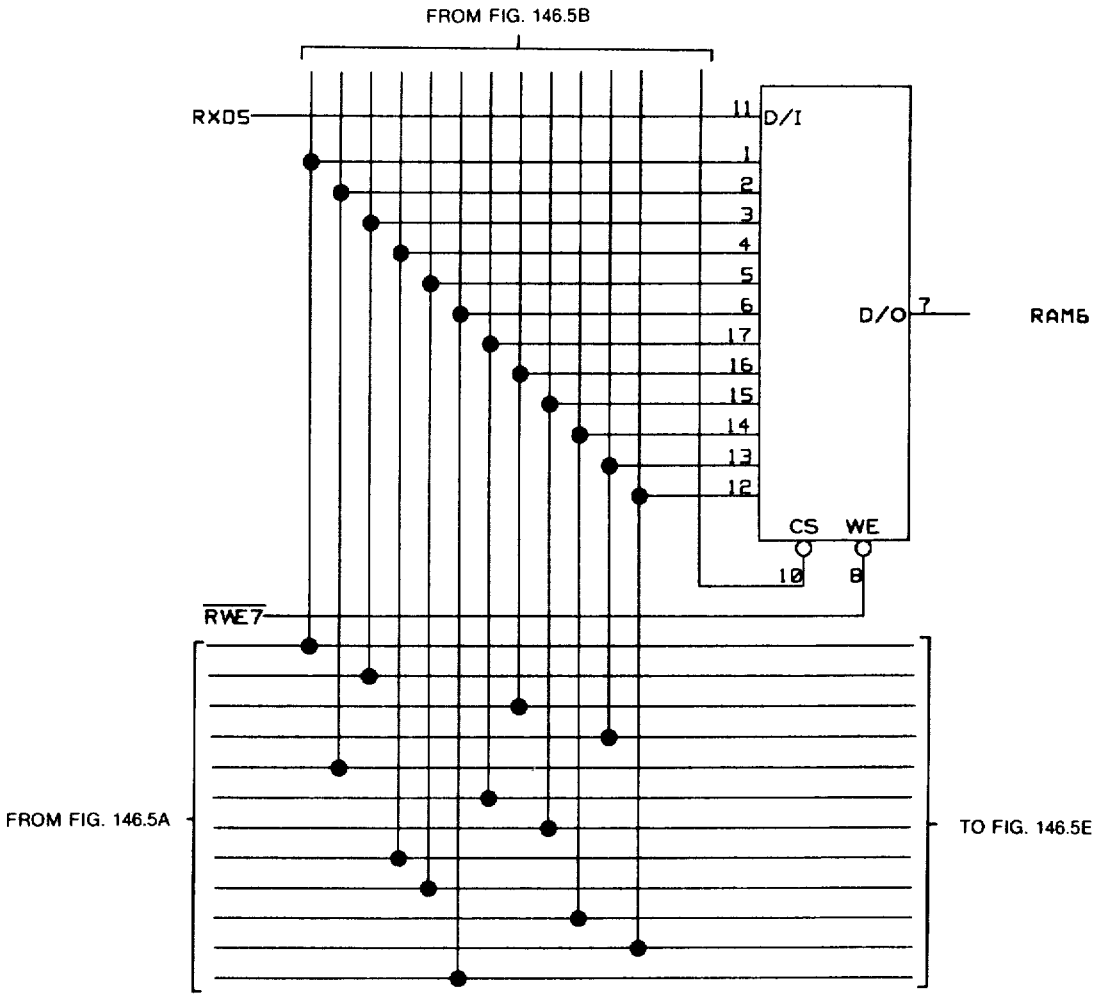

FROM FIG. 146.5B
RXD5
11
D/I
1
2
3
4
5
6
17
16
15
14
13
12
D/O
7
RAM6
CS
WE
10
8
RWE7
FROM FIG. 146.5A
TO FIG. 146.5E

FIG. 146.5C

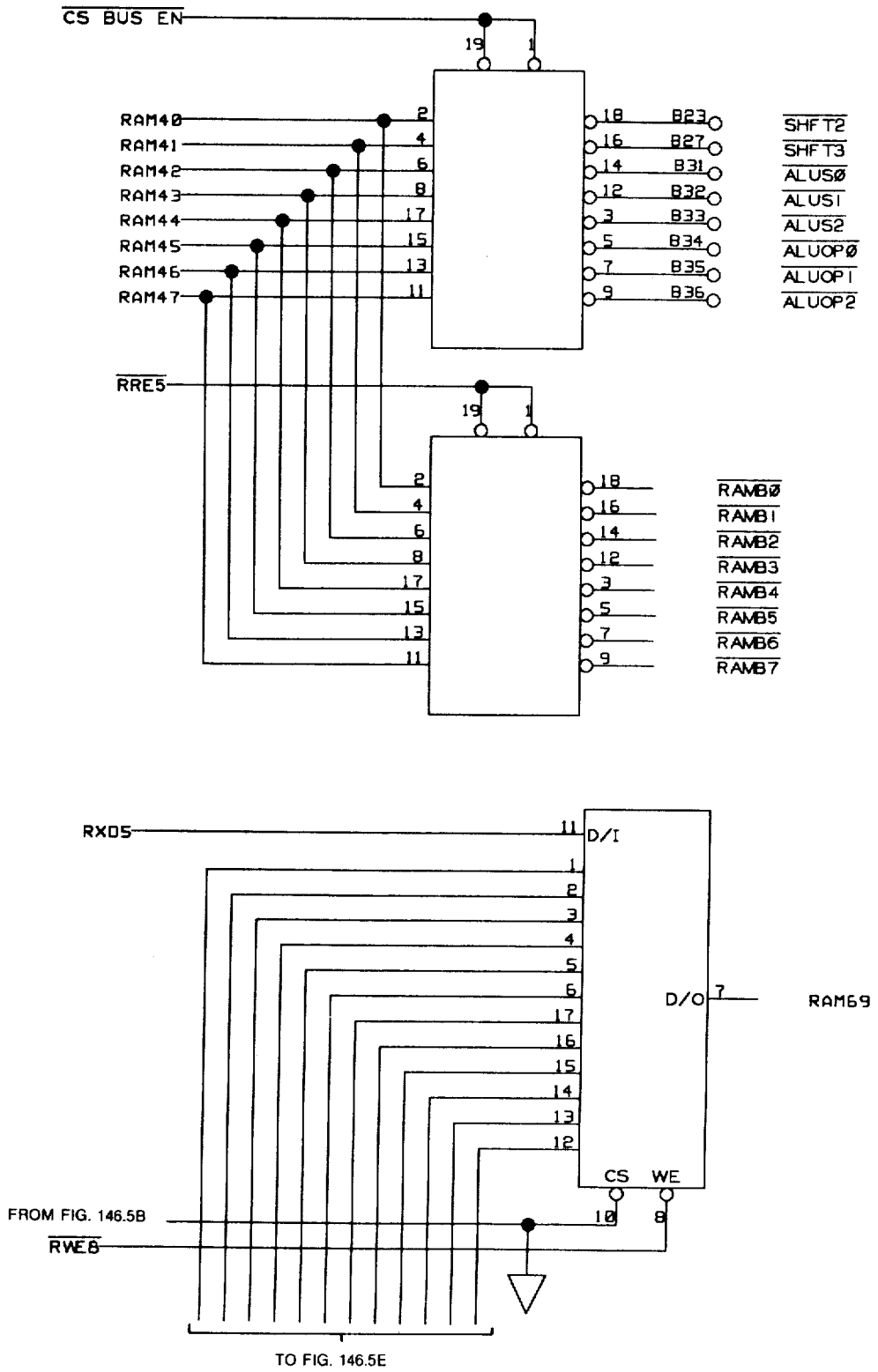

CS BUS EN
RAM40
RAM41
RAM42
RAM43
RAM44
RAM45
RAM46
RAM47
B23
B27
B31
B32
B33
B34
B35
B36
SHFT2
SHFT3
ALUS0
ALUS1
ALUS2
ALUOP0
ALUOP1
ALUOP2
RRE5
RAMB0
RAMB1
RAMB2
RAMB3
RAMB4
RAMB5
RAMB6
RAMB7
RXD5
D/I
D/O
RAM69
CS
WE
FROM FIG. 146.5B
RWE8
TO FIG. 146.5E

FIG. 146.5D

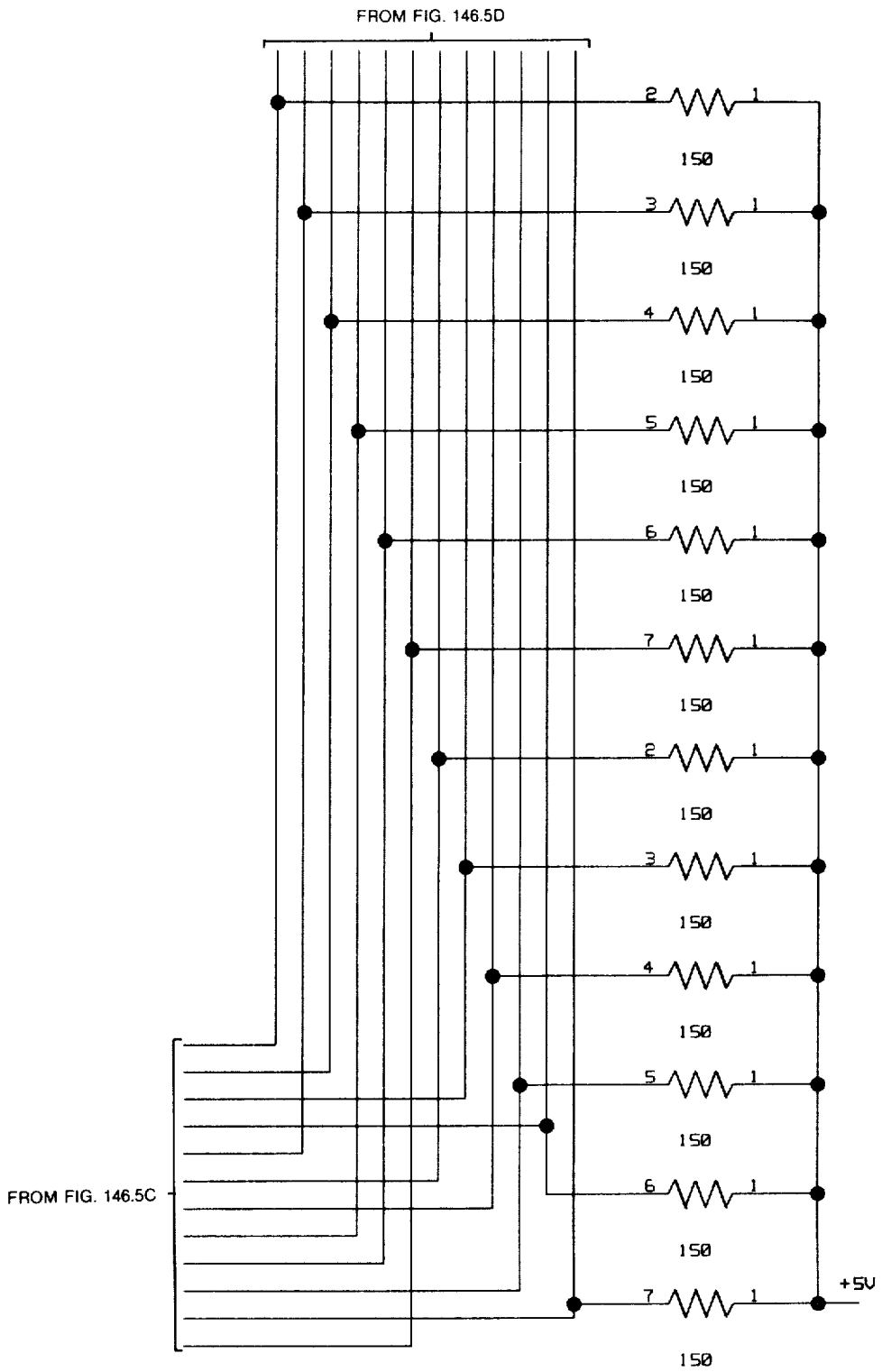
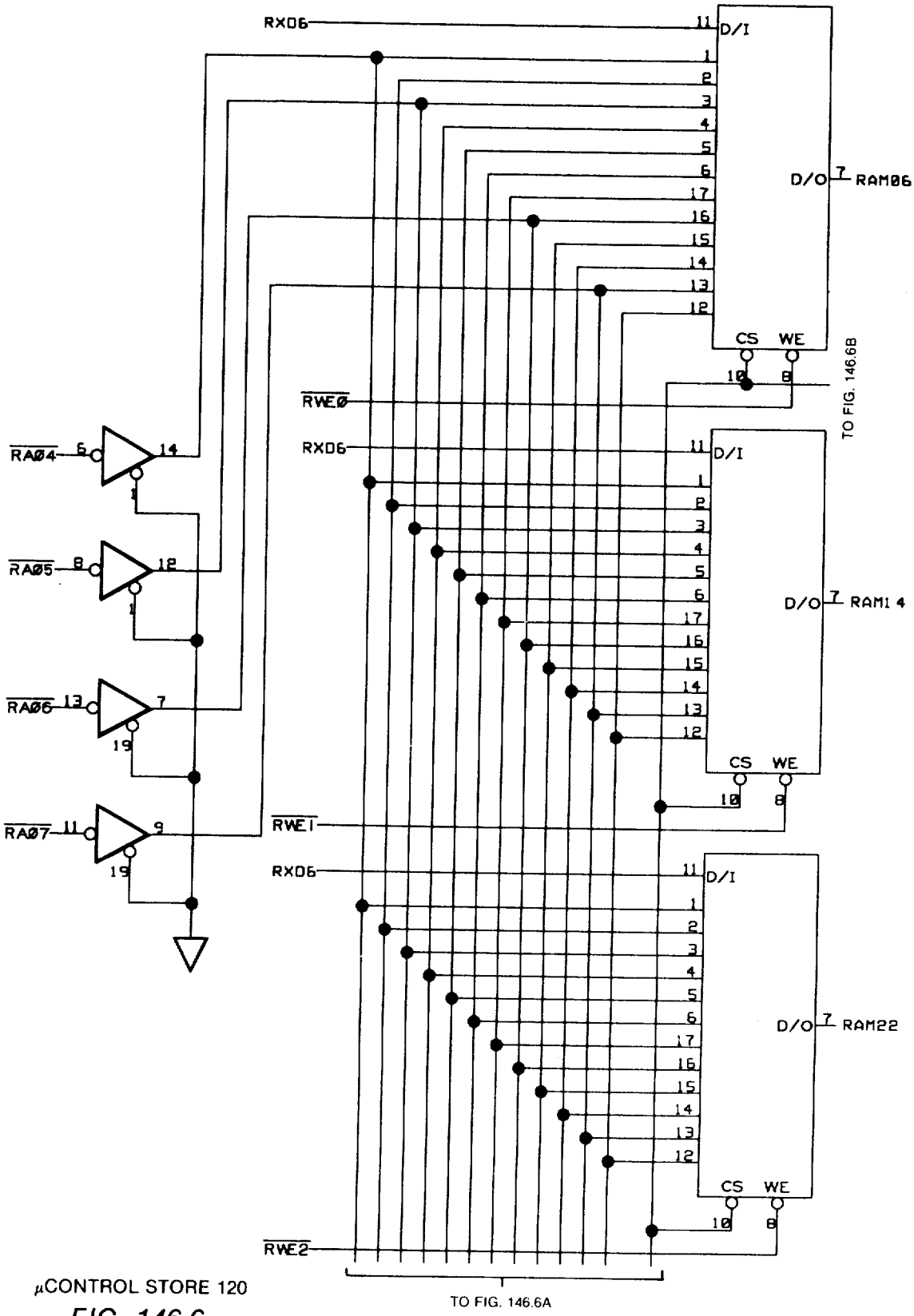


FIG. 146.5E



μCONTROL STORE 120

FIG. 146.6

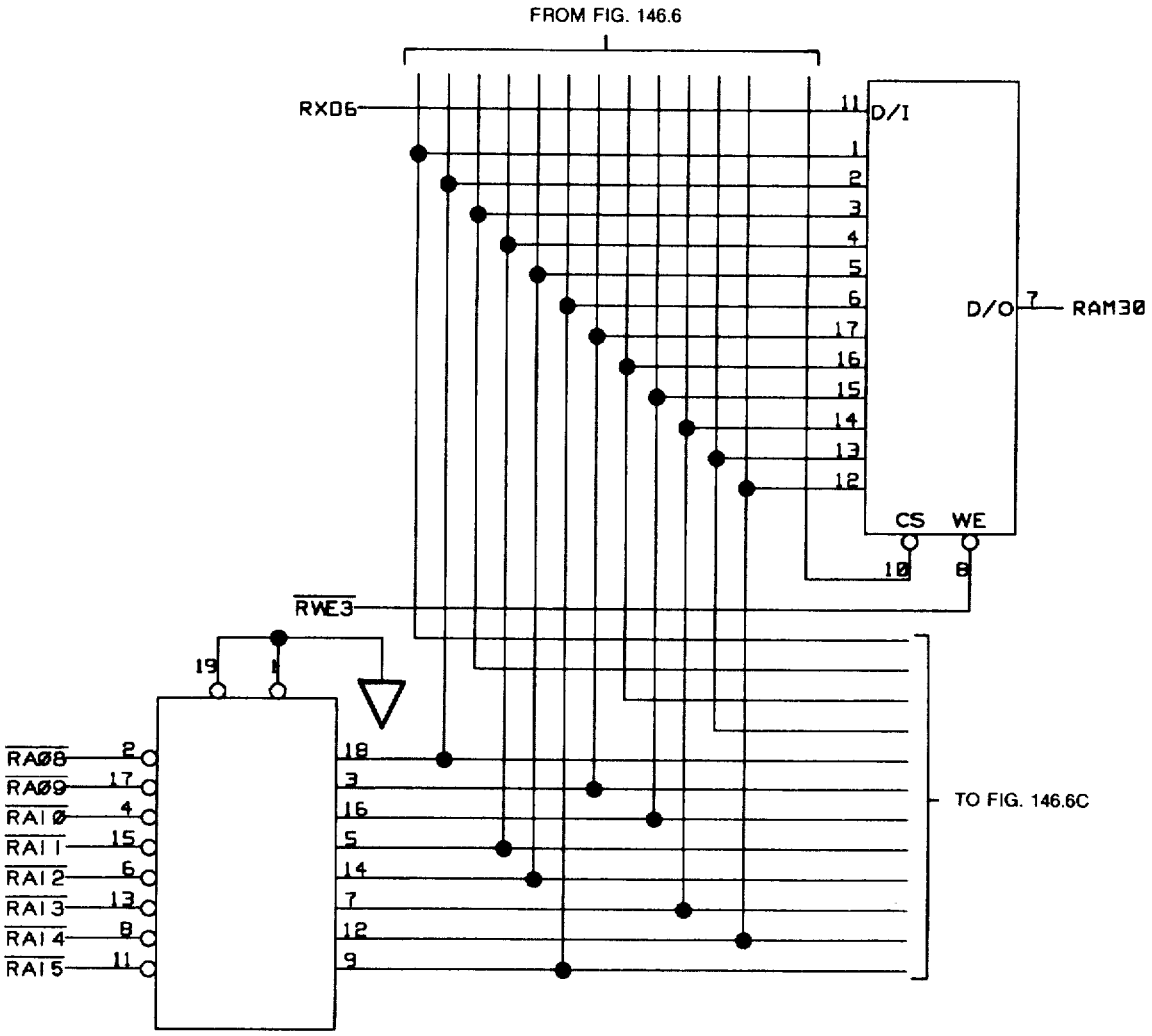

FROM FIG. 146.6
RXD6
D/I
D/O
RAM30
CS
WE
RWE3
RAØ8
RAØ9
RAI Ø
RAI 1
RAI 2
RAI 3
RAI 4
RAI 5
TO FIG. 146.6C

FIG. 146.6A

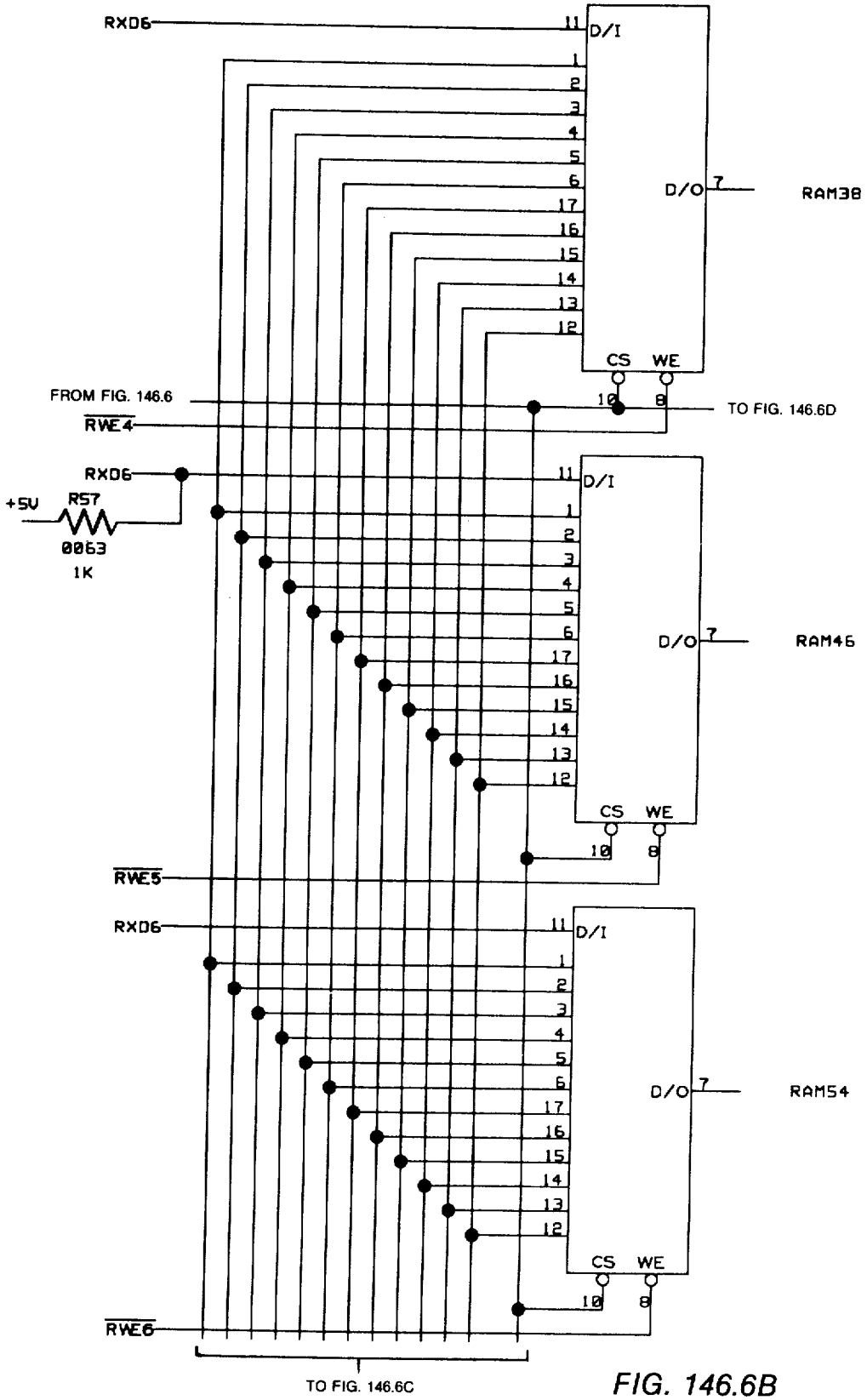

RXD6
D/I
D/O
RAM38
CS
WE
FROM FIG. 146.6
RWE4
TO FIG. 146.6D
RXD6
+5V
R57
0063
1K
RAM46
RWE5
RXD6
RAM54
RWE6
TO FIG. 146.6C

FIG. 146.6B

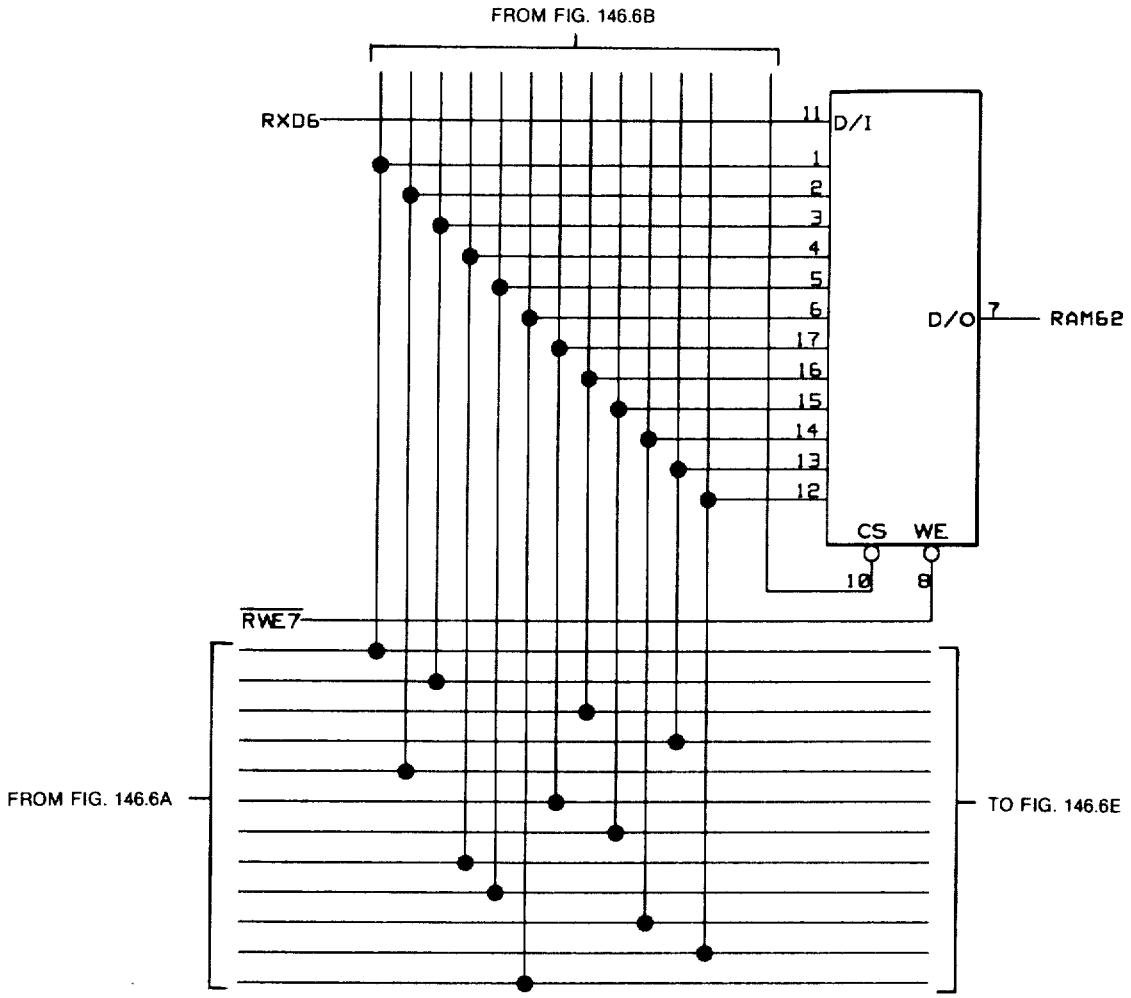

FROM FIG. 146.6B
RXD6
11
D/I
1
2
3
4
5
6
17
16
15
14
13
12
D/O
7
RAM62
CS
WE
10
8
RWE7
FROM FIG. 146.6A
TO FIG. 146.6E

FIG. 146.6C

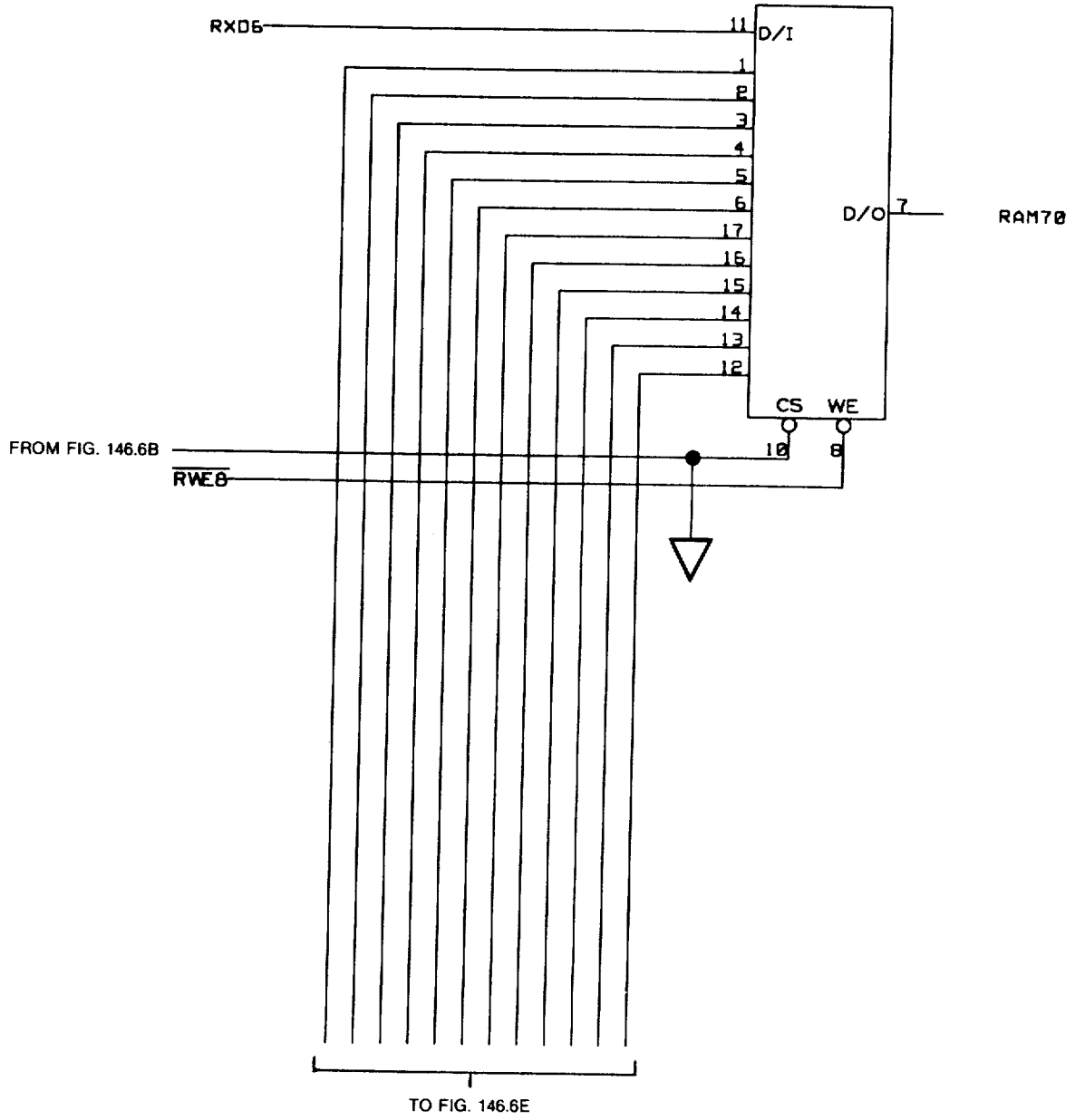

RXD6
11
D/I
1
2
3
4
5
6
17
16
15
14
13
12
D/O
7
RAM70
CS
WE
10
8
FROM FIG. 146.6B
RWE8
TO FIG. 146.6E

FIG. 146.6D

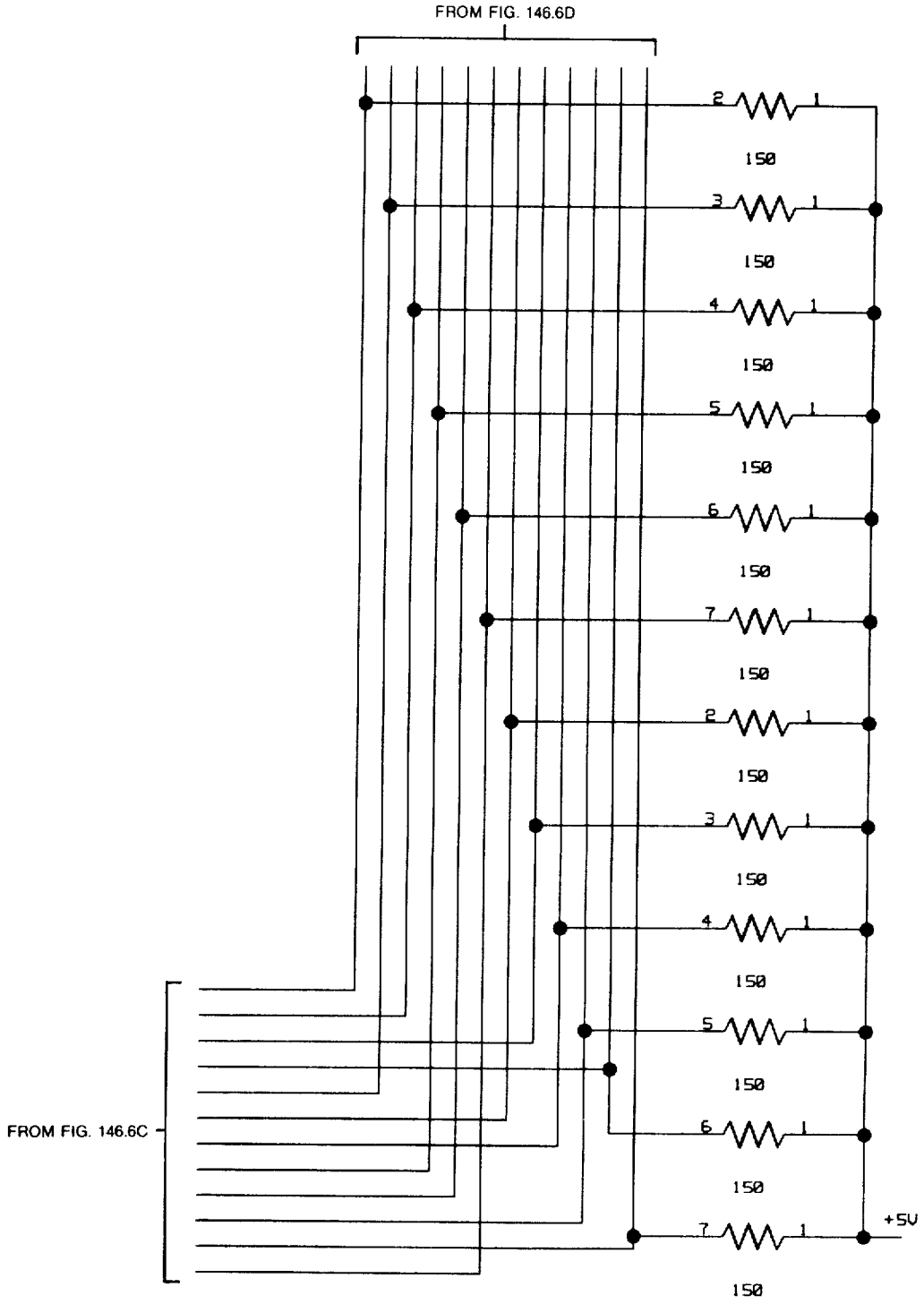


FIG. 146.6E

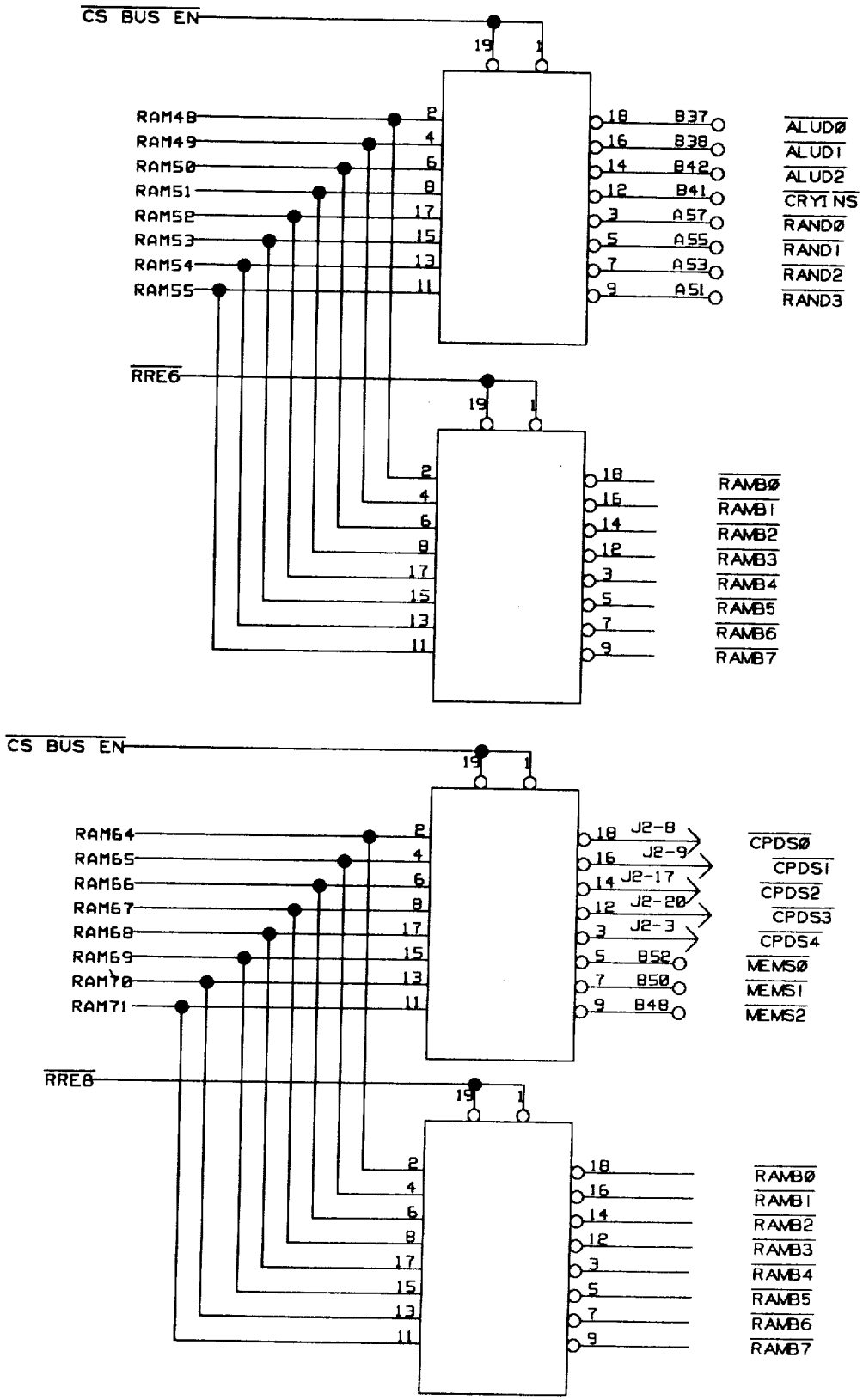
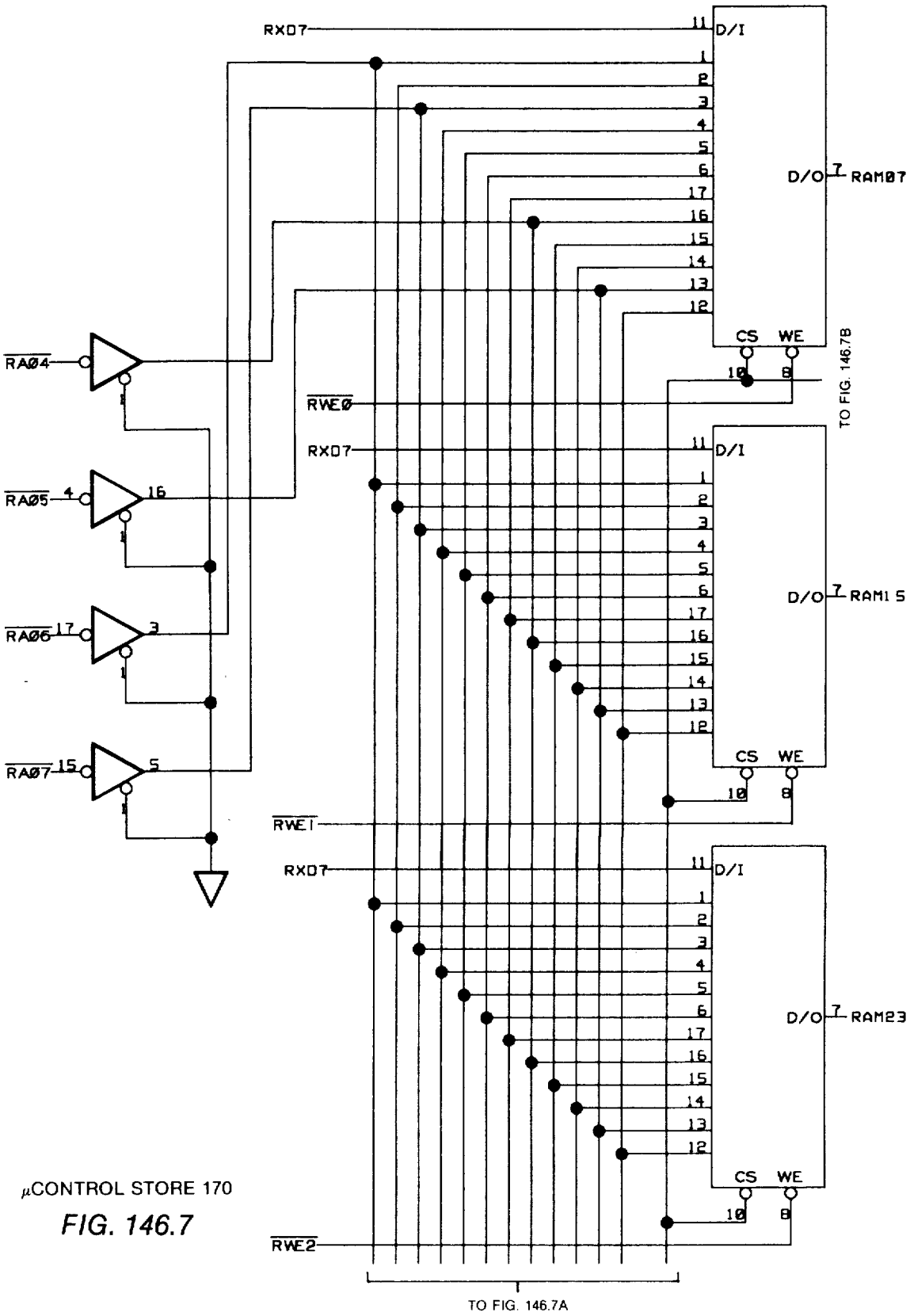

CS BUS EN
RAM48
RAM49
RAM50
RAM51
RAM52
RAM53
RAM54
RAM55
B37
B38
B42
B41
A57
A55
A53
A51
ALUD0
ALUD1
ALUD2
CRYI NS
RAND0
RAND1
RAND2
RAND3
RRE6
RAMB0
RAMB1
RAMB2
RAMB3
RAMB4
RAMB5
RAMB6
RAMB7
CS BUS EN
RAM64
RAM65
RAM66
RAM67
RAM68
RAM69
RAM70
RAM71
J2-8
J2-9
J2-17
J2-20
J2-3
B52
B50
B48
CPDS0
CPDS1
CPDS2
CPDS3
CPDS4
MEMS0
MEMS1
MEMS2
RRE8
RAMB0
RAMB1
RAMB2
RAMB3
RAMB4
RAMB5
RAMB6
RAMB7

FIG. 146.6F



μCONTROL STORE 170

FIG. 146.7

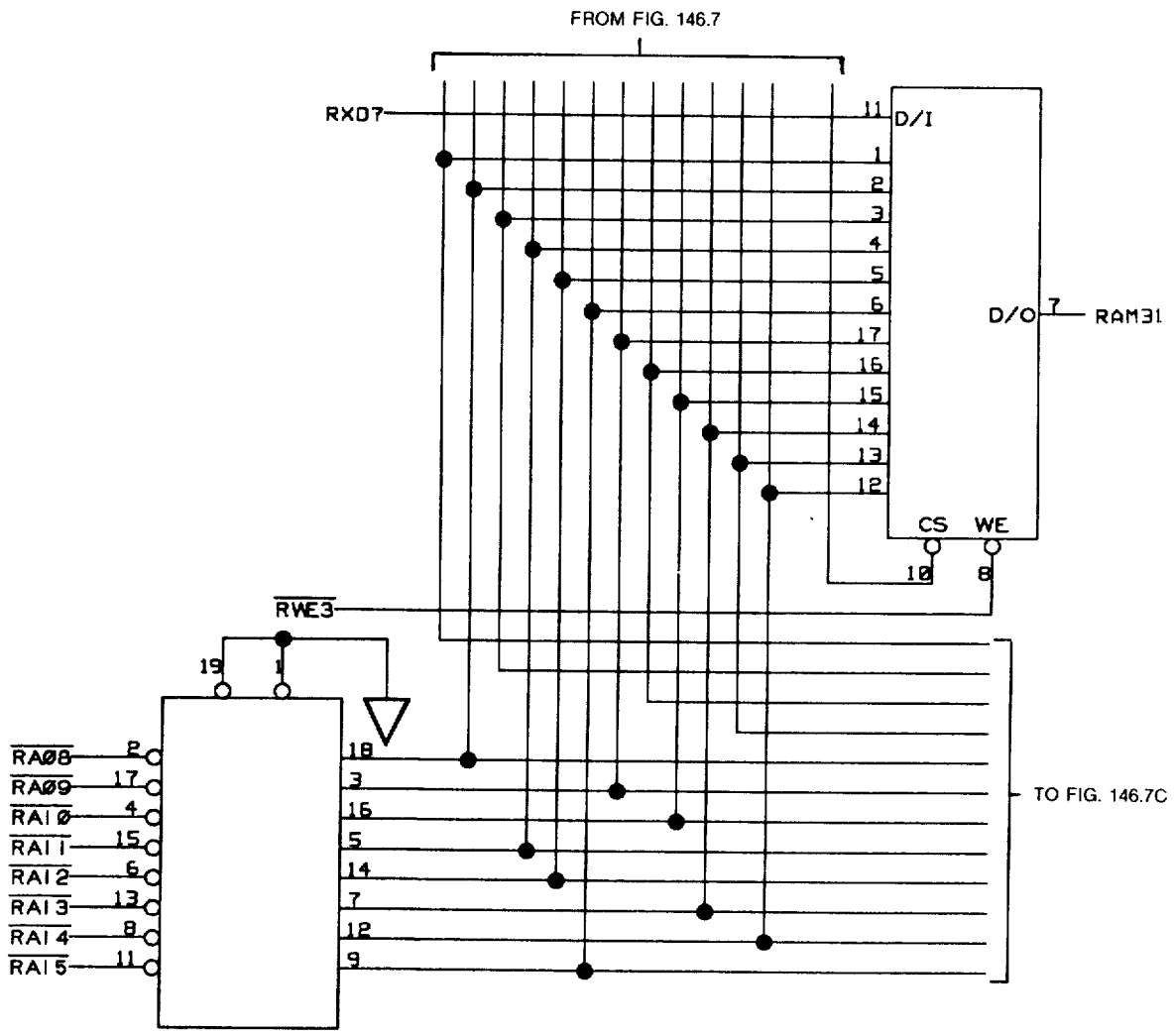

FROM FIG. 146.7
RXD7
D/I
D/O
RAM31
CS
WE
RWE3
RAØ8
RAØ9
RAI Ø
RAI 1
RAI 2
RAI 3
RAI 4
RAI 5
TO FIG. 146.7C

FIG. 146.7A

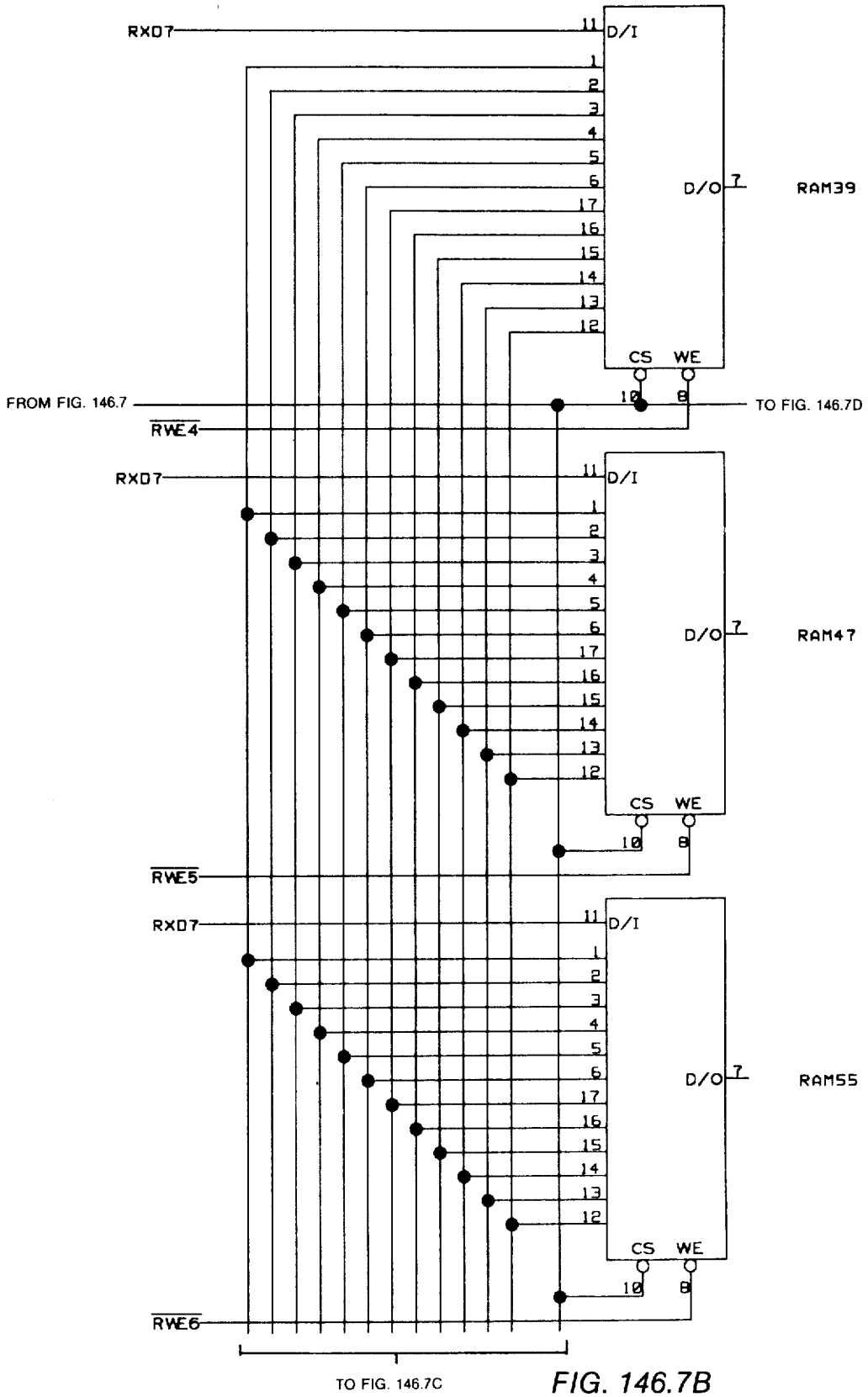

RXD7
D/I
D/O
RAM39
CS
WE
FROM FIG. 146.7
TO FIG. 146.7D
RWE4
RXD7
D/I
D/O
RAM47
CS
WE
RWE5
RXD7
D/I
D/O
RAM55
CS
WE
RWE6
TO FIG. 146.7C

FIG. 146.7B

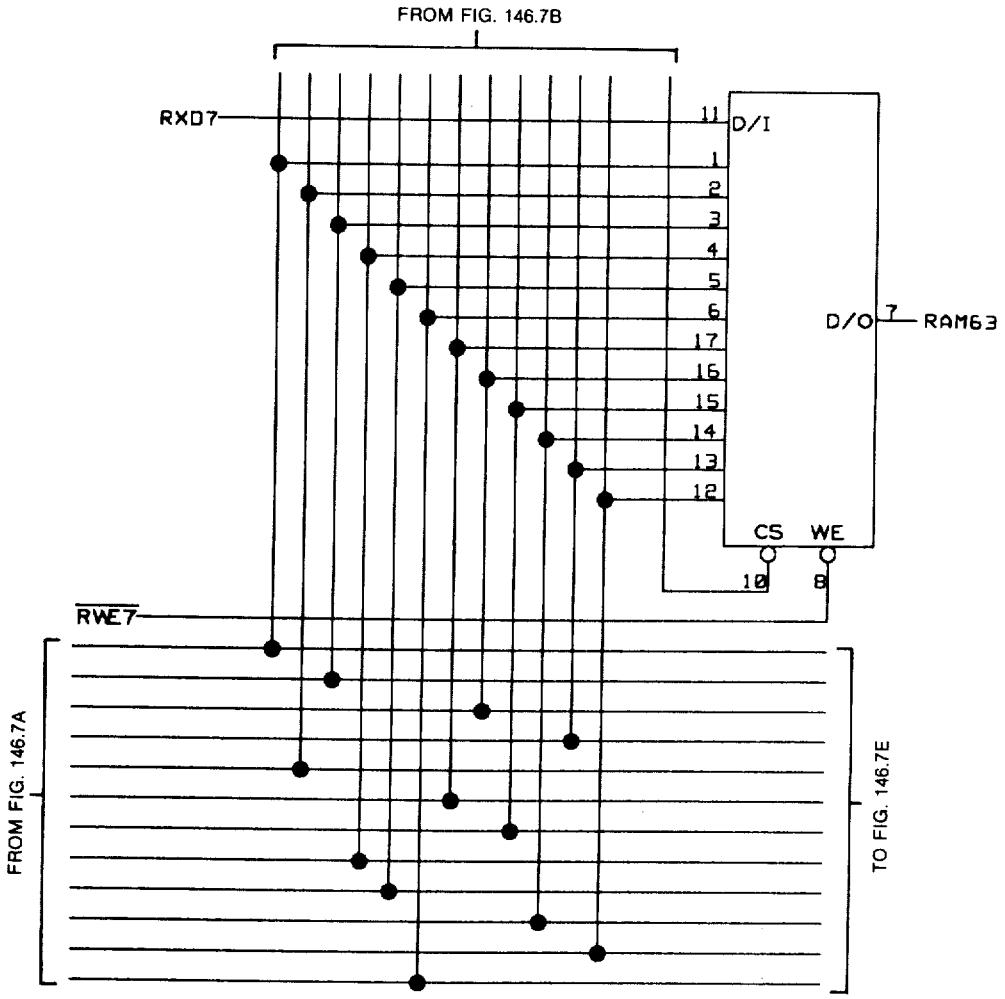

FROM FIG. 146.7B
RXD7
11
D/I
1
2
3
4
5
6
17
16
15
14
13
12
D/O
7
RAM63
CS
WE
10
8
RWE7
FROM FIG. 146.7A
TO FIG. 146.7E

FIG. 146.7C

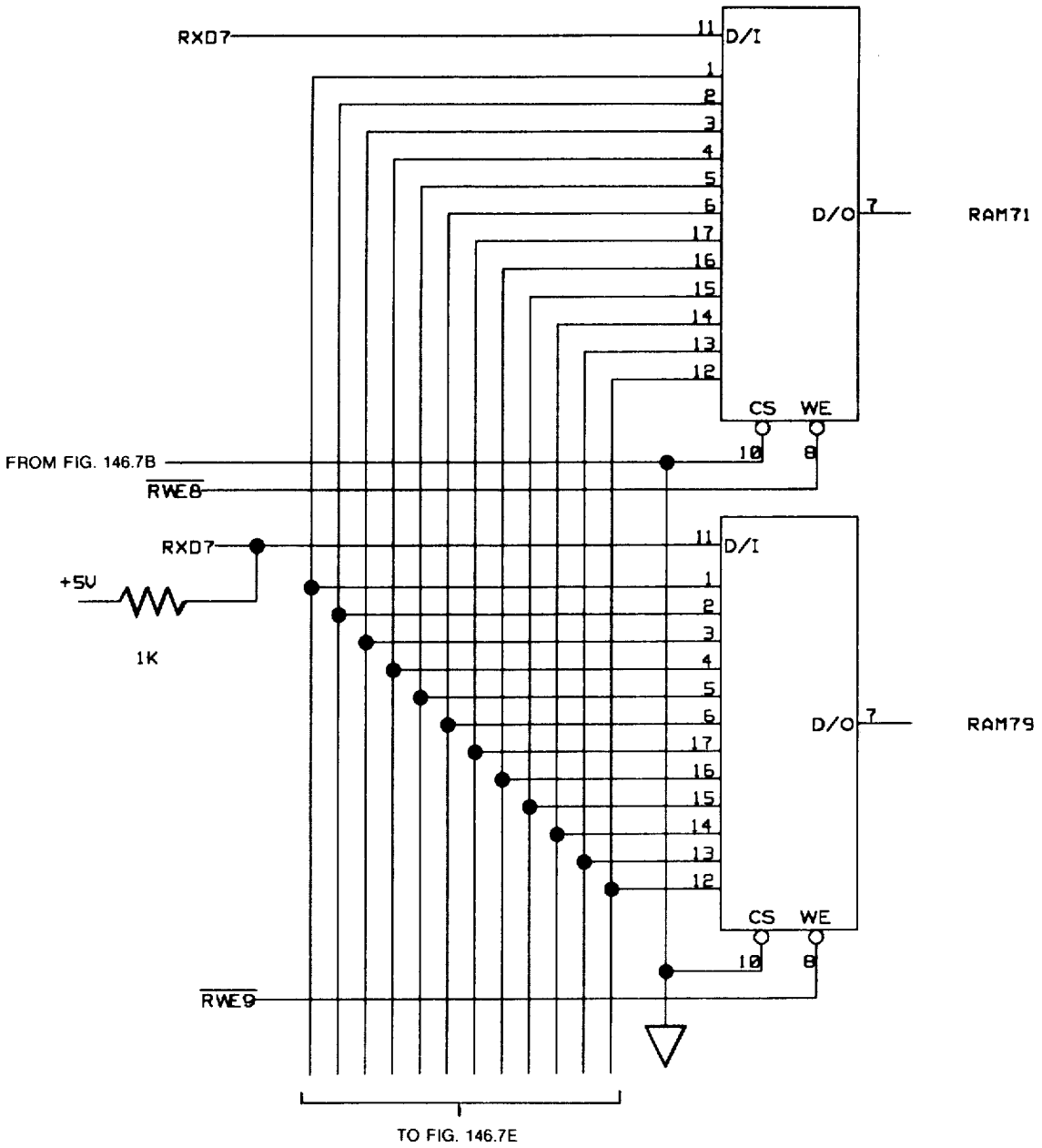

RXD7
D/I
D/O
RAM71
CS
WE
FROM FIG. 146.7B
RWE8
RXD7
+5V
1K
D/I
D/O
RAM79
CS
WE
RWE9
TO FIG. 146.7E

FIG. 146.7D

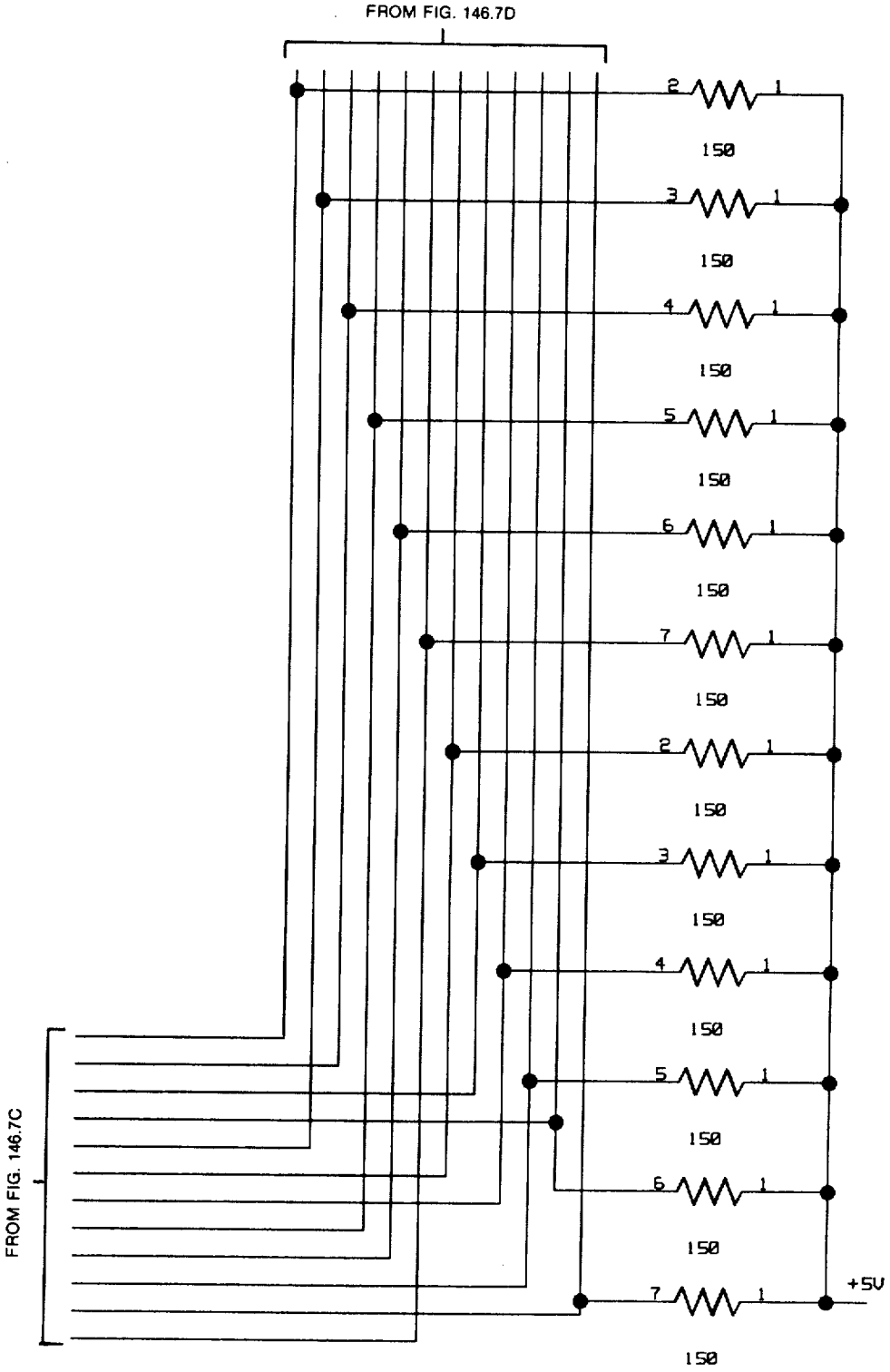


FIG. 146.7E

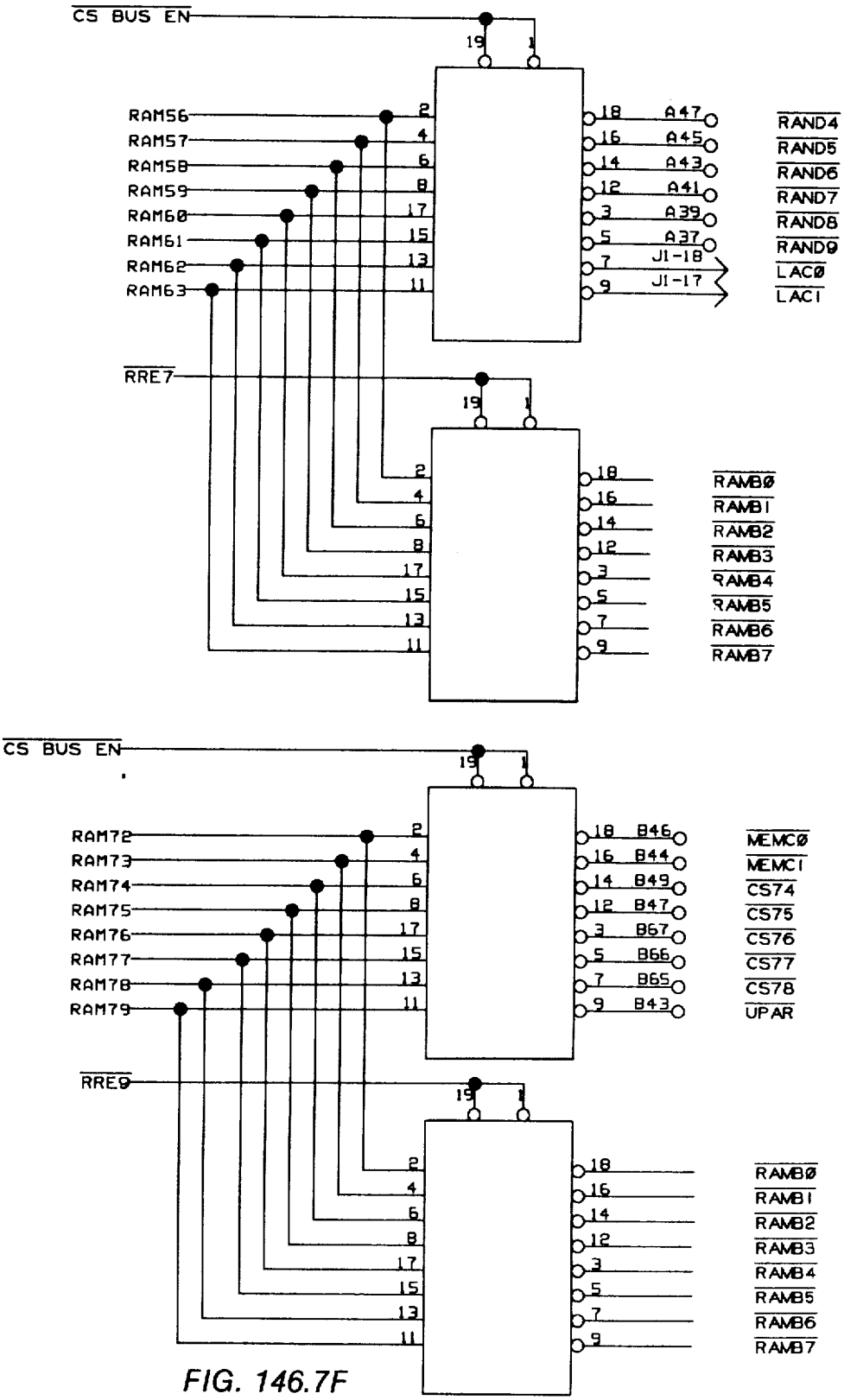
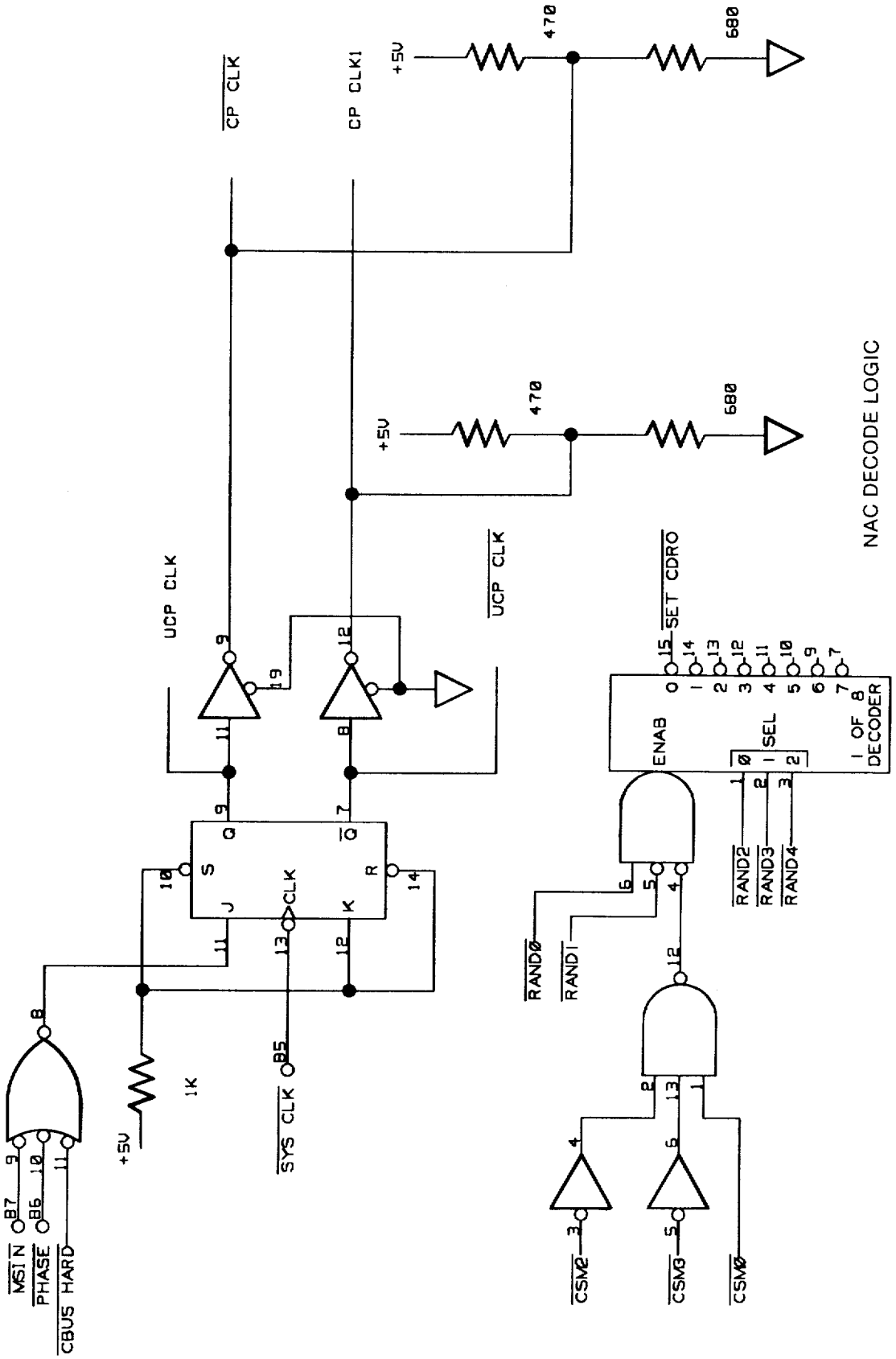

CS BUS EN
RAM56
RAM57
RAM58
RAM59
RAM60
RAM61
RAM62
RAM63
RRE7
A47
A45
A43
A41
A39
A37
J1-18
J1-17
RAND4
RAND5
RAND6
RAND7
RAND8
RAND9
LAC0
LAC1
RAMB0
RAMB1
RAMB2
RAMB3
RAMB4
RAMB5
RAMB6
RAMB7
CS BUS EN
RAM72
RAM73
RAM74
RAM75
RAM76
RAM77
RAM78
RAM79
RRE9
B46
B44
B49
B47
B67
B66
B65
B43
MEMC0
MEMC1
CS74
CS75
CS76
CS77
CS78
UPAR
RAMB0
RAMB1
RAMB2
RAMB3
RAMB4
RAMB5
RAMB6
RAMB7

FIG. 146.7F



NAC DECODE LOGIC

FIG. 147

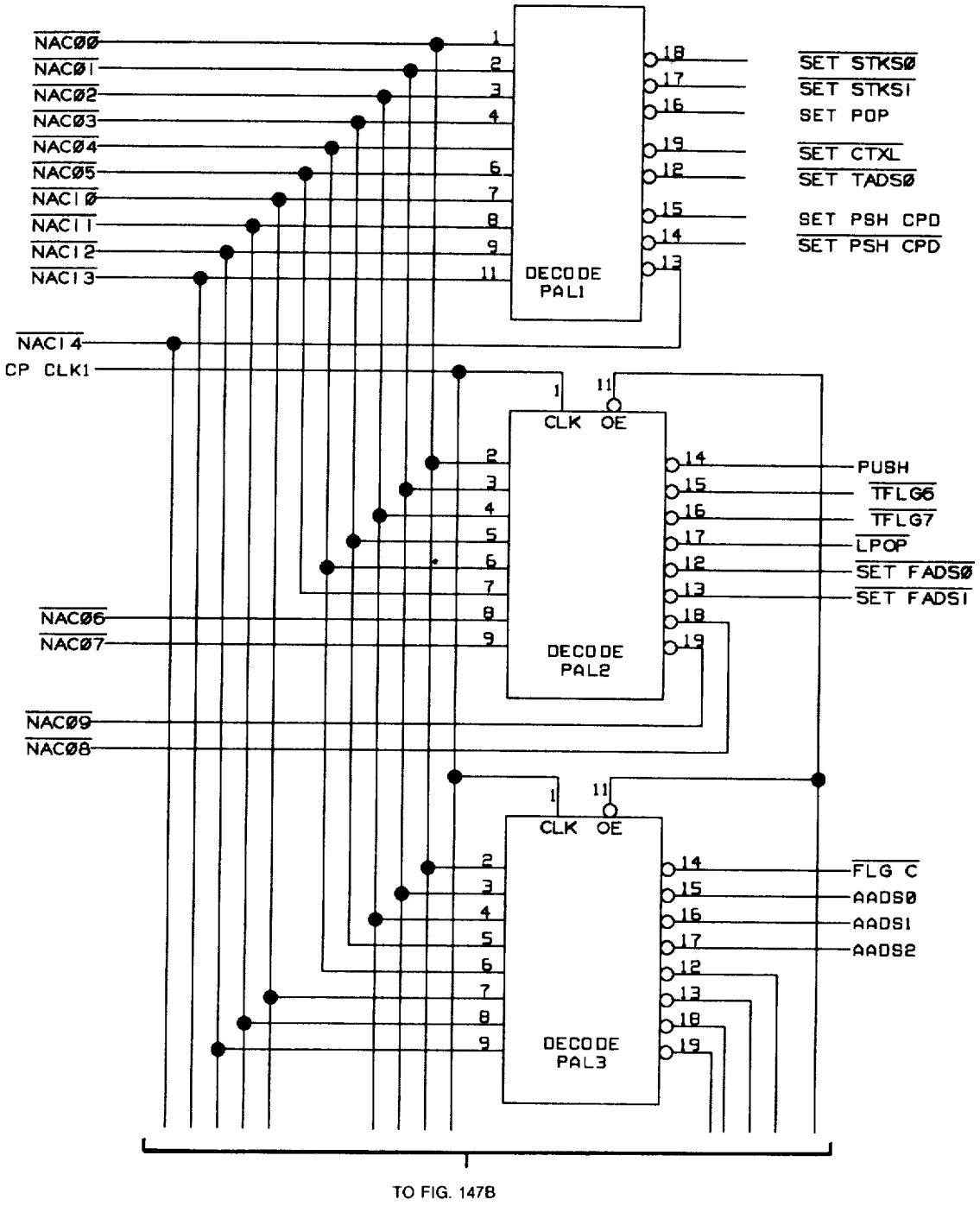

NAC00
NAC01
NAC02
NAC03
NAC04
NAC05
NAC10
NAC11
NAC12
NAC13
NAC14
CP CLK1
NAC06
NAC07
NAC09
NAC08
DECODE PAL1
SET STKS0
SET STKS1
SET POP
SET CTXL
SET TADS0
SET PSH CPD
SET PSH CPD
CLK OE
DECODE PAL2
PUSH
TFLG6
TFLG7
LPOP
SET FADS0
SET FADS1
CLK OE
DECODE PAL3
FLG C
AADS0
AADS1
AADS2
TO FIG. 147B

FIG. 147A

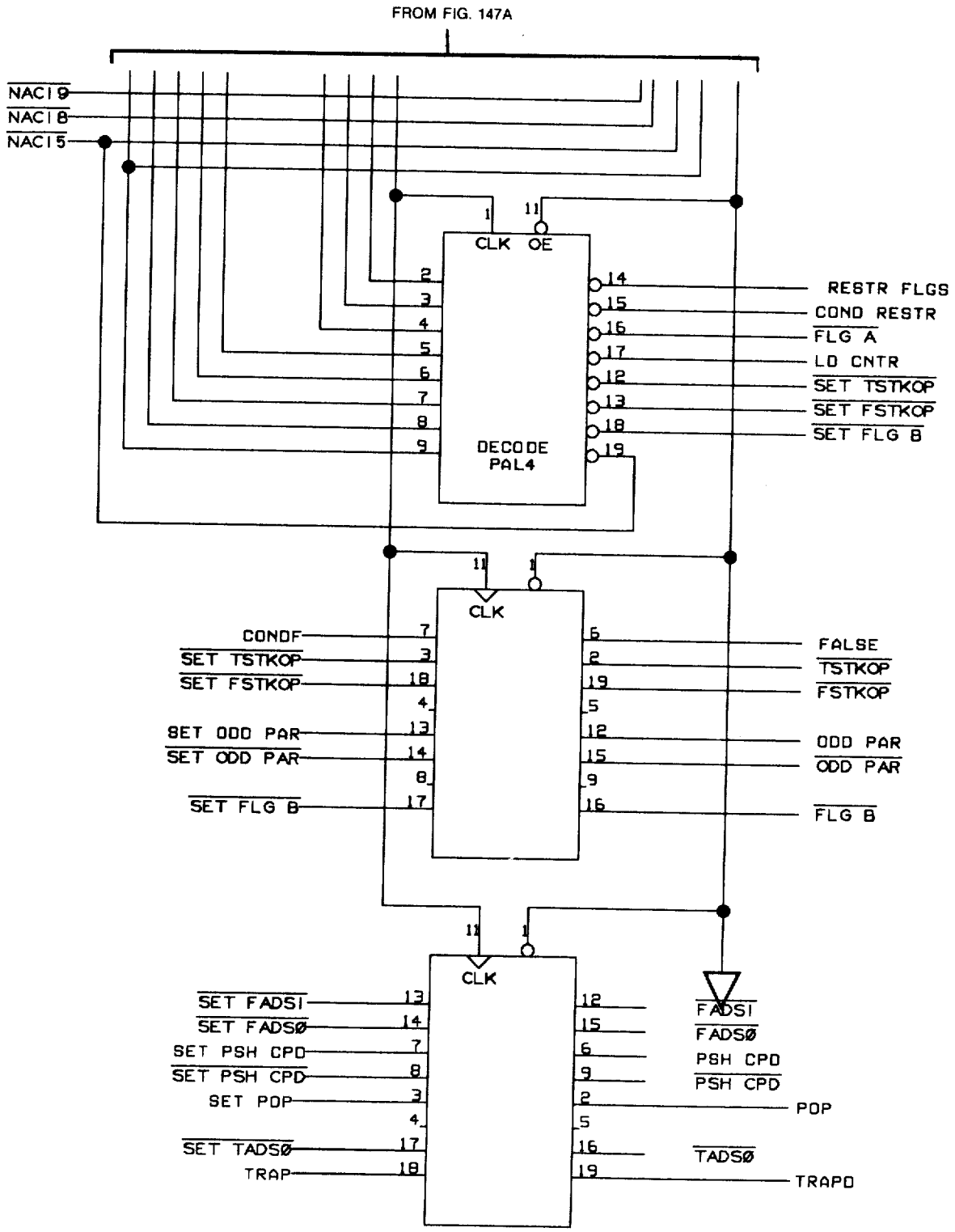

FROM FIG. 147A
NAC19
NAC18
NAC15
CLK
OE
DECODE
PAL4
RESTR FLGS
COND RESTR
FLG A
LD CNTR
SET TSTKOP
SET FSTKOP
SET FLG B
CLK
CONDF
SET TSTKOP
SET FSTKOP
SET ODD PAR
SET ODD PAR
SET FLG B
FALSE
TSTKOP
FSTKOP
ODD PAR
ODD PAR
FLG B
CLK
SET FADSI
SET FADSØ
SET PSH CPD
SET PSH CPD
SET POP
SET TADSØ
TRAP
FADSI
FADSØ
PSH CPD
PSH CPD
POP
TADSØ
TRAPD

FIG. 147B

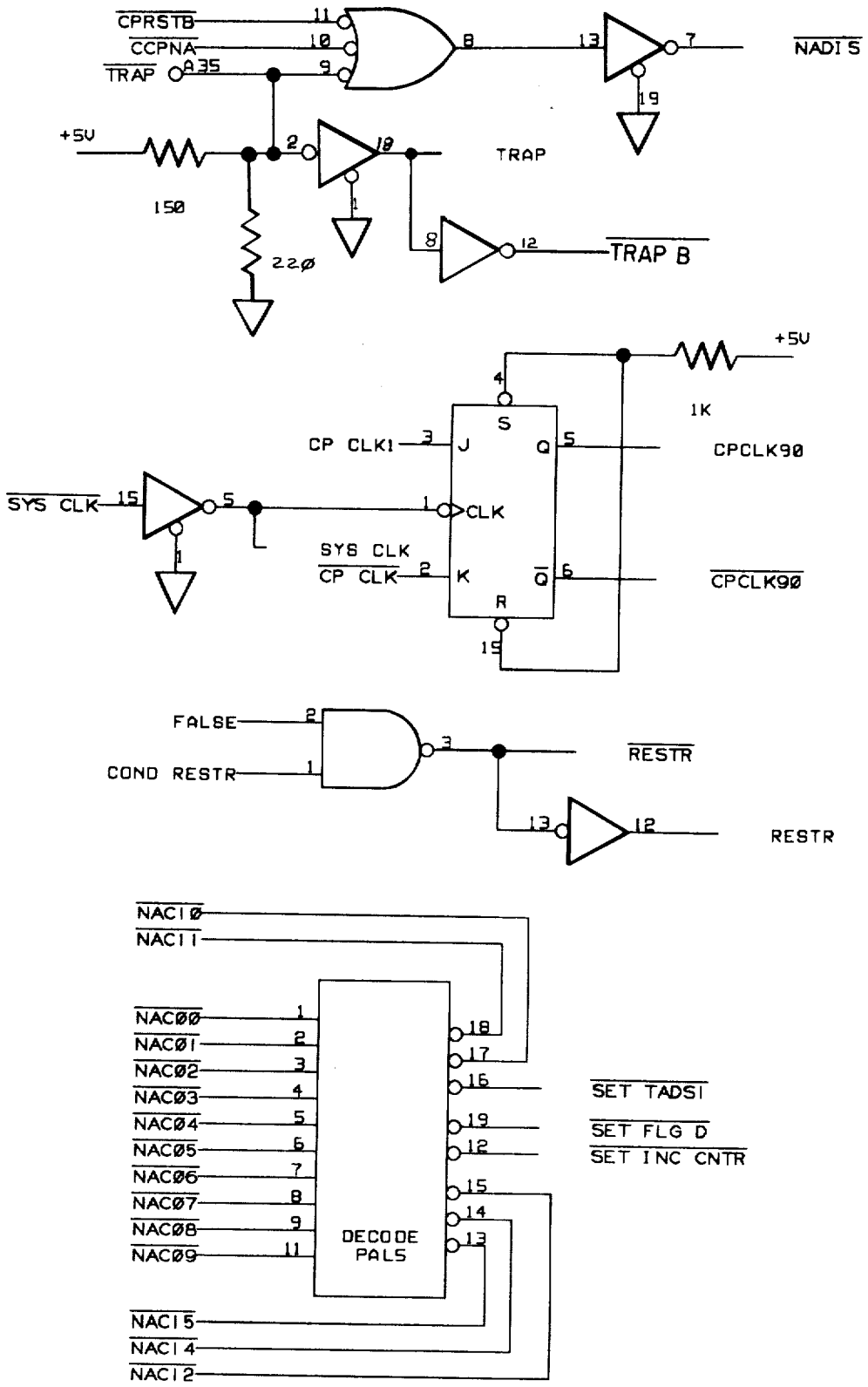

CPRSTB
CCPNA
TRAP
A35
NADI 5
+5V
150
220
TRAP
TRAP B
+5V
1K
CP CLK1
CPCLK90
SYS CLK
SYS CLK
CP CLK
CPCLK90
FALSE
COND RESTR
RESTR
RESTR
NACI0
NACI1
NAC00
NAC01
NAC02
NAC03
NAC04
NAC05
NAC06
NAC07
NAC08
NAC09
DECODE PALS
SET TADSI
SET FLG D
SET INC CNTR
NACI5
NACI4
NACI2

FIG. 147C

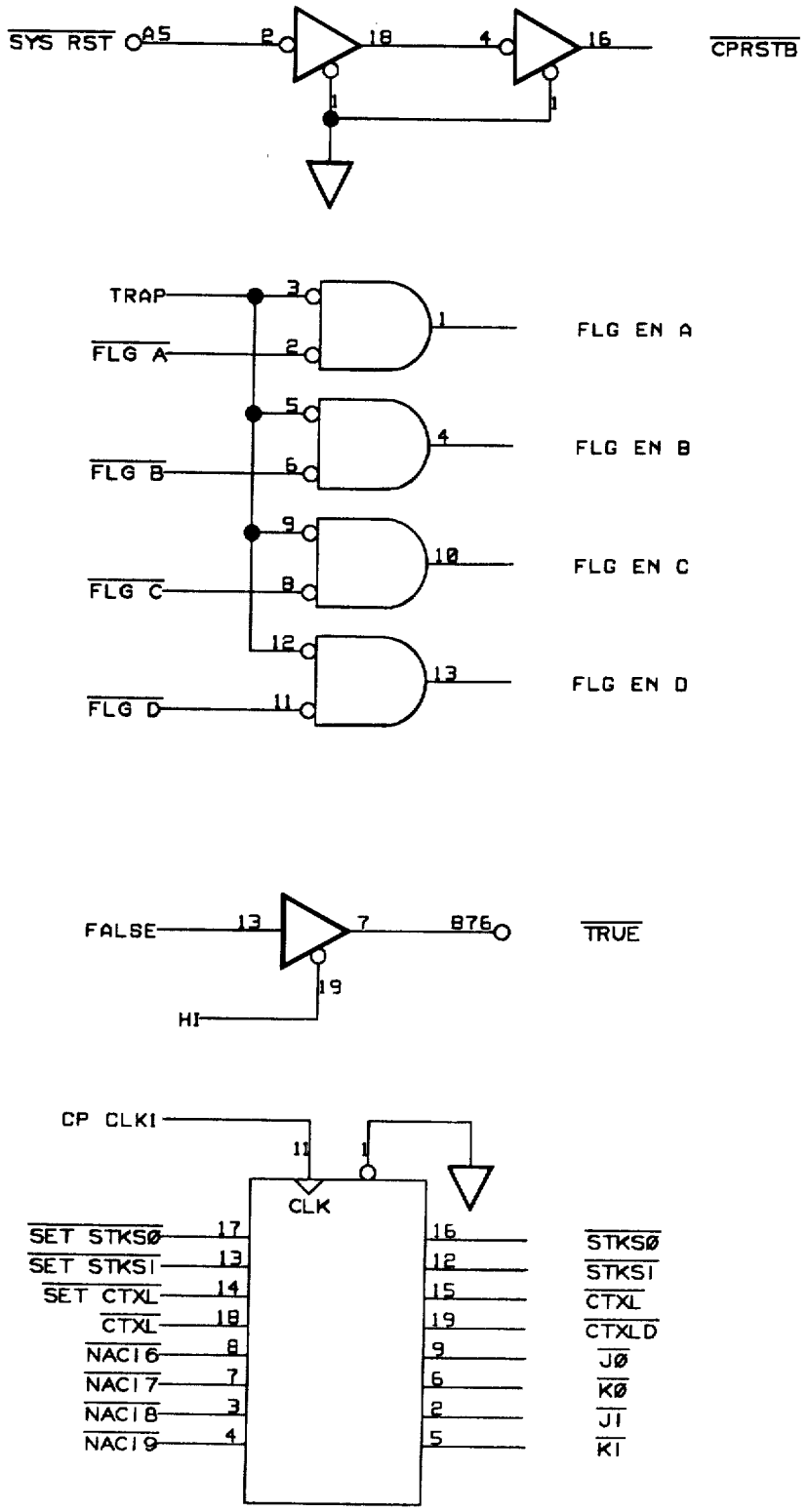

SYS RST
A5
2
18
4
16
CPRSTB
1
1
TRAP
FLG A
FLG B
FLG C
FLG D
3
2
5
6
9
8
12
11
1
4
10
13
FLG EN A
FLG EN B
FLG EN C
FLG EN D
FALSE
13
7
B76
TRUE
19
HI
CP CLKI
11
1
CLK
SET STKS0
SET STKS1
SET CTXL
CTXL
NAC16
NAC17
NAC18
NAC19
17
13
14
18
8
7
3
4
16
12
15
19
9
6
2
5
STKS0
STKS1
CTXL
CTXLD
J0
K0
J1
K1

FIG. 147D

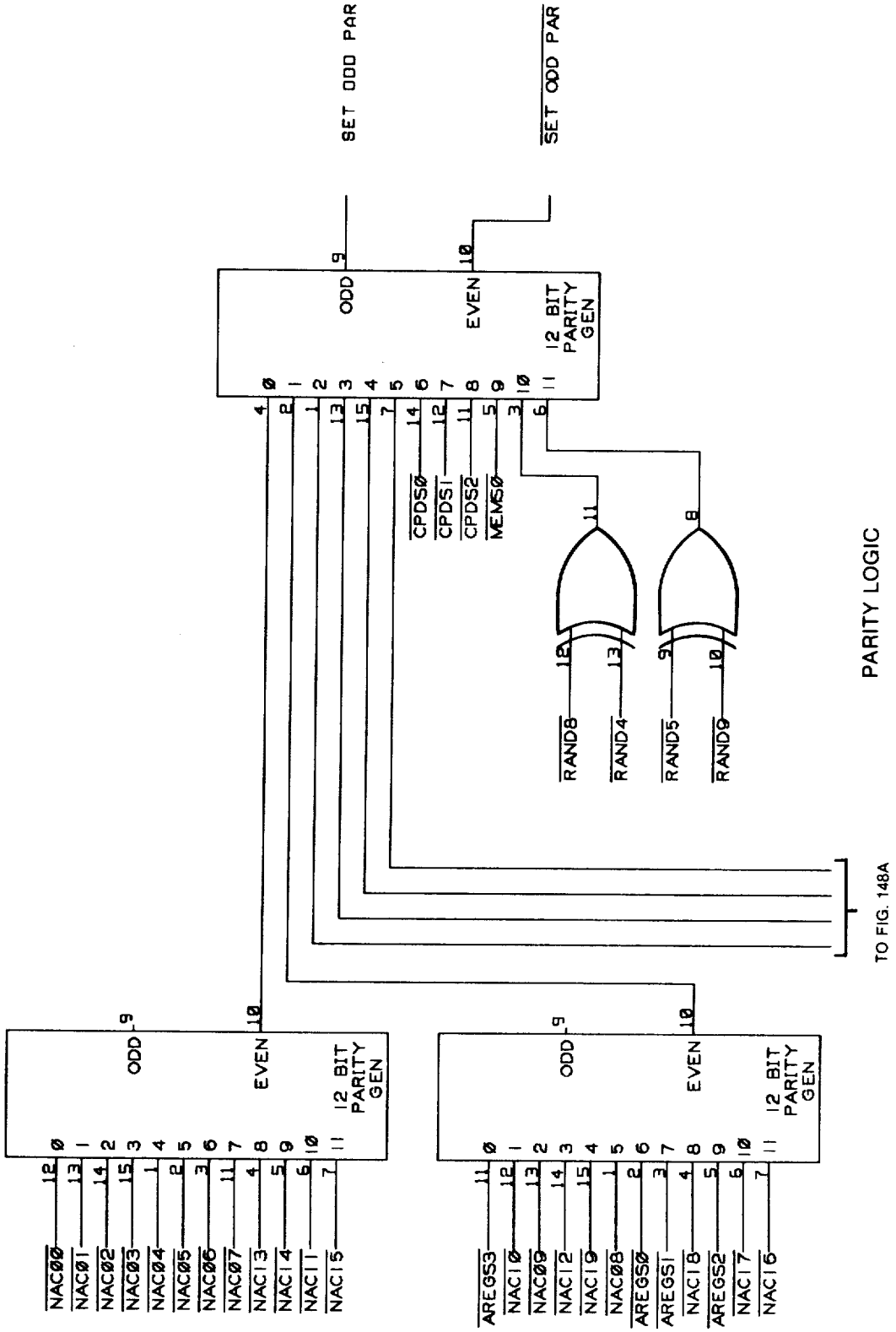

SET ODD PAR
SET ODD PAR
ODD
EVEN
12 BIT PARITY GEN
CPDS0
CPDS1
CPDS2
MEMS0
RAND8
RAND4
RAND5
RAND9
PARITY LOGIC
TO FIG. 148A
NAC00
NAC01
NAC02
NAC03
NAC04
NAC05
NAC06
NAC07
NAC13
NAC14
NAC11
NAC15
AREGS3
NAC10
NAC09
NAC12
NAC19
NAC08
AREGS0
AREGS1
NAC18
AREGS2
NAC17
NAC16

FIG. 148

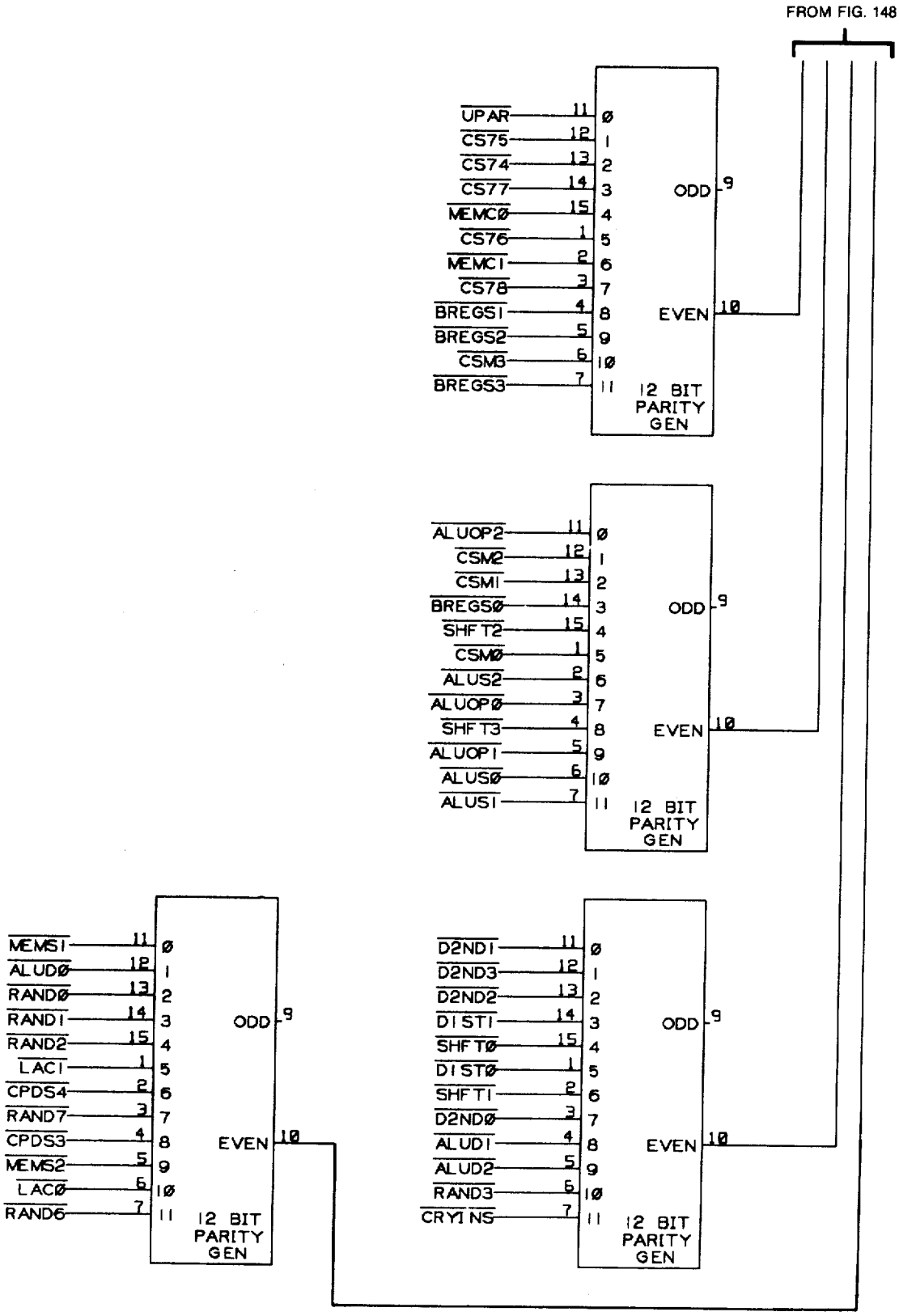
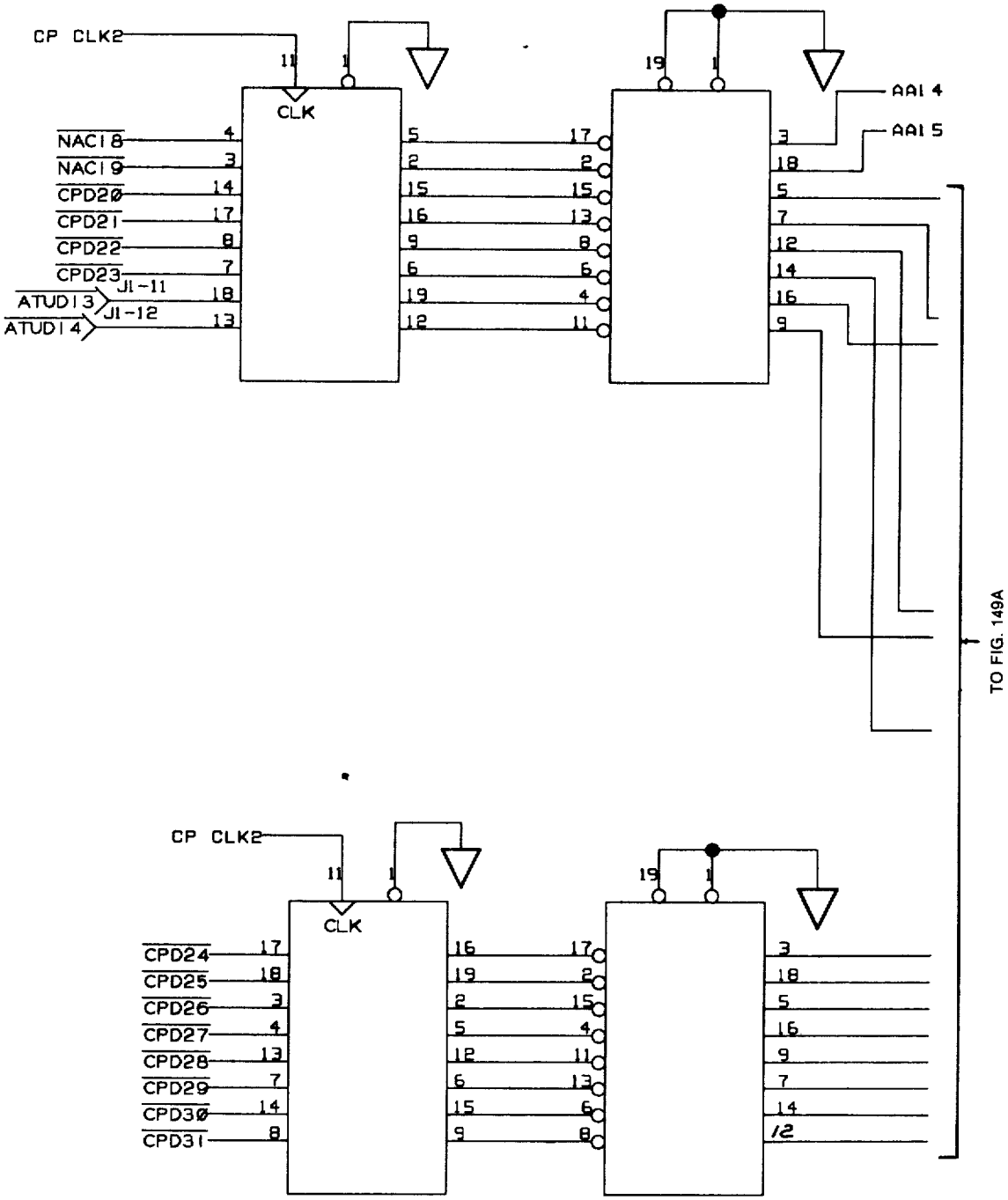

FROM FIG. 148
UPAR
CS75
CS74
CS77
MEMCØ
CS76
MEMC1
CS78
BREGS1
BREGS2
CSM3
BREGS3
ODD
EVEN
12 BIT PARITY GEN
ALUOP2
CSM2
CSM1
BREGSØ
SHFT2
CSMØ
ALUS2
ALUOPØ
SHFT3
ALUOP1
ALUSØ
ALUS1
ODD
EVEN
12 BIT PARITY GEN
MEMS1
ALUDØ
RANDØ
RAND1
RAND2
LAC1
CPDS4
RAND7
CPDS3
MEMS2
LACØ
RAND6
ODD
EVEN
12 BIT PARITY GEN
D2ND1
D2ND3
D2ND2
DIST1
SHFTØ
DISTØ
SHFT1
D2NDØ
ALUD1
ALUD2
RAND3
CRYINS
ODD
EVEN
12 BIT PARITY GEN

FIG. 148A



CONCATENATION LOGIC 185
AND DISPATCH MUX 186

FIG. 149

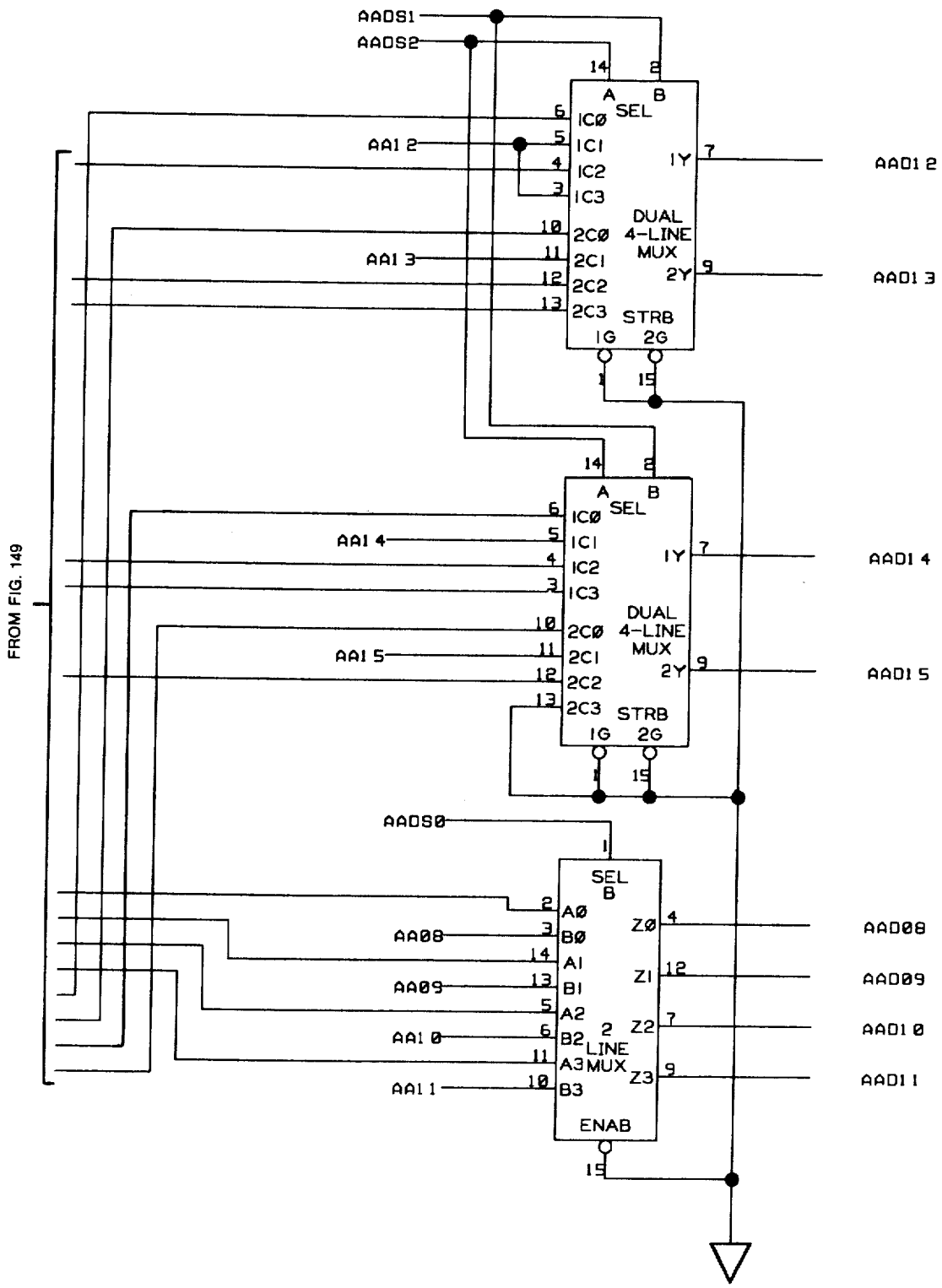
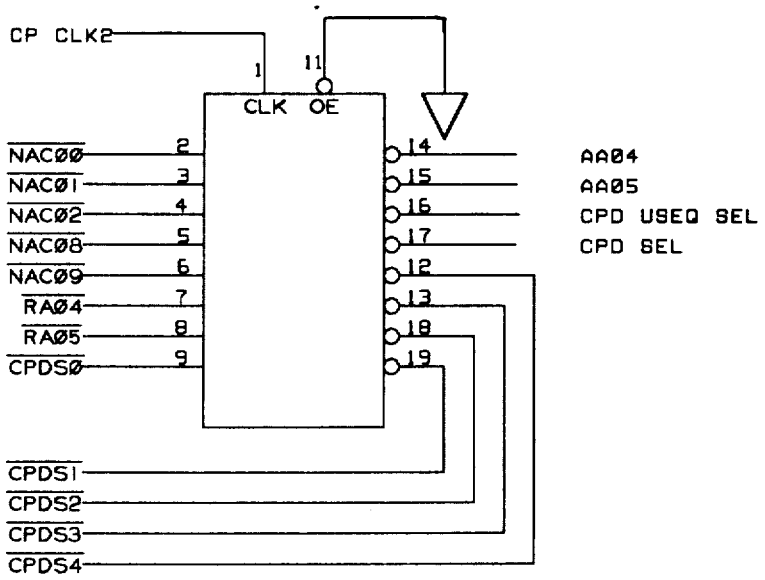

AADS1
AADS2
AA1 2
AA1 3
AA1 4
AA1 5
AADS0
AA08
AA09
AA1 0
AA1 1
FROM FIG. 149
DUAL 4-LINE MUX
SEL
STRB
1C0
1C1
1C2
1C3
2C0
2C1
2C2
2C3
1Y
2Y
1G
2G
2 LINE MUX
ENAB
A0
B0
A1
B1
A2
B2
A3
B3
Z0
Z1
Z2
Z3
AAD1 2
AAD1 3
AAD1 4
AAD1 5
AAD08
AAD09
AAD1 0
AAD1 1

FIG. 149A


CP CLK2
1
11
CLK
OE
NAC00
NAC01
NAC02
NAC08
NAC09
RA04
RA05
CPDS0
2
3
4
5
6
7
8
9
14
15
16
17
12
13
18
19
AA04
AA05
CPD USEQ SEL
CPD SEL
CPDS1
CPDS2
CPDS3
CPDS4

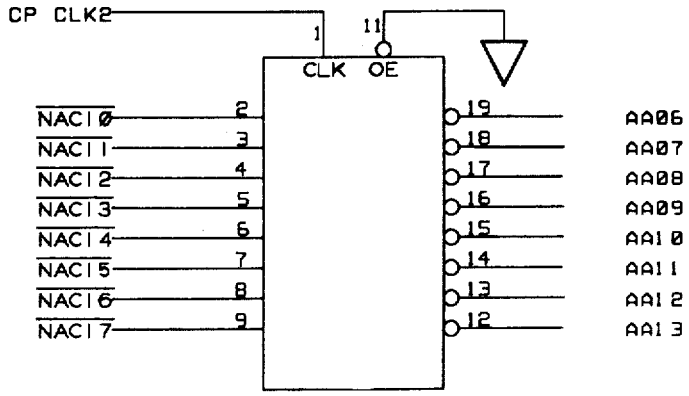
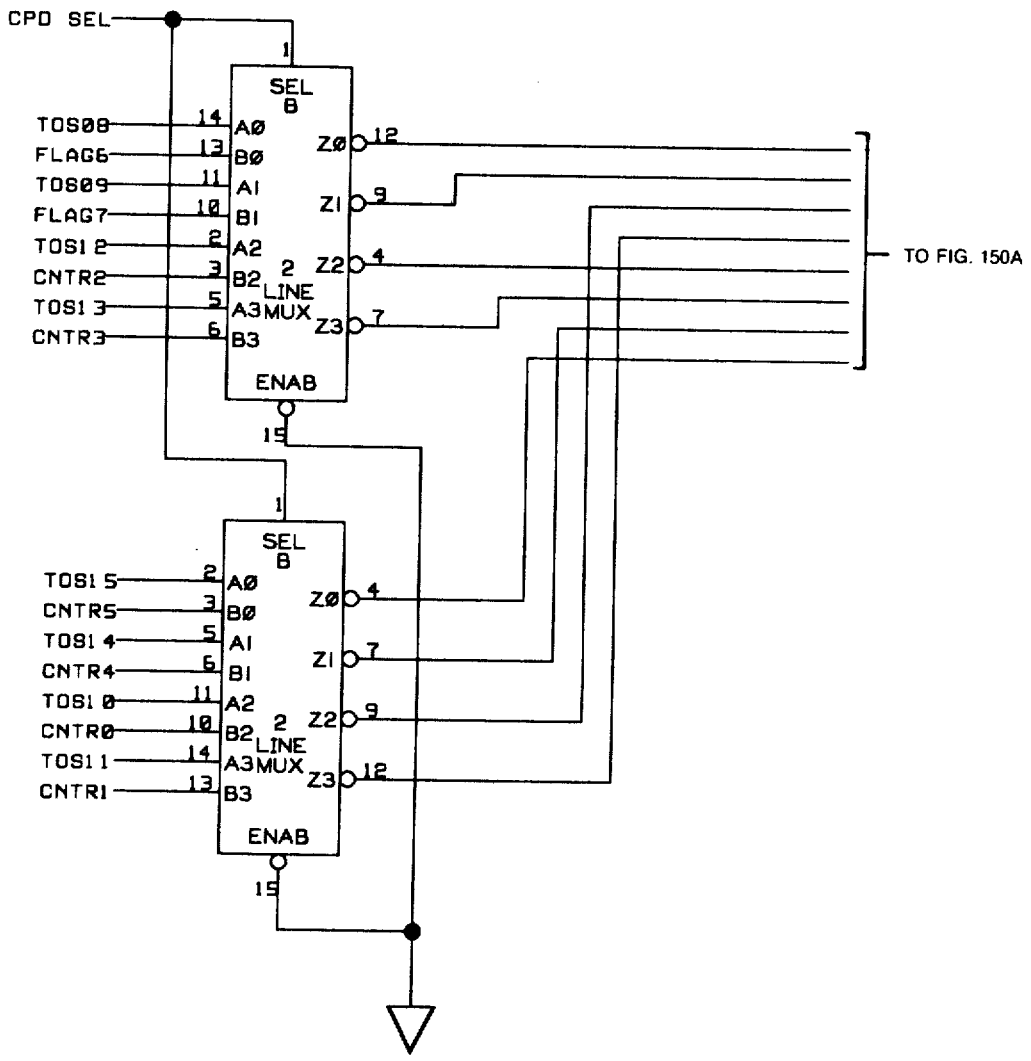

CP CLK2
1
11
CLK
OE
NAC10
NAC11
NAC12
NAC13
NAC14
NAC15
NAC16
NAC17
2
3
4
5
6
7
8
9
19
18
17
16
15
14
13
12
AA06
AA07
AA08
AA09
AA10
AA11
AA12
AA13

FIG. 149B



CPD MUX 197

FIG. 150

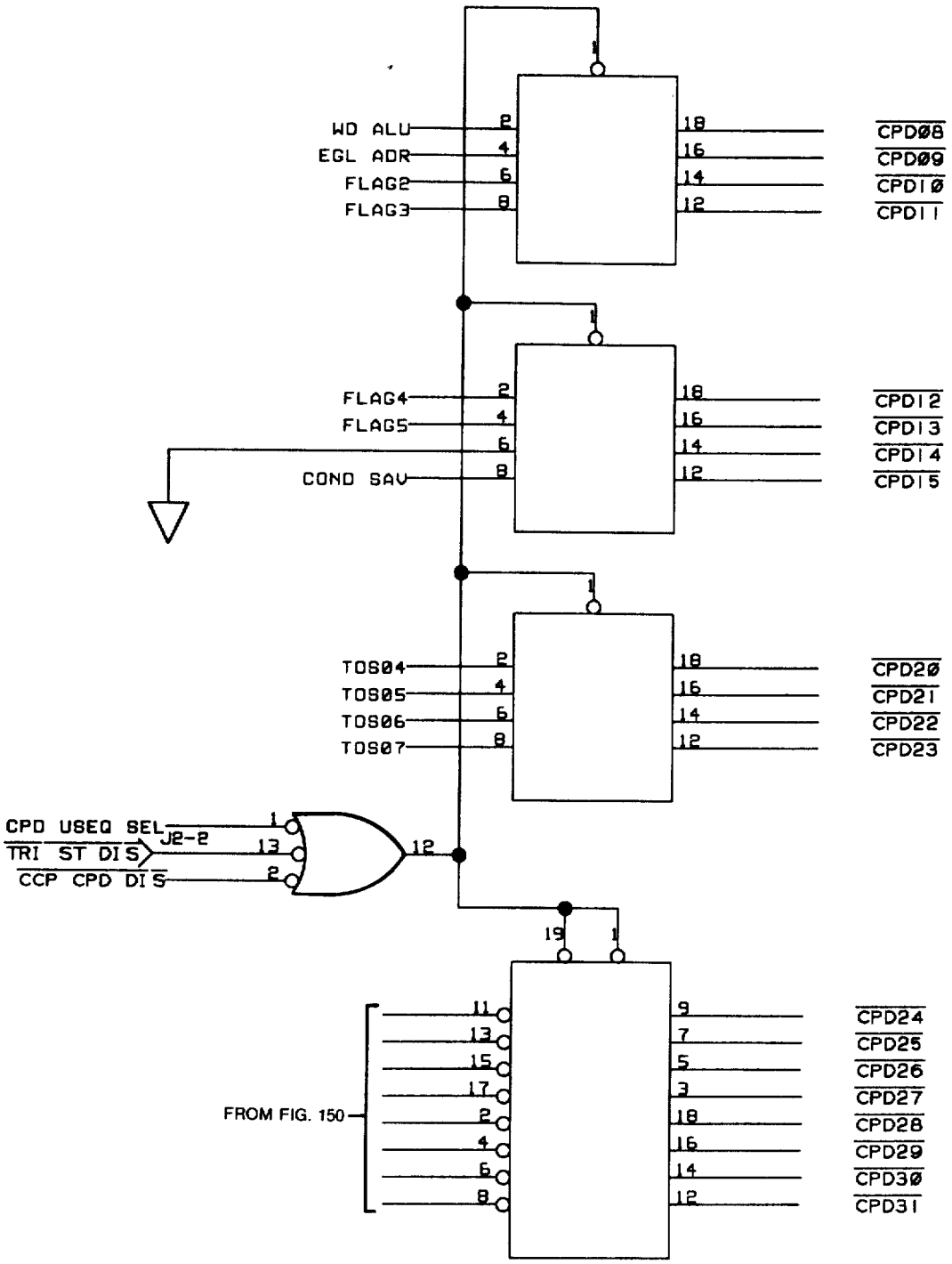
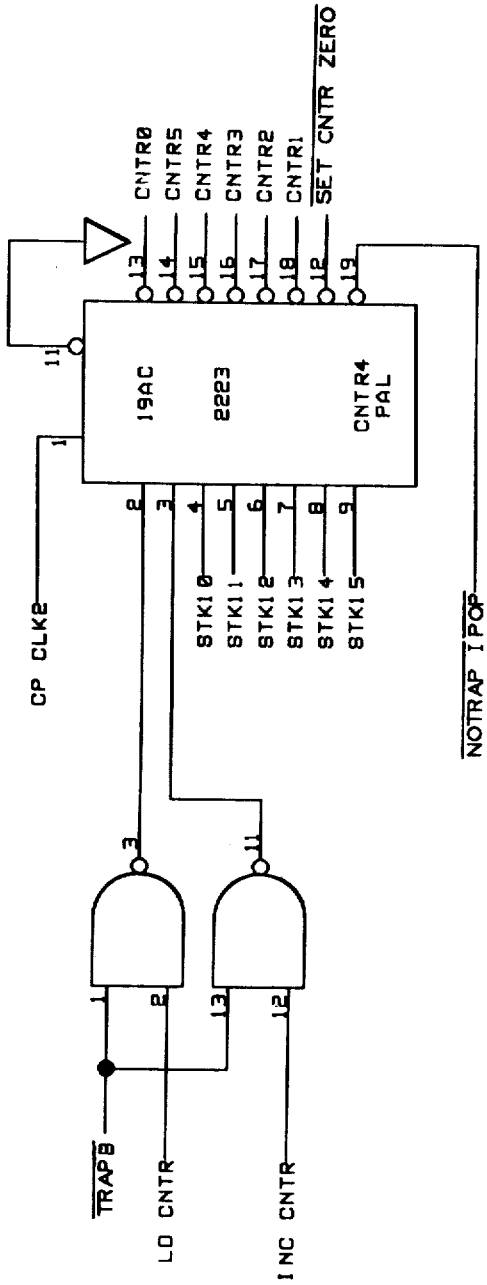
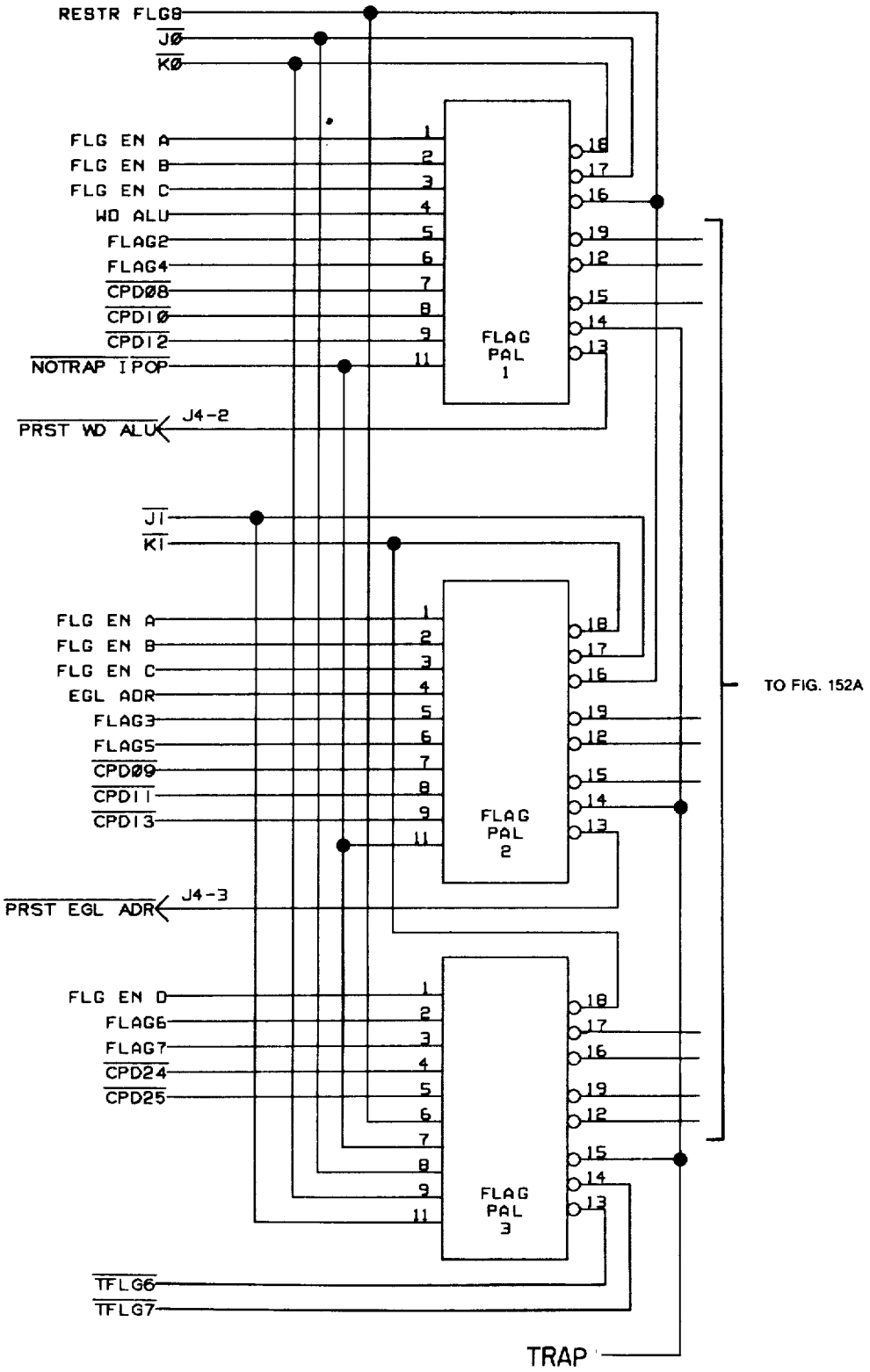

WD ALU
EGL ADR
FLAG2
FLAG3
CPD08
CPD09
CPD10
CPD11
FLAG4
FLAG5
COND SAV
CPD12
CPD13
CPD14
CPD15
TOS04
TOS05
TOS06
TOS07
CPD20
CPD21
CPD22
CPD23
CPD USEQ SEL
TRI ST DIS
CCP CPD DIS
J2-2
FROM FIG. 150
CPD24
CPD25
CPD26
CPD27
CPD28
CPD29
CPD30
CPD31

FIG. 150A



6-BIT COUNTER 196

FIG. 151



8 FLAGS UNIT 153

FIG. 152

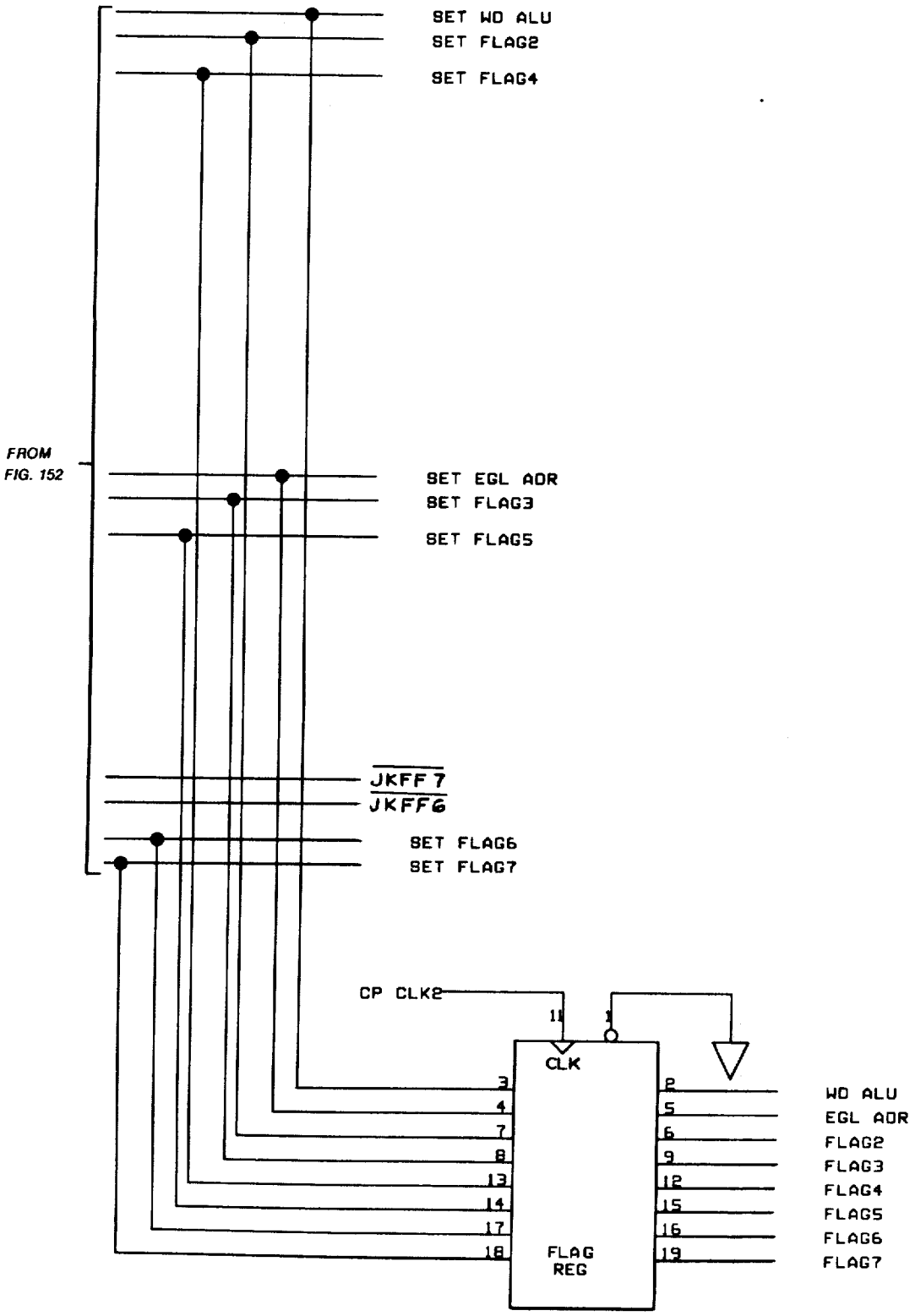
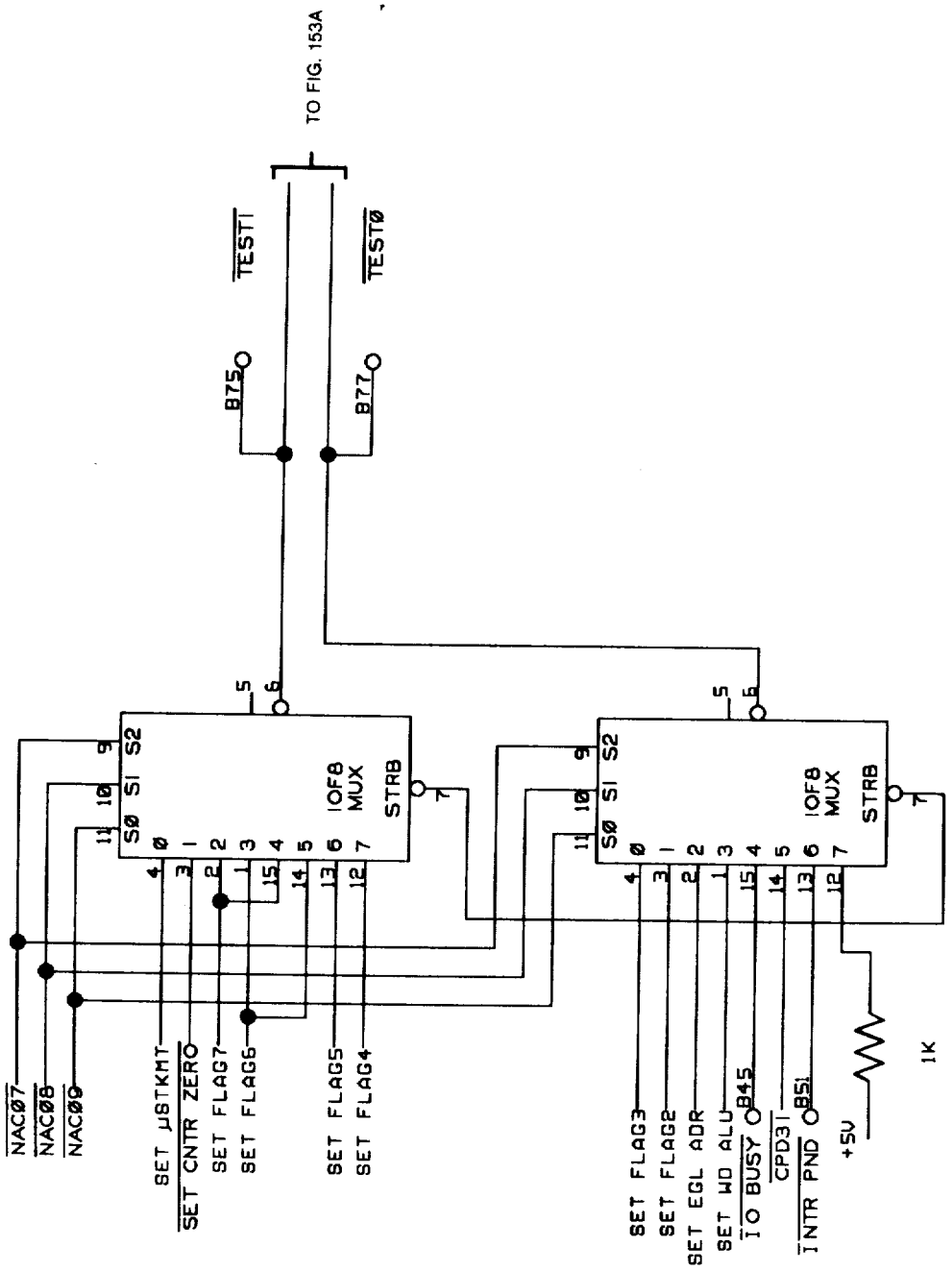

SET WD ALU
SET FLAG2
SET FLAG4
FROM
FIG. 152
SET EGL ADR
SET FLAG3
SET FLAG5
JKFF 7
JKFF6
SET FLAG6
SET FLAG7
CP CLK2
11
1
CLK
3
4
7
8
13
14
17
18
FLAG
REG
2
5
6
9
12
15
16
19
WD ALU
EGL ADR
FLAG2
FLAG3
FLAG4
FLAG5
FLAG6
FLAG7

FIG. 152A



TEST 0 AND TEST 1 MUXES 194
AND CONDITION MUX 195, COND OUT

FIG. 153

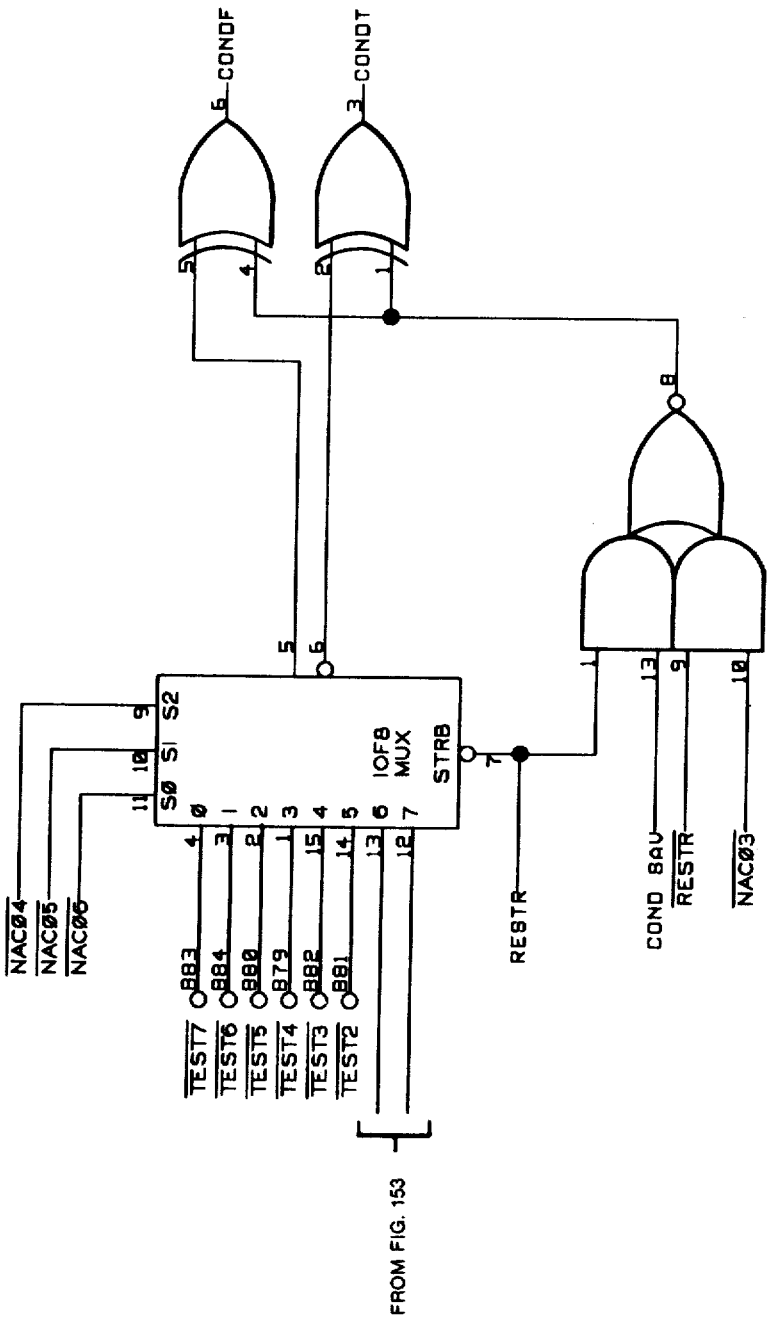

CONDF
CONDT
S0 S1 S2
IOFB MUX
STRB
NAC04
NAC05
NAC06
B83
B84
B80
B79
B82
B81
TEST7
TEST6
TEST5
TEST4
TEST3
TEST2
RESTR
COND BAV
RESTR
NAC03
FROM FIG. 153

FIG. 153A

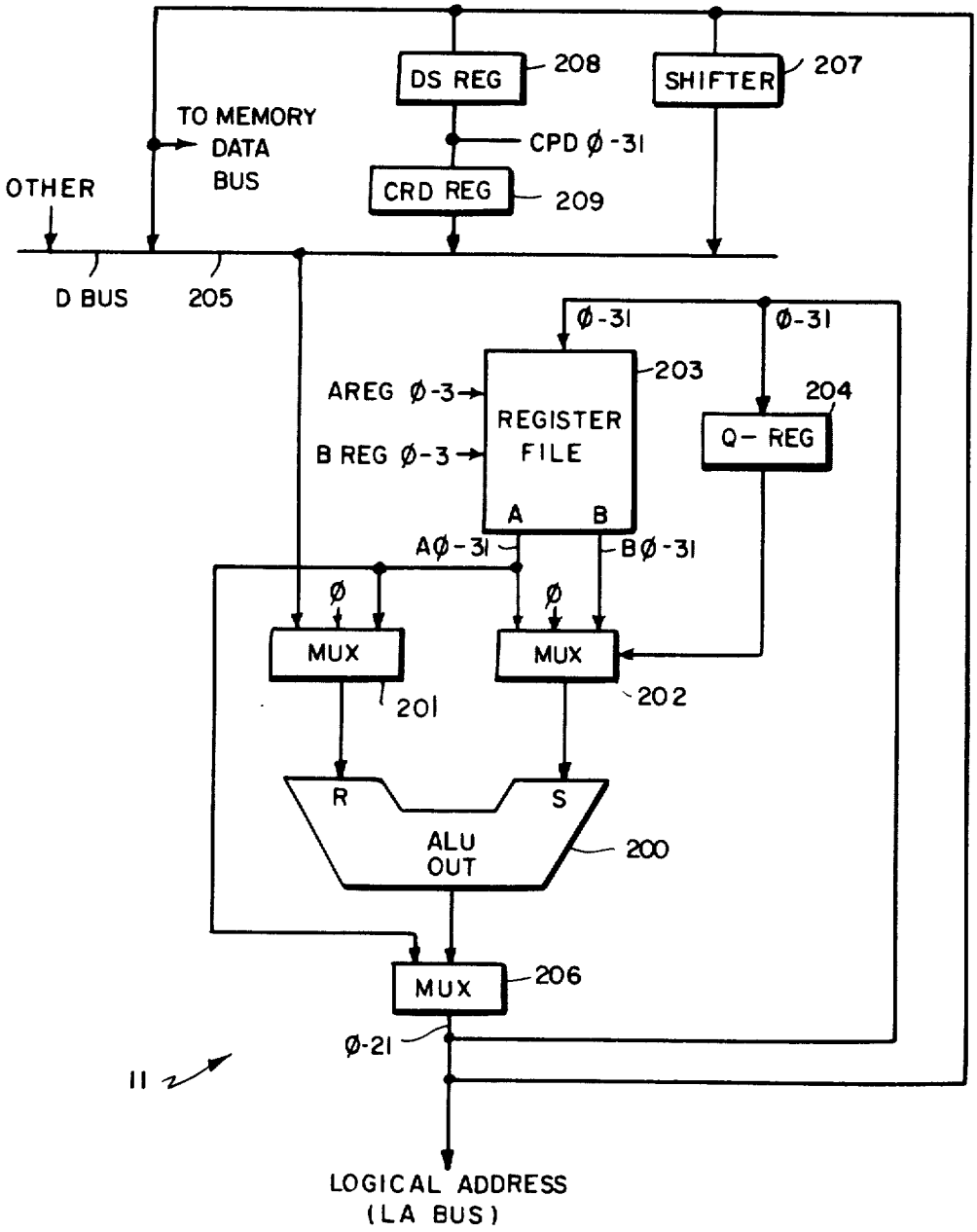

DS REG
208
SHIFTER
207
TO MEMORY
DATA
BUS
CPD Ø-31
OTHER
CRD REG
209
D BUS
205
Ø-31
Ø-31
203
AREG Ø-3
REGISTER
FILE
204
Q- REG
B REG Ø-3
A
B
AØ-31
BØ-31
Ø
Ø
MUX
MUX
201
202
R
S
ALU
OUT
200
MUX
206
Ø-21
11
LOGICAL ADDRESS
(LA BUS)

FIG.154

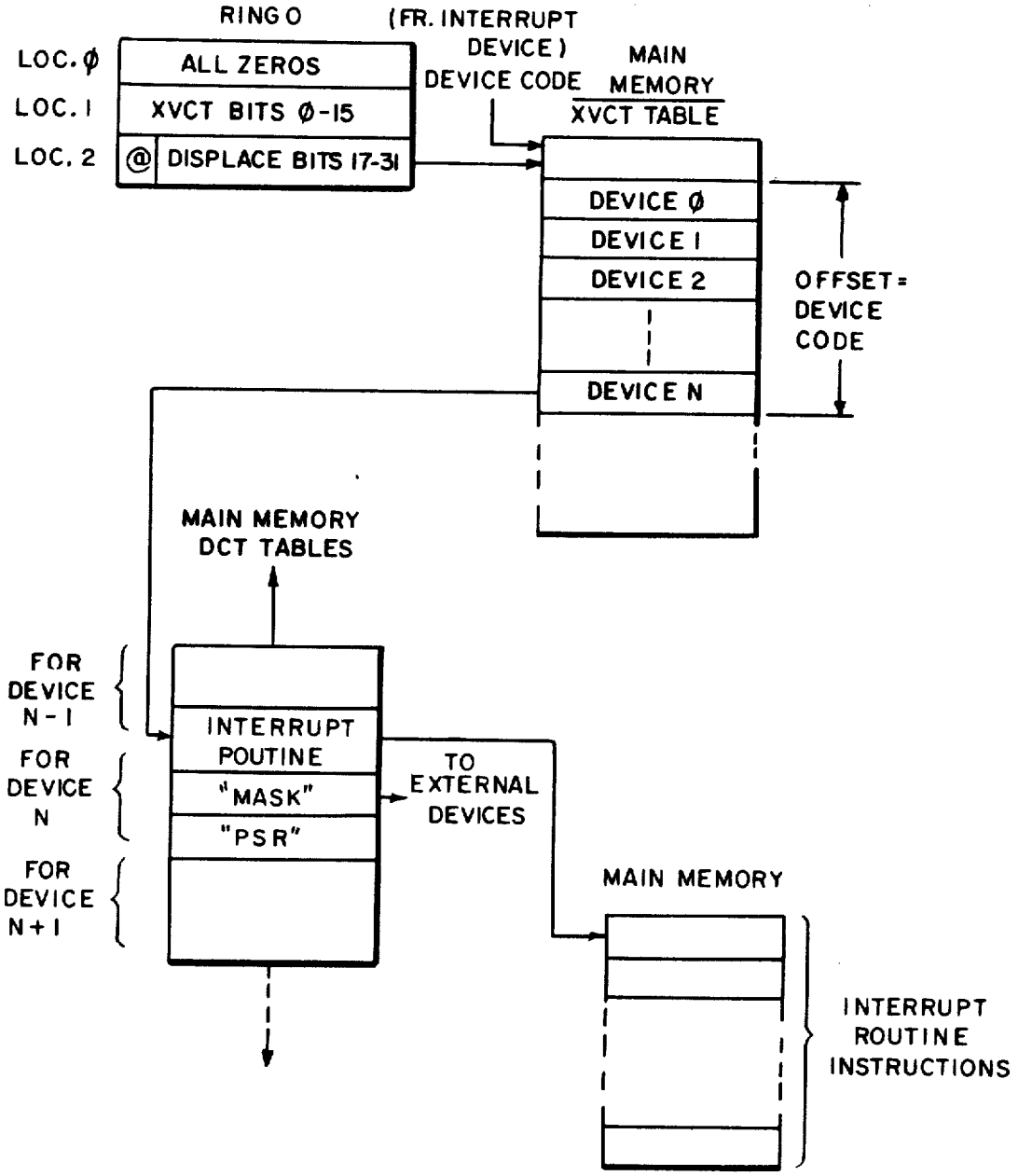

RING 0
LOC. Ø
LOC. 1
LOC. 2
ALL ZEROS
XVCT BITS Ø-15
@
DISPLACE BITS 17-31
(FR. INTERRUPT DEVICE)
DEVICE CODE
MAIN MEMORY
XVCT TABLE
DEVICE Ø
DEVICE 1
DEVICE 2
DEVICE N
OFFSET = DEVICE CODE
MAIN MEMORY DCT TABLES
FOR DEVICE N-1
FOR DEVICE N
FOR DEVICE N+1
INTERRUPT POUTINE
"MASK"
"PSR"
TO EXTERNAL DEVICES
MAIN MEMORY
INTERRUPT ROUTINE INSTRUCTIONS

FIG. 155

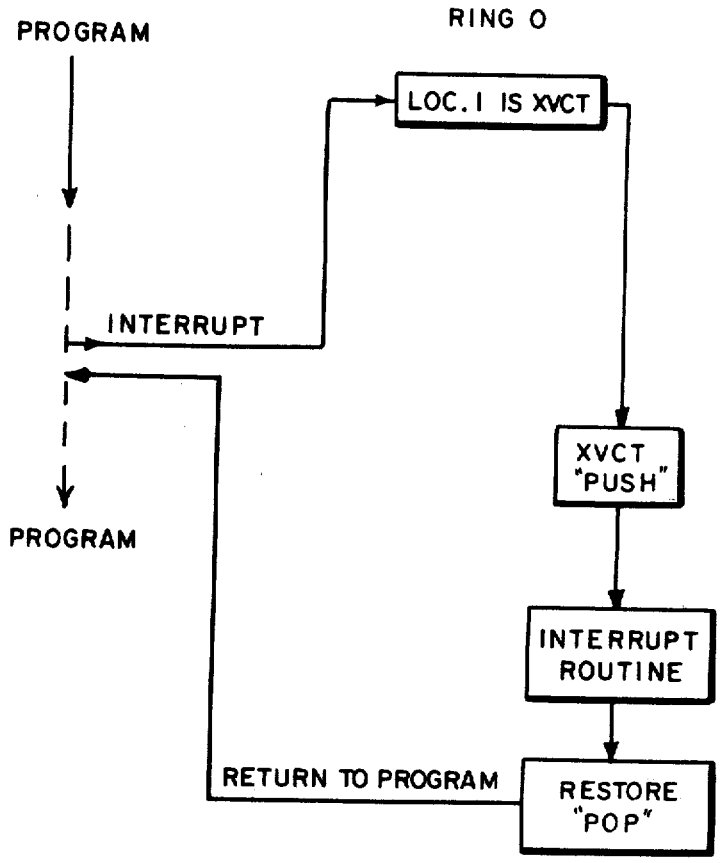

PROGRAM
RING O
LOC. I IS XVCT
INTERRUPT
XVCT "PUSH"
PROGRAM
INTERRUPT ROUTINE
RETURN TO PROGRAM
RESTORE "POP"

FIG. 156

DATA PROCESSING SYSTEM

INTRODUCTION

This invention relates generally to data processing systems and, more particularly, to such systems which can handle 32 bit logical addresses at a size and cost which is not significantly greater than that of systems which presently handle only 16 bit logical addresses.

RELATED APPLICATIONS

This application is one of the following groups of applications, all of which include the same text and drawings which describe an overall data processing system and each of which includes claims directed to a selected aspect of the overall data processing system, as indiciated generally by the titles thereof as set forth below. All of such applications are being filed concurrently and, hence, all will have the same filing date.

(1) Data Processing System, Ser. No. 143,561, filed by E. Rasala, S. Wallach, C. Alsing, K. Holberger, C. Holland, T. West, J. Guyer, R. Coyle, M. Ziegler and M. Druke;

(2) Data Processing System Having A Unique Address Translation Unit, Ser. No. 143, 681, filed by S. Wallach, K. Holberger, S. Staudener and C. Henry;

(3) Data Processing System Utilizing a Hierarchical Memory Storage System, Ser. No. 143,981, filed by S. Wallach, K. Holberger, D. Keating and S. Staudener;

(4) Data Processing System Having A Unique Memory System, Ser. No. 143,974, filed by M. Ziegler and M. Druke;

(5) Data Processing System Having A Unique Instruction Processor System, Ser. No. 143,651, filed by K. Holberger, J. Veres, M. Ziegler and C. Henry;

(6) Data Processing System Having A Unique Microsequencing System, Ser. No. 143,710, filed by C. Holland, K. Holberger, D. Epstein, P. Reilly and J. Rosen;

(7) Data Processing System Having Unique Instruction Responsive Means, Ser. No. 143,982, filed by C. Holland, S. Wallach and C. Alsing.

BACKGROUND OF THE INVENTION

Presently available data processing systems which are often referred to as belonging to the "mini-computer" class normally handle logical addresses and data words which are 16 bits in length. As used herein, the term "logical" address, sometimes referred to by those in the art as a "virtual" address, is used to denote an address that is programmer visible, an address which the programmer can manipulate. In contrast, a "physical" address is the address of a datum location in the main memory of a data processing system. Operating data processing systems utilize appropriate translation tables for converting logical addresses to physical addresses.

Such mini-computers have been successfully used in many applications and provide a high degree of data processing capability at reasonable cost. Examples of such systems which have found favor in the market place are those known as the "Nova" and the "Eclipse" systems designed and developed by Data General Corporation of Westboro, Mass. The Nova and Eclipse family of mini-computers are described in the publications available from Data General Corporation which

are listed in Appendix A incorporated as part of this specification.

The Nova system provides a logical address space of 64 kilobytes (the prefix "kilo" more accurately represents 1024, or 2^{10}) and the Eclipse system also provides a logical address space of 64 kilobytes, both being proven systems for handling many applications at reasonable cost. It is desirable in the development of improved systems to provide for an orderly growth to an even larger logical address space than presently available in Nova and Eclipse systems. Such an extended logical address base permits a larger set of instructions to be utilized by the system, the enlarged instruction set being capable of including substantially all of the basic instructions now presently available in the prior Nova and Eclipse systems as well as a large number of additional, or extended, instructions which take advantage of the increased or expanded logical address space.

Accordingly, such an improved system should be designed to be responsive to software which has been previously designed for use in Nova and Eclipse systems so that those presently having a library of Nova and Eclipse software, representing a substantial investment, can still use such software in the improved, expanded address system. The improved system also would provide for a greater flexibility in performance at a reasonable cost so as to permit more on-line users at a larger number of on-line terminals to utilize the system. The expanded address space would further permit the system is support more extensive and sophisticated programs devised specifically therefor, as well as to support all of the previous programs supported by the unextended Nova or Eclipse systems.

BRIEF SUMMARY OF THE INVENTION

The system of the invention utilizes a unique combination of central processor and memory units, the processor comprising an address translation unit, an instruction processor unit, an arithmetic logic unit and a microsequencing unit, while the memory unit includes a system cache unit, a main memory unit and a bank controller unit for controlling data transfers therebetween. The system handles thirty-two bit logical addresses which can be derived from either sixteen bit or thirty-two bit addresses. Unique means are provided for translating the thirty-two bit logical addresses. The system uses hierarchical memory storage, wherein information is stored in different segment storage regions (rings), access to the rings being controlled in a privileged manner so that access to different rings are governed by different levels of privilege.

The memory system uses a main memory comprising a plurality of memory modules each having a plurality of memory planes. The main memory normally interfaces with the remainder of the system via a dual port system cache memory unit, block data transfers between the main memory and the system cache are controlled by a bank controller unit.

Macro-instructions are decoded using a unique programmable read-only-memory means which is capable of decoding instructions of two types, i.e., instructions from a first basic instruction set or instructions from a second extended instruction set, the instruction which is being decoded containing in itself selected bit patterns which uniquely identify which type of instruction is to be decoded.

The decoded instructions provide the starting address of one or more microinstructions, which starting ad-

dress is supplied to a unique microinstruction sequencing unit which appropriately decodes a selected field of each microinstruction for determining the address of the next successive microinstruction, such address being suitably selected from a plurality of microaddress sources.

The overall system includes means responding to certain macro-instructions which perform unique operations indigenous to the overall system.

DESCRIPTION OF THE INVENTION

The invention can be described in more detail with the help of the drawings wherein:

FIG. **1** shows a block diagram of the overall data processing system of the invention as described therein;

FIG. **2** shows a block diagram of the system cache unit of the system of FIG. **1**;

FIG. **3** shows a block diagram of the bank controller unit of the system FIG. **1**;

FIG. **4** shows a block diagram of a module of the main memory unit of the system of FIG. **1**;

FIGS. **5–44** show specific logic circuitry for implementing the system cache of FIG. **2**;

FIGS. **45–63** show specific logic circuitry for implementing the bank controller of FIG. **3**;

FIGS. **64–78** show specific logic circuitry for implementing the main memory modules of FIG. **4**;

FIGS. **79–81** show block diagrams which represent the address translation unit of the system of FIG. **1**;

FIGS. **82–100** show specific logic circuitry for implementing the address translation unit of FIGS. **79–81**;

FIGS. **101–106** show block diagrams which represent the instruction processor unit of the system of FIG. **1**;

FIGS. **107–136** show specific logic circuitry for implementing the instruction processor unit of FIGS. **101–106**;

FIGS. **137** and **138** show block diagrams of the microsequencer unit of the system of FIG. **1**;

FIGS. **139–153** show specific logic circuitry for implementing the microseqencer unit of FIGS. **137** and **138**;

FIG. **154** shows a block diagram of a representative arithmetic logic unit of the system of FIG. **1**;

FIG. **155** shows a diagrammatic representation of certain memory locations used to explain the operation of a particular macro-instruction used in the system of FIG. **1**; and

FIG. **156** shows a diagrammatic representation of certain operations performed in the macro-instruction discussed with reference to FIG. **155**.

In connection with the above figures, where a particular figure requires more than one sheet of drawings, each subsequent sheet is designated by the same figure member with sequential letters appended thereto (e.g., FIG. **5** (for sheet 1); FIG. **5A** (for sheet 2); FIG. **5B** (for sheet 3) . . . etc.). With respect to FIG. **146** in particular, which depicts the microcontrol store **170**, fifty-six sheets of drawing are used. The sheets are numbered **146**, **146A**, **146B**, **146C**, **146D**, **146E**, **146F**; **146.1**, **146.1A**, **146.1B**, **146.1C**, **146.1D**, **146.1E**, **146.1F**; **146.2**, **146.2A**, **146.2B** . . . etc. to **146.8**, **146.8A**, **146.8B** . . . **146.8F**.

GENERAL DESCRIPTION

Before describing a specific implementation of the system of the invention, it is helpful to discuss the overall concept thereof in more general terms so that the characteristics that are desired can be described and the description of a particular implementation can be better understood.

A significant aspect of the system of the invention, as discussed above, is the size of the logical address space which is available. For purposes of convenience in distinguishing between the previous NOVA and Eclipse systems, the extended system as discussed herein will sometimes be referred to as the "Eagle" system. In the Eagle system, for example, the logical address space can be as high as 4 gigabytes (more accurately the prefix "giga" is 1,073,741,824, or 2^{30}, so that 4 gigabytes is, more accurately, 4,294,967,296) where a byte is defined as having 8 bits of precision. As used hereinafter, a "word" is defined as having 16 bits of precision (i.e., equivalent to 2 bytes) and a "double-word" as having 32 bits of precision (equal to two words, or four bytes). Because of the increased logical address space the overall system is able to support an instruction set which is larger than that supported by a Nova system or an Eclipse system having, for example, a much smaller logical address space. The overall capability of the system can be best understood by those in the art by examination of the set of the extended instructions which are capable of being performed by the system. Such an instruction set in accordance with the invention is set forth in Appendix B incorporated as a part of this specification. Such instruction set includes the extended instruction set (which can be referred to as the Eagle instruction set) and the Eclipse C-350 instruction set, as well as the Nova instruction set, all of which are capable of being handled by the system, the latter two instruction sets being already disclosed as part of the above publications. All Nova and Eclipse instructions are executed according to the principles and specifications presented in the above-referenced publications.

The binary encodings of the extended instructions which are supported by the sytem of the invention are shown in Appendix B. A significant difference exists between the systems having extended instructions in accordance with the invention and systems having extended instructions which have been suggested by others. In any system in which an extended instruction set effectively represents a "super" set of a previous, or original, set of instructions, all of the instructions must be suitably decoded for machine operations. Normally, such systems utilize a decoding sub-system for decoding both the original instruction set and for decoding the extended instruction set. The decoder operates so as to permit the decoding of only one of the instruction sets at a time, the original instruction set and the extended instruction set being in effect, mutually exclusive. In order to determine which instruction is to be decoded, a unique instruction must be used to set a "mode bit", i.e., a single bit which in one state indicates that the original instruction set is to be decoded and in the other state indicates that the extended instruction set is to be decoded. However, in neither case can the decoding subsystem be made available to decode either of the both sets simultaneously. Such approach inserts a limitation on the overall machine operation since it is never possible to simultaneously decode instructions from different instruction sets of an overall super set thereof.

The system of the invention, however, avoids such mutual exclusivity and is arranged to be capable of decoding instructions from either set or both sets at any one time. A decoder PROM (programmable read-only-memory) system is utilized for decoding both the extended Eagle instruction set and the original or basic

instruction sets as, for example, the original Nova and Eclipse instruction set. Each instruction to be decoded includes the information which determines which decoder is to be utilized, such determination thereby being inherently carried in each instruction word which is to be decoded. As seen in Appendix B, for example, the information is contained in bits **Ø** and **12–15**. Thus, in the extended Eagle instruction set, bit **Ø** is always a "1" while bits **12–15** are always "1001" for all instructions of the extended instruction set except for those extended instructions which use a "1" in bit **Ø** and the encoding "011000" **in bits** 10–15 **and a** "1" in bit "0", a "0" in bit **5**, and the encoding "111000" in bits **10–15**. On the other hand, the original Eclipse instructions are such that bit **Ø** is 0 and bits **12–15** are "1000". Further, in cases where the instruction does not carry either the Eagle coded bits or the Eclipse coded bits, such instruction is interpreted as a NOVA instruction.

Because each instruction carries with it an identification as to which instruction set the instruction belongs, the system operates to decode instructions on a non-mutually exclusive basis.

In order to support the extended operations of the system, the configuration thereof requires an augmentation of the registers which were previously available in the original system of which the new system is an extension. The following registers are utilized in the system and are discussed in more detail later with respect to the particular implementation described in connection with specific figures below.

The register set includes fixed point registers, floating point registers, stack management registers and memory management registers.

Fixed Point Registers

The system includes four fixed point accumulators (ACC **Ø**-3), one program counter (PC) and one processor status register (PSR). Each of the accumulators has 32 bit precision which can accomodate (1) a 16 bit operand which can be sign extended to 32 bits; (2) a 15 bit address which can be zero extended to 28 bits, the higher order 3 bits of the program counter being appended thereto together with a zero bit, all of which can be appended for storage in the accumulator; or (3) an 8 bit byte which can be zero extended to 32 bits before storage in the accumulator.

The program counter has 31 bits of precision, bits **1–3** identifying one of 8 current memory rings (discussed in more detail below) and bits **4–31** of which accomodate an address offset for instruction addresses. For Eclipse operation, for example, which normally requires only a 15 bit program counter, the bits **1–3** identify the current memory ring as in a 31 bit extended operation while the 15 least significant bits **17–31** represent the 15 bit Eclipse program counter and bits **4–16** are all zeros.

The processor status register is a 16 bit register which provides an overflow mask bit which if set will result in a fixed point overflow. Additionally the register includes a fixed point overflow indicator bit and a bit which indicates that a micro interrupt has occurred. Other bits in the register are reserved and are thus available for potential future use.

Floating Point Registers

The system includes four floating point accumulators (FPAC **Ø**-3) and one floating point status register (FPSR). Each of the floating point accumulators contains 64 bits of precision which is sufficient to wholly contain a double precision floating point value. The floating point registers of the extended system are identical to the Eclipse floating point accumulators (FPAC) which are discussed in the aforare identical to the Eclipse floating point accumulatorprecision, 32 bits of which act as the floating point program counter. In the event of a floating point fault the floating point program counter bits define the address of the floating point instruction that caused the fault. Four other bits are utilized, respectively, to indicate an exponent overflow condition, an exponent underflow condition, a divide-by-zero condition and a mantissa overflow condition. Another counter bit will result in a floating point fault if any of the above latter four bits are also set. The floating point counter also includes a zero bit and negative bit, as are generally used in status registers, as well as bits for indicating a floating point rounding mode of operation and an interrupt resume operations.

Stack Management Registers

The system of the invention utilizes four 32 bit registers to manage the memory stack, which registers include a stack pointer, a stack limit, a stack base, and a frame pointer. The stack pointer register references the double word entry at the top of the stack. When a "push" operation occurs, all the bits of the stack pointer are incremented by 2 and the "pushed" object is placed in the double word addressed by the new value of the stack pointer. In a "pop" operation the double word addressed by the current value of the stack pointer is placed in a designated register and 32 bits of the stack pointer are then decremented by 2.

The frame pointer register references the first available double word minus two in the current frame. The stack limit contains an address that is used to determine stack overflow. After any stack operation pushes objects onto the stack, the stack pointer is compared to the stack limit. If the stack pointer is greater that the stack limit a stack fault is signaled. The stack base contains an address that is used to determine the stack underflow. After any stack operation that pops objects from the stack, the stack pointer is compared to the stack base. If the stack pointer is less than the stack base a stack fault is signaled.

Memory Management Registers

Eight registers are used to manage memory, such registers each being designated as a segment base register (SBR) having 32 bits of precision, the memory being divided into eight segments, or rings, thereof. The SBR's in the system described herein are formed as part of scratch pad registers on an address translation unit (ATU) of the system, as discussed in more detail below. One bit of each SBR indicates whether or not the segment associated therewith can be referenced (i.e. is there a valid or an invalid reference to such segment). Another bit indicates the maximum length of the segment offset field i.e. whether or not the reference is a one level page table or a two level page table, as explained in more detail below. A third bit of each segment base register indicates whether a Nova/Eclipse instruction for loading an effective address of a Nova/Eclipse I/O instruction is being executed. Another bit represents a "protection" bit which indicates whether or not an I/O instruction can be executed or whether the execution thereof would be a violation of the protection granted to such segment. Nineteen of the bits contain a physical address which identifies the

physical address in the memory of the indicated page table. Discussions of the addressing of page tables in the memory are presented in more detail below including a discussion of the memory locations in each segment.

Overall System

A block diagram of a preferred embodiment of the invention is shown in FIG. **1**. The central processor portion of the system comprises an arithmetic logic unit (ALU) **11**, an instruction processor unit **12**, a micro-sequencer unit **13** and an address translation unit (ATU) **14**. The memory system includes a main memory unit **16**, an auxiliary cache memory unit **17** and a memory control unit identified as bank controller unit **18**. A central processor address bus **19** permits the transfer of addresses among the instruction processor unit **12**, the address translation unit **14** and the memory system. A control processor, memory (CPM) bus **20** permits the transfer of instructions and operands among arithmetic logic unit **11**, instruction processor unit **12**, address translation unit **14** and the memory system **15**.

I/O address bus **21** and I/O memory/data bus **22** permit the transfers of addresses and data respectively with respect to I/O devices via I/O channel unit **23**, as well as the transfers thereof between the memory system and a console control processor unit **24**. Suitable control buses for the transfer of control signals among the various units of the overall system are provided as buses **25–31** described in more detail below. Appropriate teletype and floppy disc systems **33** and **34**, respectively, can be utilized with the system, particularly in the diagnostics mode of operation via console control processor unit **24** by way of a suitable micro processor computer **35**.

The inventive aspects of the system to be described herein require a more detailed discussion of the memory system, the address translation unit, the instruction processor unit and the micro sequencer unit. The arithmetic logic unit, the console processor unit and the I/O channel unit with their associated controls need not be described in detail.

Memory System

In accordance with a preferred embodiment of the invention the memory system comprises up to two megabytes of main memory **16** and, if desired, the system can be expanded even further as, for example, to 4 megabytes. It should be noted that sufficient bits are reserved in the physical address fields so as to allow for system expansion to one billion bytes of memory. The interface between the main memory unit **16** and the remainder of the system is via the dual port cache memory unit **17**, data being transferred between the main memory and the cache memory unit in blocks of 16 bytes. The cache memory unit herein will usually be referred to as the "system cache" (SYS CACHE) to distinguish it from a separate cache memory in the instruction processor unit which latter memory will normally be referred to as the "instruction cache" (I CACHE) unit. The system cache unit **17** services CPU requests for data transfers on port **17A** of its two ports and services requests from the I/O system at port **17B** thereof. CPU data transfers can include "byte-aligned-byte" transfers, "word-aligned-word" transfers, and double word transfers. I/O data transfers can include "word-aligned-word" transfers, "double word-aligned-double word" transfers and 16 byte block transfers.

The main memory unit **16** can include from one to eight 256-kilobyte memory modules, as shown in FIG. **4**. Each memory module contains a memory array of 156 16 K dynamic random access memories (RAMs), organized at each module in the form of four planes **Ø-3** of 16 K 39-bit words each. Each word comprises 32 bits of data and 7 error correction bits, as discussed in more detail below. Memory timing and control for the RAMs of each memory module is accomplished on the memory bank controller board **18**. The control signals from the memory bank controller are clocked into a register on each memory module, the outputs thereof driving the "plane-Ø" RAMs. The outputs from such register are clocked a fixed time later into another register which drives the "plane-**1**" RAMs. Such pipe line operation continues through "plane-**2**" RAMs and "plane-**3**" RAMs so that all four planes receive the same control signals at fixed intervals (e.g. 110 nanosecond intervals), resulting in the transfer of a block of four consecutive 39-bit words.

Memory bank controller **18** has three main functions. First of all, it provides an interface between the system cache **17** and the memory modules of the main memory unit **16**. Secondly, it performs necessary error checking and correction operation and, thirdly, it controls the refresh operation of the dynamic RAMs on each of the memory modules. The details of the interface between the system cache and the bank controller are discussed in more detail below.

The error checking and correction logic on the bank controller performs single-bit error correction and double-bit error detection using a 7 bit error correction Hamming code as is well known in the art. The 7 check bits generated for each 32 bit data word are stored with such word in the main memory modules. When the word is subsequently read from memory, all 39 bits are decoded to produce a 7 bit pattern of syndrome bits which pattern identifies which, if any, single bit is in error and indicates when more than one bit is in error. When a correctable single-bit occurs, the console processor **24** is provided with the address and the syndrome bit pattern of the failing bit. The data is thereupon corrected and sent to the system cache after a fixed time delay equal to a system clock period, e.g. 110 nanoseconds in a particular embodiment, in accordance with well-known error correcting operation, the remaining words in the pipe line operation being prevented from transfer until the corrected signal is made available by the use of a suitable inhibit signal identified as the BC ERROR signal.

Substantially immediate correction of single bit errors is desirable so that such errors do not grow into multiple bit errors. A conventional technique can be used in which the corrected data is written back into memory only when it has been read and found to be in error. Two problems arise with such a technique. First of all, the memory locations which are not often read are not often corrected and, secondly, significant time can be wasted in trying to correct a failure if it occurs in a frequently accessed memory location. The system of the invention can avoid such problems by utilizing a separate process for monitoring all of the main memory locations so that each location therein is checked and corrected, if necessary, once every two seconds. Such checking is performed during the memory refresh cycle and does not reduce the availability of the memory to the system. A detailed description of such a technique is disclosed in U.S. Patent Application Ser. No. 143,974,

filed concurrently by M. Ziegler, M. Druke, W. Baxter and J. VanRoeckle, which application is incorporated by reference herein.

The system cache unit **17** represents the sole connection between the main memory unit **16** and the remainder of the system and consists of a memory system port **38** for connection to the main memory and two requestor ports, **17A** and **17B**, as discussed above, one intended primarily for handling CPU requests and one intended primarily for handling I/O requests. The system cache board also provides a direct access path **39** between the I/O port and the memory system port providing for direct block transfers therebetween. Cache board **17** also includes a 16-kilobyte, direct mapped high speed cache data store **40** having a block size of 16 bytes which can be accessed from either the I/O or the CPU requestor port. Block diagrams of the logic utilized in the system cache unit **17**, the bank controller unit **18** and a typical memory module of the main memory unit **16** are shown in FIGS. **2**,**3**, and **4**.

As can be seen in FIG. **2**, the system cache data store **40** receives all requests for data from the memory other than block transfer requests from the I/O port which are serviced by the main memory directly. In the particular embodiment described, the cache data store receives the data address at the address input of either CPORT **17A** or IPORT **17B** which address is placed in either CPORT address register **41** or IPORT address register **42**. The incoming address includes a Tag portion, an Index portion and a word pointer portion as follows:

9 18	19 28	29 31
TAG	INDEX	WP

The three least significant bits **29–31** of the cache data store address specify the word pointer, which identifies the desired, word within a block of the 16 byte 8 word block of the data store. The remaining bits **9–28** identify the block address which corresponds exactly to the address which would be used to fetch the desired block from the main memory. The latter bits are divided into Tag bits **9–18** and Index bits **19–28** as shown.

The system cache as depicted in FIG. **2** includes a "Tag " Store Unit **43**. Data store **40** is a high speed memory array of 4K×32 bit words (i.e. 1K 16-byte blocks) and holds a copy of a block of words from main memory. The data store is addressed by the index and word pointer bits of the cache data store address word, the index being a 10-bit address of a block within the data store **40** and the three word pointer bits pointing to the desired word within the selected block, as mentioned above. A data store block may be used to buffer any data block of main memory which shares the same index.

The function of the Tag store **43** is to identify which of the many possible blocks from the main memory is buffered in each 16 byte block of the data store **40**. Tag store **43** is a high speed array of 1K 12-bit words and is addressed by the 10-bit index portion of the memory address. Each 12-bit word contains ten bits which identify the block from the main memory which is buffered in data store **40**. When the main memory is 4 megabytes or less, the first two bits of this tag are needed only for future expansion of the main memory capacity and can be zero. Bits **10** and **11** are flags to indicate the status of the data. Thus a "valid" flag V indicates that the indentifiable data store block contains valid data. For example, if an I/O port operation were to request a block "write" to main memory which modifies the contents of a block which has already been buffered in the data store **40**, the valid flag of that block would be reset to indicate that its data is no longer valid.

A "modify" flag M indicates that the contents of the data store block have been mofidied. Thus, if a data block is removed from the data store **40** to make room for a new data block from main memory, the removed data block is written back to main memory if the modified data flag is set.

A second tag store unit **44** is shown on the system cache board, which latter tag store is a replica of the instruction cache (ICACHE) tag store which is described later. The ICACHE tag store is used on the system cache board to determine when a write to memory would affect the contents of the instruction cache at the instruction processor. When such an effect would occur, as indicated by a comparison at comparator **45** of the incoming address and the ICACHE addresses, the system cache alerts the instruction processor by asserting an "instruction cache write" signal, as indicated in FIG. **2**, to inform the instruction cache (ICACHE) at the instruction processor board of the location of the block which has been so modified.

In the operation of the system cache all requests are initially assumed to be "read" requests, since even when a "write" request occurs it is possible that the data to be written will need to be read and modified (a "read-modify-write" operation) before the write operation is to be performed. If the system cache is not busy when a request is received at an input port, the data store **40** and the tag store **43** are accessed simultaneously, using the appropriate portions of the received input address as discussed above. The data from the location in the data store **40** which has been addressed is loaded into the cache write data register **46** via multiplexer **48** if the data transfer is a write into memory operation so that in the next cycle the contents of the write data register **46** can be enabled onto the bus via multiplexer **47** and bus driver unit **49**. If the data is a read operation the data output from data store **40** is supplied at the CPORT or IPORT, as required, via multiplexer **48** and driver units **50** and **51**, respectively.

The data from the tag store **43** is first examined to determine if the requested data, is, in fact, in the data store **40**. The tag portion of the word which is read from the tag store is compared at comparator **52** with the tag portion of the address which has been submitted by the requestor and the valid flag checked to see that it is set. If such comparison is successful (a system cache "hit") the data from data store **40** is the desired data and the requester is permitted to receive it or to write it into memory. If the comparison fails (a system cache "miss") the data block which has been requested is not in the cache data store **40** and must be brought in from the main memory. Such an occurrence is termed a "cache fault" condition and, when such fault occurs, the requestor is prevented from loading in data until after the fault is resolved.

Once the data is available for the requestor the requestor must signal that it wishes to accept the data and, if the requestor does not do so when the data first becomes available, the read operation will be repeated until the requestor indicates its willingness to accept the data.

Because access to the data in data store **40** requires two system clock cycles to complete, the cache addresses as received from requestors can be "pipe-lined" in a manner such that two accesses can be in progress at any one time. Advantage is taken of this ability to pipeline access requests by intertwining the accessors of one of the input ports with those of the other input ports. An appropriate clocking signal, which has a frequency one-half that of the basic system clock, is used to indicate which requestor port is allowed to access the cache data store at any given time. As a result there is no interference between CPU and I/O port accesses except during a cache fault. The only exception is that both I/O and CPU ports are not allowed to be in the process of accessing the same data store block a the same time. An example of the intertwining operation between the ports for a read operation is discussed below. In the particular example described the CPU port requestor does not choose to take the data at the first opportunity so that a read repeat occurs.

t0	t1	t2	t3	t4	t5
CPU PORT READ	Address and START Signal on bus.	Tag and Data Stores read.	Data ready. Requestor does not assert RT Signal.	Data Store read again.	Data Ready. Requestor asserts RT Signal and loads data.
IO PORT READ	Idle cycle or end of last access.	Address and START Signal on bus.	Tag and Data Stores read.	Data ready. Requestor asserts RT Signal and loads data.	Idle cycle or start of next access.

For a cache write operation, the cache, at the time the memory write access is initiated, assumes that a read-modify-write operation will be performed and accordingly does a read as described above. However, even if the transfer is to be a simple write operation, the tag store information must be read to determine the location at which the incoming data will be written so that in actuality no time is lost in performing a superfluous data store read operation. For a simple write operation, or for the write portion of a read-modify-write operation, the requestor asserts a write transfer (WT) signal to indicate completion of the transfer. Instead of driving the data from the output register onto the memory port **38** the system cache loads an input register **53** with the data which is to be written from the data bus at the end of the cycle and writes it into the data store **40** during the next cycle. If a cache fault results from such a write request, the system cache accepts the data to be written into the input register but does not write it into the data store **40** until after the fault is resolved. An example of a CPU port write request in a manner similar to that discussed above for a read request is shown below.

t0	t1	t2	t3	t4	t5
CPU PORT WRITE	Address and START and WRITE Signals on bus.	Tag and Data Stores read.	Data ready. Requestor asserts WT Signal and sends data.	Data Store written.	Idle cycle.
IO PORT READ	Idle cycle or end of last access.	Address and START Signal on bus.	Tag and Data Stores read.	Data ready. Requestor asserts RT Signal and loads data.	Idle cycle or start of next access.

The examples discussed above show a single read or single write operations. It is also possible for a requestor to submit a new address and a START signal along with the read transfer (RT) and/or write transfer (WT) signal, so that consecutive read operations or consecutive write operations from a single port can be performed every two cache cycles (a CPU cycle, for example, is equivalent to two cache cycles) unless a cache fault occurs. However, if a read access is initiated at the same time that a write transfer is performed, the data store **40** cannot be read on the next cycle because it is being written into at that time. When this condition happens, the read operation requires an additional two cache cycles for completion. If the requestor is aware that a read operation is following a write transfer and wishes to avoid a wasted cycle, the requestor can either delay starting the read request until the next cycle or it may start the read request to wait an extra cycle before requesting the data transfer. In either case useful work could be done in the otherwise wasted cycle, although initiation of a read followed by a wait for an extra cycle is usually more desirable because it allows a cache fault to be detected at an earlier point in time.

A read-modify-write operation can be accomplished by asserting a START signal and WRITE signal along with the address, followed by a read transfer at a later cycle and a write transfer at a still later cycle. When a WRITE signal is signaled at the start of an access, the system cache will not consider that the access has been completed until a write transfer is performed. During such operation all other requestors are prohibited from accessing the same data. Thus, requestors utilizing the same input port are prevented from access by the fact that the first requestor controls the bus during the entire read-modify-write operation. Requestors on the other port are prevented from access by the fact that both ports are prohibited from accessing the same data store block at the same time. Such prohibition also prevents requestors at another port from removing a block of data from the cache data store when the system cache is in the middle of an operation.

If the system cache board receives a write transfer request when a write operation has not been previously

indicated or, if it receives a read transfer and a write transfer request simultaneously, access to the system cache data store is aborted without the transfer of any data. If such simultaneous read and write transfer requests are asserted at the beginning of the next cycle after the START request, the access may be avoided without even initiating an unnecessary cache fault indication.

In addition to the above transfers, the system cache board has the capability of performing direct write transfers between the input ports and the main memory, the bulk of such data traffic being capable of being handled without affecting the contents of the cache data store **40**. if the requested transfer is a block write transfer, the data is written directly into the main memory via data write register **40A**, MUX **48** and write data register **46**. Data transfers at the I/O port are not allowed when the CPU port is in the process of accessing data which has the same Index as the I/O block which is to be transferred. Data read-modify-write transfers are also not permitted by the system.

In the overall system cache block diagram shown in FIG.2, the input registers for the CPU request port and the I/O request port are shown as data registers **54** and **55**. Addresses associated with the data at such registers are supplied to the CPU address register **41** and the I/O address register **42**, each address comprising the Index, Tag and Word Pointer as discussed above.

Specific logic diagrams of the system cache board **17** depicted in FIG. **2** are shown in FIGS. **5–44**, which latter figures are appropriately labeled as follows to show more specifically a particular embodiment of the various portions of the system cache **17** depicted therein.

FIG. **5** shows the cache data store **40**; FIG. **6** the Tag store **43**; FIG. **7** the ICACHE tag store copy unit **44**; FIG. **8** the tag store comparator **52**; FIG. **9** the ICACHE tag store comparator **45**; FIG. **10** the CPORT and IPORT registers **41** and **42** and the write back tag unit; FIGS. **11** and **12** the INDEX SV WP SV unit of FIG. **2**; FIG. **13** the INDEX and WP multiplexer units; FIG. **14** data write register **40A**; FIG. **15** the multiplexer unit **48** and the index driver unit **48'** which supplies an input to multiplexer **48**; FIG.**16** the write data register **46**; FIG. **17** the multiplexer unit **47**; FIG. **18** the driver units **50** and **51** and driver logic associated therewith; FIG. **19** the INDEX/INDEX SV comparator unit; FIG. **20** the CPU buffer data register **54**, the I/O buffer data register **55**, and the CRD IN register **53**. The specific system cache parity logic is shown in FIGS. **21–25**. The main memory and other interface control logic is shown in FIGS. **26–28**. As in any data processing system board, adequate control signals for the various units thereon must be provided and control logic for the particular embodiments of the system cache board depicted in FIGS. **5–27** are shown in FIGS. **29–43**.

FIG.3 depicts an overall block diagram of the bank controller **18** which interfaces between the system cache at the left hand side of the drawing and the memory modules at the right hand side thereof. Words which are read from the memory modules, identified as RD **Ø-38**, including 7 parity bits, are supplied to the bank controller for transfer to the system cache, such words being identified ad CRD **Ø-31** in FIG. **3**, via the error correction logic **70** which also supplies four parity bits, identified as CRD PAR **Ø-3**. Address and data words which are to be written into the main memory modules are supplied from the system cache such words being identified as CA/WD **Ø-31**, together with the parity bits therefor, identified as CA/WD PAR **Ø-3**, the data being supplied to the write data bus for the memory modules WD **Ø-31** and parity bits WD **32-38** via error correction logic **70**. The addresses therefor are supplied in the form of information which is required to select the desired memory module (MODSEL **Ø-3**) to identify up to 16 modules) and to select the desired RAM within the selected module (ADDRØ-7)

Further, the bank controller supplies the following control signals to the main memory which responds thereto as required. The RAS and CAS signals represent the row address and column address strobe signals for the RAM's of the main memory. The LDOUT signal causes the selected module to load its output register at the end of the current cycle and to enable the register to place the contents of the output register on the read data bus during the next cycle. The LDIN signal causes the selected module to accept data from the write bus during the next cycle and to write such data into the RAMs during the following cycle. The REFRESH signal overrides the module selection for the row address strobe (RAS) signal only. During a refresh operation one module is read normally and all others perform an RAS refresh only.

The bank controller also interfaces the system cache to supply 32-bit words (CRD **Ø-31**) to the cache along with 4 parity bits (CRD PAR **Ø-3**) for byte parity and to receive 32 bit address and data words (CA/WD **Ø-31**) from the cache along with byte parity bits (CA/WD PARØ=3). The bank controller also supplies the following control signals to the cache. The BC BUSY signal indicates that the bank controller is not able to accept a BC START (see below) request. The BC ERROR signal indicates that the data word placed on the read data bus during the last cycle contained a correctable error and must be replaced with the corrected word for the data which is on the bus during the current cycle. Once a BC ERROR signal has been asserted all subsequent words of the same block transfer are also passed through the error correction logic. Accordingly, BC ERROR need be asserted only once for each block transfer.

The BC DATABACK signal indicates that the first word of the four word block to be transferred will be at the read data bus in the next cycle. The BC REJECT signal indicates that the bank controller cannot accept the contents of the write data bus at the end of the current cycle. The BC START indicates that a bank controller transfer operation is to commence.

Specific logic diagrams for the particular units of the bank controller board **18** of FIG. **3** are shown in FIGS. **44–63**, which latter figures are appropriately labelled as follows to show more specifically a particular embodiment of the various portions of the bank controller **18** depicted therein.

The error correction logic **70** is shown in FIGS. **44–63** and includes the multiplexer store unit shown in FIG. **44**; the C-bit generator unit **45**; the (32 bits) register and (8 bits) register shown in FIG. **46**; the drivers for the write data bus shown in FIG. **47**; the S-bit generator shown in FIG. **48**. The read save register shown in FIG. **49**; the S save register shown in FIG. **50**; the read parity save register and parity logic shown in FIG. **51** and the correction logic shown in FIG. **52**. The direct read driver unit is shown in FIG. **53**.

With reference to the control units at the lower part of FIG. **3**, the R/W module selection unit and the

RADDR and CADDR units are shown in FIG. **54**; the MODSEL unit and drivers therefor are shown in FIG. **55**; and the ADDRESS unit and driver therefor are shown in FIG. **56**.

Appropriate timing and control logic both for address and data transfer and for memory refresh operation is shown in FIGS. **57–59**, the drivers for the principal control signals supplied to the memory module being shown in FIG. **60**; and various bus interface logic as shown in FIGS. **61–63**.

FIG. **4** depicts the overall block diagram for a typical memory module of the main memory system of the invention and shows the memory array **60** of dynamic NMOS random access memories (RAM's) organized as four planes of 16K 39-bit words each and identifiable as planes **Ø-3**. A word which is to be written into the memory array is received from the bank controller as $\overline{\text{WDØ-38}}$ via buffer **62**. Words being stored in even planes **Ø** and **2** are stored in even plane data register **63** while words to be stored in odd planes **1** and **3** are stored in odd plane data register **64**. The control signals are supplied from the bank controller to control logic **65**. The module select code bits MOD SELØ-3 are supplied to a comparator **66** to provide a MODSEL signal if the particular module has been selected. Control signals from control logic **65** are supplied to appropriate latching circuitry **67** to provide appropriate signals for controlling the operation of the memory array via drivers **61**. The control signals from the memory bank controllers are first clocked into the plane **Ø** latching registers **67A** and the outputs thereof drive the plane **Ø** RAMs via drivers **61A**. The outputs of the first latch register are those clocked at a fixed time period later into the next latch register set **67B** which drives the plane **1** RAMs. Such pipeline operation continues in order to drive the plane **2** and plane **3** RAMs such that all four RAM planes receive the same control signals at fixed intervals, resulting in the transfer of a block of four consecutive 39-bit words. While the RAM address from the bank controller includes eight bits, only seven bits of address are used for the 16K RAMs discussed above, the extra bit allowing for possible future expansion. Thus, addressed bits ADR **Ø-5** are clocked at fixed intervals to each of the latches **67A-67D** of the planes **Ø-3** at fixed intervals. ADR **6** is supplied to RAM selection logic **68** together with the plane **Ø** latch signal $\overline{\text{RPL Ø RAS}}$ to provide the JADR **6** signal for the plane **Ø** latch register **67A**. The RAS and CAS signals provide the necessary control signals via the control logic **65** and latch registers **67** for driving the row address strobe (RAS) and the column address strobe (CAS) signals for the RAMs.

The LDOUT signal to the input of control logic **65** causes the module to load its output register at the end of the current cycle and enable it onto the read data bus during the next cycle via the data out register and multiplexer logic **69** and read bus driver **69A**. The LDIN signal at the input to control logic **65** causes the module to accept data from the write data bus via registers **63** and **64** for writing into the RAM during the following cycle. The following timing diagrams show the status of the various signals for block read and block write operations at each fixed time interval (in the particular embodiment described, for example, each cycle can be 110 ns). As can be seen, the plane **Ø-3** data is provided in the read operation in sequence and the input data is written into such planes in sequence.

			Block Read				
t0	t1	t2	t3	t4	t5	t6	t7
Control Signals	RAS MODSELS	RAS,CAS MODSELS	RAS,CAS MODSELS	LDOUT MODSELS	<pre-charge>	<next access>	
Address Lines	ROW ADDRESS	COLUMN ADDRESS	COLUMN ADDRESS				
Read Data bus					PLANE DATA	PLANE DATA.	<etc.> <etc.>
Write Data Bus							

			Block Write				
t0	t1	t2	t3	t4	t5	t6	t7
Control Signals	RAS,LDN MODSELS	RAS,CAS MODSELS	RAS,CAS MODSELS			<next access>	
Address Lines	ROW ADDRESS	COLUMN ADDRESS	COLUMN ADDRESS				
Read Data Bus							
Write Data Bus		PLANE 0 DATA	PLANE 1 DATA	PLANE 2 DATA	PLANE 3 DATA		

More specific detailed logic circuitry for implementing the units shown in the block diagram of FIG. **4** to achieve the desired operation as described above are shown in FIGS. **64–78**. Data in registers **63** and **64** are shown in FIGS. **64** and **65**, respectively. The memory array **60** is shown in FIGS. **66–73** wherein plane **Ø** RAMs and the control input circuitry therefor are shown in FIGS. **66** and **67**; plane **1** RAMs and the control input circuitry therefor are shown in FIGS. **68** and **69**, plane **2** RAMs and the control input circuitry therefor are shown in FIGS. **70** and **71**, and plane **3** RAMs and the control input circuitry therefor are shown in FIGS. **72** and **73**. The data out register and multiplexer unit **69** are shown in FIG. **74**. Latching and driver logic is shown in **75**. The RAM select logic unit ($\overline{\text{RAMSEL}}$ LOGIC) is shown in FIG. **76**, while the $\overline{\text{MODSEL}}$ comparator unit **66** and the various control logic units and latching circuitry associated therewith and with the input control signals from bank controller unit **18** are shown in FIG. **77**. Memory module timing logic is shown in FIG. **78**.

ADDRESS TRANSLATION UNIT

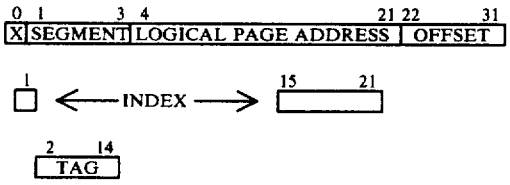
The address translation unit (ATU) **14** is shown broadly in FIGS. **79–81**, the primary function of such unit being to translate a user's logical address (LA) into a corresponding physical address (PA) in the physical

address space of the processor's memory modules discussed above. Such translation is effectively performed in two ways, one, by accessing a page from the system cache or from main memory at the particular page table entry specified in a field of the logical address and placing the accessed page in a translation store unit for use in performing the address translation, a sequence of operations normally designated as a Long Address Translation (LAT) and, the other, by accessing additional references to a page that has already been selected for access after an LAT has been performed and the page selected by the LAT is already present in the translation store. The latter translation provides an accelerated address reference and can be accomplished by saving, at the end of every Long Address Translation, the address of the physical page which has been accessed. As mentioned, the physical page involved is stored in a high speed random access memory (RAM) file designated in FIG. **79** at ATU translation store **100**.

Translations of addresses on the physical page which is stored in the ATU translation store **100** are available to the processor within one operating time cycle of the CPU, while normally the Long Address Translation will take a plurality of such cycles for a reference which requires a single level page table reference (e.g. 3 cycles) or a two-level page table reference (e.g. 5 cycles), where the page in question is available in the system cache memory. Even longer times may be required if the page involved can not be found in the system cache memory and must be accessed from main memory.

A secondary function of the ATU is to emulate all operations of the previous system of which the present system is an extension, e.g., in the system described, to perform all Eclipse memory management processor unit (MMPU1) address translation operations, as described in the above referenced publication for such systems, in an efficient and compatible way, such emulated operations being described in more detail later.

In order to understand more clearly the translation of a logical word address (a byte address when shifted right by one position produces a word address), the logical word address can be defined as shown below:



As seen therein, the segment and logical page address is 21 bits long, the segment and logical page address being divided into two fields, the Tag field and the Index field. The Tag field is defined as bits LA **2-14** while the Index field is defined as bit LA **1** plus bits LA **15-21**.

As seen in FIG. **79**, when a logical word address LA**Ø-31** is received from the arithmetic logic unit (ALU) on the logical address bus **26** it is latched into a logical address register (LAR) **101**. The Index bits LA **15-21** are taken directly from the logical address bus to address four RAM stores, the first being a Tag store **102**, which retains the tag portions of the logical addresses corresponding to the physical addresses saved in the ATU physical address (PA) translation store **100**. The Index bits LA **15-21** are also supplied to a validity store RAM unit **103** and to a protection store RAM unit **104**, as discussed below.

If the physical address translation store **100** contains valid address translations, when a memory access is started the logical address is loaded into the logical address register **101** and the Index (bits LA **15-21**) is used to select a location in the store.

In the particular system described, even though there is a valid address translation at such location in translation store **100**, it may not be the correct one. Corresponding with each index of the logical addresses (and each address location in the translation store) there are a selected number of possible "tags", each tag corresponding to a unique physical page address. Only one of such tags and its corresponding physical page address can be saved in the translation store **100** at the location selected by the Index. Therefore, the "tag" (TAG **2-14**) that corresponds to the Index in question and is currently stored in the tag store **102** is compared at comparator **105** to the "tag" in the logical address register (LA **2-14**). If the "tags" correspond, the address translation contained in the translation store **100** is the correct one and can be used to supply the desired physical address (signified by an ATU HIT signal at the output of comparator **105**). If they do not match, a Long Address Translation operation must be performed to obtain the desired physical page address from the system cache or main memory. The physical page address which is thereby accessed by such LAT procedure to replace the physical page address previously contained in the ATU translation store **100** is placed on the appropriate transfer bus (CPM bus **20**). At the completion of the long address translation, the "tag" taken from the logical address register (LAR **2-14**) is written into the tag store **102** at the location selected by the index and the physical page address from the memory data register **106** (MD **18-31**) is written into the translation store **100** at the location specified by the index.

The ATU configuration shown in FIG. **79** also contains further components which are used to place the translated physical address of a desired physical page table on the physical page address (PA) bus **27**. There are three other physical sources of physical page table addresses, the first of which is bits SBR **18-31** of a segment base register which segment base register can also be located in scratch pad units of the address translation unit. This address is used to reference either a high order page table (HOPT) of a two-level page table or the low order page table (LOPT) of a one-level page table. Since the segment base registers are located at the ATU, such address can be obtained from the logical address bus **26** as LA **18-31**.

The following diagrams depict the results of the control actions initiated by the arithmetic translation unit (ATU) to perform a long address translation in which a physical address is derived from a logical address by traversing the one-and two-level page tables in the main memory. Diagram A depicts a one-level page table traversal, while Diagram B depicts a two-level page table traversal, the physical address bits **3-21** of the final physical address (i.e., the desired memory allocation data) being placed in the translation store **100** so that when the corresponding logical address is subsequently requires a translation, the physical address is available (an ATU HIT occurs) and there is no need for subsequent long address translation.

The logical word address to be translated for a one-level page table translation has the format shown in

FIG. 157. Bits 1-3 of the word address specify one of the eight segment base registers (SBRs). The ATU uses the contents of this valid SBR to form the physical address of a page table entry (PTE), a shown at point ① of the diagram.

The selected SBR contains a bit (bit 1) which specifies whether the page table traversal is a one-level (bit 1 is zero) or a two-level (bit 1 is a one) page table. In Diagram A a page table entry address comprising the starting address of a selected page table and page table entry offset specifying a page address therein.

To form this physical page address, the ATU begins with the physical address as shown at ② of the diagram. This address becomes bits 3-21 of the PTE address. Bits 13-21 of the logical word address become bits 22-30 of the PTE address. The ATU appends a zero to the right of the PTE address, making a 29-bit word address.

Bits 3-21 of the PTE address (unchanged in the step above) specify the starting address of a page table. Bits 22-31 of the PTE address specify an offset from the start of the table to some PTE (labelled PTEn in Diagram A). This PTE specifies the starting address of a page of memory, as shown at ③ of the diagram.

PTEn bits 13-31, the page address, becomes bits 3-21 of the physical address, as shown at ④ of FIG. 157. The page offset field specified in bits 22-31 of the logical word address becomes bits 22-31 of the physical address. This is the physical word address translated from the original word address. The physical address bits 3-21 are placed in the translation store as the memory allocation data for subsequent use if the same logical word address requires subsequent translation. It should be noted that when using a one-level page table, bits 4-12 of the logical word address must be zero. If they are not zero and bit 1 of the SBR indicates a one-level page table is required, a page fault occurs.

Just as in the one-level page table translation process, in the two-level page table translation depicted in FIG. 158, the processor produces a physical address. The logical word address to be translated has the format shown in the diagram, the steps (1) through (4) being substantially the same as in FIG. 157 except that bis 4—12 of the logical word address become bits 22-30 of the PTE address. The ATU appends a zero to the right of the PTE address, making a 29-bit word address. Bits 1-3 of the word address specify one of the eight segment base registers (SBRs).

Bits 3-21 of the PTE address specify the starting address of a page table. Bits 22-31 of the PTE address specify an offset from the start of the table to some PTE (labelled PTEn). The PTE specifies the startion address of a page table. Thus, the ATU now constructs the address of a second PTE from the address at ④. The physical address specified in bits 13-31 of the first (PTEn) becomes bits 3-21 of the address of the second PTEm. Bits 13-21 of the logical word address becomes bits 22-30 of the second PTE's address. The ATU appends a zero to the right of the second PTE address to make a 29-bit word address.

Bits 3-21 of the second PTE address specify the starting address of a second page table. Bits 22-31 of the second PTE address specify an offset from the start of the second table to some PTE (labelled PTEm in FIG. 158). The second PTE specifies the starting address of a page, as shown at ⑤ in FIG. 158.

The second PTEm's bits 13-31, the page address, become bits 3-21 of the physical address and the page offset specified in bits 22-31 of the logical word address becomes bits 22-31 of the physical address, as shown at ⑥ in FIG. 158. This last value is the final physical word address.

The physical page table address for the low order page table of a two-level page table is in bits 18-31 of the high order page table entry (HOPTE) which must be fetched from the main memory. Thus, the second possible source of the physical page table address is the memory data register (MD) 105 which holds the data that arrives on the physical memory data (CPM) bus 20 as $\overline{\text{MD 18-31}}$. A suitable page table multiplexer 107 is used to select which of the two sources will drive the physical address bus when its outputs are enabled.

The third and final source is to drive the physical page address bus 27 directly through a physical mode buffer 108, such buffer being used to address physical memory directly ($\overline{\text{PHY 8-21}}$) from bits $\overline{\text{LA 8-21}}$ of the logical address bus. Such buffer is enabled while the ATU unit is turned off (i.e., no address translation is required) since the physical address in that mode is the same as the logical address and no translation is necessary.

Bits $\overline{\text{PHY 22-31}}$ of the physical address are offset by displacement bits, there being three possible origins for the offset. The first source of such offset is from bits $\overline{\text{LA 22-31}}$ of the logical address bus which bits are used while in physical mode (no address translation is necessary) as well as the offset in the object page. The second source of the offset is bits $\overline{\text{LAR 4-12}}$ (referred to as two-level page table bits in Diagram B above) of the logical address register which is used as an offset within the high order page table during a long address translation. Since this source is only nine bits long and page table entries are double words aligned on even word boundaries, a ten bit offset (to form $\overline{\text{PHY 22-31}}$) is constructed by appending a zero bit to the least significant bit. The final source for the offset is bits $\overline{\text{LAR 13-21}}$ (referred to as one-level page table bits in FIG. 158 above) of the logical address register which is used as an offset within the low order page table during the long address translation. A zero bit is appended to the least significant bit of this source also. Offset multiplexer 109 is used to select the desired one of such three offset sources.

The following discussion summarizes the address bit sources for forming a low order or high order page table entry address in main memory in making a long address translation. The address of the page table entry is formed from address fields in a segment base register (SBR) and from address fields in the logical address register. The address fields of a segment base register can be depicted as follows:

0	1	2	3	4	12 13	31
V	L	LEF	I/O	RESERVED	PHYSICAL ADDRESS	

SEGMENT BASE REGISTER

Depending on whether a one-level (low order) or a two-level (high order) page table entry is called for, the SBR address field comprising bits 4-12 or the SBR address field comprising bits 13-21 is transferred to the memory data register 105 to form the higher order bits of the page table entry. As mentioned above, the eight SBR registers are located in 8 of the 256 locations of scratch pad registers on the ATU. This use of such scratch pad locations for the segment base registers can

be contrasted with prior known systems wherein the segment base register (or registers comparable thereto) in a segment, or ring, protection memory system as all located at specified locations in the main memory. By placing them in a scratch-pad memory located in a processing unit of the system, as in the ATU unit here, the higher order page table entry bits are acquired more rapidly than they would be if it were necessary to fetch them from main memory and, hence, the speed at which page table entries can be made is improved considerably.

One of the bits of an SBR (identified above as "V" bit) is examined to determine whether the SBR contents are valid. Another bit (identified above as "L" bit) is examined to determine whether a 1-level or a 2-level page table entry is required so that the correct field is supplied to the memory data register.

Other bit fields of the SBR are used to determine whether a Load Effective Address (LEF) instruction (such LEF instruction is part of the Eclipse instruction set as explained more fully in the above cited publications therein) or I/O instruction is required. Thus in a selected state the LEF Enable bit will enable an LEF instruction while a selected state of the I/O Protect bit will determine whether an I/O instruction can be permitted. The remaining field of the SBR contains the address offset bits.

As is also seen in FIG. **79** a variety of protection checks are made for each reference to memory, which protection checks are made by the use of protection store unit **104**, protection logic unit **110** and ring protection logic unit **111** for providing appropriate fault code bits (FLTCD Ø-3) which are supplied to the microsequencer (described below) via driver **112** on to the CPD bus **25** for initiating appropriate fault micro-code routines depending on which fault has occured.

The following six protection checks can be made:

1. Validity storage protection
2. Read protection
3. Write protection
4. Execute protection
5. Defer protection
6. Ring maximization protection

A validity storage protection check determines whether the corresponding block of memory to which a memory reference is made has been allocated and is accessible to the current user of the system. The validity storage field is a one-bit field which is located, for example, at bit zero of each of the segment base registers (located on an ATU board as discussed above) or at bit zero in each of the high order page table entry addresses and low order page table entry addresses. In a particular embodiment, for example, a "1" indicates that the corresponding block has been so allocated and is accessible whereas a "0" indicates that the user cannot use such a memory block.

Generally when a new user enters the system all pages and segments in the logical address space which are allocated to that user, except those containing the operating system, are marked invalid. Validity bits are then set valid as the system begins allocating logical memory to such new user. If a user makes a memory reference to an invalid page, an invalid page table, or an invalid segment, the memory reference is aborted and a validity storage protection error is then signaled by the fault code bits on the CPD bus.

The read protection field is a one-bit field normally located at a selected bit (bit **2**, for example) in each of the low order page table entry addresses and a check thereof determines whether the corresponding object page can or cannot be read by the current user. If the page cannot be read, a rear error is signaled by the fault code bits on the CPD bus. In a similar manner a check of the write protection error field determines whether the corresponding object page can be written into by the current user, an appropriate write error being signaled by the fault code bits if the user attempts to write into a page to which he is not allowed.

The execute protection field is a one-bit field which is located at a selected bit (e.g. bit **4**) in each of the low order page table entry addresses and a check thereof determines whether instructions from a corresponding object page can or cannot be executed by the current user. If such an instruction fetch is not allowed, an execute error is signaled by the fault code bits on the CPD bus. Execute protection is normally checked only during the first fetch within a page and any additional instruction fetches are performed using the physical page address from the first fetch, which for such purpose is retained by the instruction processor.

When a user is attempting to reference a location in memory and is utilizing a chain of indirect addresses to do so, the system will abort the operation if a chain of more than a selected number of said indirect addresses is encountered. For example, in the system under discussion if a chain of more than sixteen indirect addresses is encountered the operation is appropriately aborted and a defer error is signaled by the fault code bits on the CPD bus. Such protection is utilized, for example, normally when the system has performed a loop protection and the system, because of a fault in the operation thereof, continues to repeat the indirect loop addressing process without being able to break free from the loop operation.

Ring maximization protection is utilized when the user is attempting to reference a logical location in memory in a lower ring (segment) than the current ring of execution (CRE **1-14 3**). Since such operation is not permitted by the system, the operation must be aborted if the user attempts to reference a lower ring than currently being used and a ring maximization error is signaled on the CPD bus. Since the logical address space is divided into eight rings, or segments, a ring which the user desires to reference can be indicated by bits **1–3**, for example, of the logical address.

The specific logic circuitry utilized for such protection checks (i.e., the protection store **104** and the protection logic **110** and the protection logic **111** associated therewith) is shown in FIGS. **80** and **81**. Thus, logic for the generation of the read error, write error, execution error and validity error signals is shown in FIG. **80** and logic for generating the defer error and ring maximization error signals being shown in FIG. **81**.

With respect to the protection system, since logical address space is partitioned into eight hierarchical regions (i.e. the "rings" or "segments") the partitioning can be delineated by the segment field of the logical address. Thus, segment number **0** is always assigned to ring **0** (ring **0** being the ring in which only priviledged instructions can be executed), segment **1** is always assigned to ring **1**, and so forth. Such approach differs from previous systems using a segmented hierarchical address space in that the ring number is not independent of the logical address sace. In contrast, in the system discussed here, each ring is directly bound in the space

so that segment **0** is always allocated to ring **0**, segment **1** to ring **1**, and so forth.

The access field in a page table entry comprises three bits (MD **2-4**) is shown in FIG. **79** and indicates the capabilities o the referenced data item in the logical address space, i.e. as to whether the reference data item is to be a read access, a write access, or an execute access, the protection store **104** responding to such bits to produce either a read enable signal ($\overline{RD\ ENB}$), or a write enable ($\overline{WR\ ENB}$) or an execute enable ($\overline{EX\ ENB}$). The ring protection governs the proper interpretation of the access privileges of the user to a particular ring, a user being permitted access only to selected, consecutively numbered rings. Thus, access can only be made to a bracket of rings (an access bracket) if the effective source for such reference is within the appropriate access bracket. For example, the read bracket of a data reference in any ring is the ring number. That is, a data address reference to segment **5** (ring **5**), for example, can never legitimately originate from an effective source which is greater than 5. In other words an effective source in segment **5** can never reference a ring lower than ring **5** and, therefore, if a reference from an effective source greater than 5 attempts to access ring **5** a ring maximum error (MAX ERR) will be signaled as shown by the logic in FIG. **13**. A table showing such ring protection operation is shown below:

Effective Source Space	Target Space RING 0	RING 1	Ring 2	...	RING 7
RING 0	Val-R0	Val-R1	Val-R2	...	Val-R7
RING 1	Fault	Val-R1	Val-R2	...	Val-R7
RING 2	Fault	Fault	Val-R2	...	Val-R7
.	.	.	.		.
.	.	.	.		.
.	.	.	.		.
RING 7	Fault	Fault	Fault	...	Val-R7

In summary, in order to make a ring access, the ring maximization function is used to determine whether or not the reference is a valid ring reference and, if it is, the page table entry that references the address datum is examined to see if the page is a valid one. Then, if the read protection bit indicates that such valid page can be read, the read can be performed. If any one of the examinations shows a protection error (i.e., ring maximization error, validity error, or read error) the read is aborted and an appropriate fault code routine is called. Similarly, appropriate examination for protection errors for write access and execute access situations can also be performed.

In an hierarchical address space such as discussed above, it is desirable to mediate and authenticate any attempt to switch rings, i.e., to obtain access to a ring (segment) other than the ring which is currently being used (a "ring crossing" operation). The performing of a ring crossing operation is authenticated as follows.

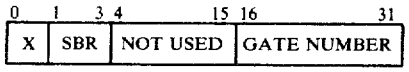
Any ring crossing attempts occur only as a result of an explicit attempt to do so by a program control instruction, and such explicit attempt can occur only if the following conditions are satisfied.

(1) The program control instruction is of the form of a subroutine "call", i.e., where access is desired to a subroutine in another ring (LCALL - see Appendix B), or a subroutine "return", i.e., where a subroutine in another ring has been accessed and it is desired to return to the original ring (WRTN and WPOPB - see Appendix B). All other program control instructions (e.g., JUMP) ignore the ring field of the effective address required for the instruction and such instructions can only transfer to locations within the correct segment.

(2) The direction of a subroutine call crossing must be to a lower ring number (i.e., inwardly toward ring **0**) wherein the lower ring has a higher order of protection and the current ring of execution and the direction of a subroutine return crossing must be to a higher ring number (i.e., outwardly away from ring **0**) wherein the higher ring has a lower order of protection than the called ring containing the subroutine. Outward calls and inward returns are trapped as protection faults.

(3) The target segment of the effective branch address is not in the segment identified by bits **1-3** of the program counter.

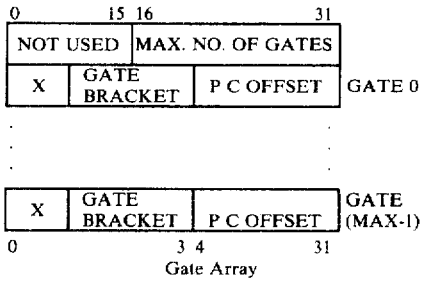
In the above conditions are met the return address for outward returns is merely interpreted as a normal word address. However, if the above conditions are met for an inward call, the branch address is interpreted as follows:



Inward Call Branch Address

Bits **16-31** are interpreted as a "gate" into the specified segment (SBR of bits **1-3**) in the target space. The gate number is used to verify that the specified gate is in the called segment and, upon verification, to associate on instruction address with the specified gate via a "gate array" in the called segment, as discussed below.

The location of the gate array in any called segment is indicated by a pointer contained in particular locations of the called segment (e.g., in a particular embodiment the pointer locations may be specified as locations **34** and **35** in each segment. The structure of the gate array is as follows:



Gate Array

The gate number of the pointer which referenced the target segment is compared with bits **16-31** of the first 32 bits of the gate array. If the gate number is greater than or equal to the maximum number of gates in the gate array, the ring crossing call is not permitted and a protection fault occurs (if the maximum number of gates is 0, the segment involved cannot be a valid target of an inward ring crossing call operation).

If the gate number is less than the maximum number of gates, the gate number is then used to index into one of the gates of the gate array which follows the first 32 bits thereof. The contents of the indexed gate are read and are used to control two actions. First, the effective source is compared to the gate bracket bits **1-3** of the indexed gate. The effective source must be less than or equal to the referenced gate bits and, if so, the PC offset

bits 4–31 become the least significant 28 bits of the program counter and bits 1–3 of the program counter are set to the segment containing the gate array.

If the gate in a ring crossing operation, as described above, is a permitted entry point to the ring to which the crossing is made, a new stack is constructed. In order to do so a stack switching operation must occur since there is only one stack per ring. Thus, before the new stack can be created, the contents of the current stack management registers must be saved at specified memory locations of the caller's ring. The callee's stack can then be created, the arguments from the caller's stack being copied onto the newly created callee's stack, the number of such arguments being specified by the X or the LCALL instruction (see Appendix B). An appropriate check is first made to determine whether copying of all of the arguments would created a stack overflow condition. If so, a stack fault is signalled, the ring crossing being permitted and the fault being processed in the called ring.

In order to emulate operation of ECLIPSE address translation operations appropriate emulation control signals for placing the ATU in the ECLIPSE operating mode are required as shown by emulation control logic unit **115** which, in response to coded instructions generated by the microsequencer board **13** produces such signals to permit operation for 16-bit addresses equivalent to the memory management protection unit (MMPU) of ECLIPSE comparators as described in the aforesaid publications thereon.

Specific logic circuitry for implementing the various blocks of the address translation unit shown in FIGS. **79–81** are shown in FIGS. **82–100**. FIG. **82** depicts the translation store unit **100** supplied with bits $\overline{\text{MD 18–31}}$ from the memory data register **105** and in turn supplying the translated physical address bits **8–21** which have resulted from a translation of the logical address bits $\overline{\text{LA 15–21}}$. FIG. **82** also shows the page table address multiplexer unit **107** and physical mode buffer unit **108**. In addition, such figure includes the "last block" register unit **116** which during an ECLIPSE MMPU emulation operation provides the physical address bits $\overline{\text{PHY 8–21}}$. FIG. **82** also shows the LMP Data Register. FIG. **83** shows Tag Store **102** and Protection Store **104**. Tag comparator unit **105** is depicted in FIG. **84**. FIG. **85** shows the logical address register **101**, while physical address offset multiplexer **109** and the logical address register CPD bus driver unit are shown in FIGS. **86** and **87**, respectively. The physical address bus driver units for filing the appropriate physical address bit $\overline{\text{PHY 8–21}}$ are shown in FIG. **88**.

Protection logic including fault detection and cache block crossing trap logic is depicted in FIGS. **89–92**, protection logic identification encoder unit **110** being shown in FIG. **89**, the fault code bit drive unit **112** being shown in FIG. **90**, ring protection logic circuit **111** being shown in FIG. **91** and the fault detection and cache block crossing logic being shown in FIGS. **92** and **93**.

Validity store unit **103** is shown in FIG. **94** together with translation purge logic and the multiplexer associated therewith. The translation register of FIG. **79** is depicted in detail in FIG. **95**. The reference/modify storage and control logic unit is shown in FIG. **96**, the state save drive unit associated therewith being depicted in FIG. **97**. The 16 bit MMPU emulation control logic is shown in FIG. **98**.

ATU timing logic is shown in FIG. **99** and suitable system code interface logic is shown in FIG. **100**.

INSTRUCTION PROCESSOR

The instruction processor (IP) **12** is utilized to handle the fetching and decoding of macro-instructions for the data processing system of the invention. The instruction processor operates both at and ahead of the program counter and its primary function is to provide a starting micro-address (STμAD) for each micro-instruction, which starting micro-address is supplied to the microsequencer unit **13**. Subsidiary functions of the instruction processor are (1) to provide the source and destination accumulator designations, (2) to provide the effective address calculation parameters for the arithmetic logic unit and (3) to provide sign or zero extended displacements for making memory references or for in-line literals (immediates) to the arithmetic logic unit (ALU).

As seen in FIG. **101**, the instruction processor includes instruction cache logic **120** (ICACHE), macro-instruction decoding logic **121** (which includes an instruction decode register as shown in FIG. **103**) and program counter/displacement logic **122** as described below. The ICACHE logic functions as a pre-fetcher unit, i.e., the instruction cache (ICACHE) thereof obtains a block of subsequent macro-instructions for decoding, which block has been accesses from memory while the previous macro-instructions are being executed. The ICHACHE stores the subsequent block of macro-instructions even if such macro-instructions are not immediately going to be used by the microsequencer. The decoding logic **121** of the instruction processor responds to a macro-instruction from ICACHE, decodes the operational code thereof (opcode) to provide the opcode description information for control and status logic **123** and to supply the information needed therefrom to the starting micro-address STμAD) register **124** (and thence to the microsequencer) to identify the starting micro-address of the required micro-instructions.

The displacement logic **122** supplies the displacement data to the ALU if the index for such displacement is on the ALU board. If the index for the displacement is the IP program counter, the displacement logic combines the displacement information with the program counter information available at the instruction processor to form the logical address for supply to the LA bus.

Thus, in an overall IP operating sequence, a macro-instruction is read from an ICACHE storage unit of the ICACHE logic **120** into the decode logic **121** which thereupon decodes the instruction opcode and generates the starting micro-address for the micro-sequencer. During the decoding and starting micro-address generation process, the instruction processor simultaneously reads the next macro-instruction from the ICACHE into the decode logic. While the micro-sequencer is reading the first micro-instruction, the decode logic is decoding the next macro-instruction for generating the next starting micro-address. When the micro-instruction at the starting micro-address is being executed, the micro-sequencer reads the next micro-instruction from the next starting micro-address. Accordingly, a pipeline decoding and execution process occurs.

As seen in the more detailed FIG. **102**, the ICACHE logic **120** includes an ICACHE data store unit **130**, a tag store unit **131** and a validity store unit **132**. As discussed with reference to the system cache **17** of the memory

system, the operation of the ICACHE is substantially similar in that the tag portion (PHY ICP 8-21) of the address of each desired word of the macro-instruction is compared at comparator **131** with the tag portions of the addresses stored in the TAG store **131** of those words which are stored in the ICACHE data store **130**. In addition, the validity store unit demonstrates whether the desired address is a valid one. If the address is valid and if a tag "match" occurs, the 32-bit double word at such address is then supplied from the ICACHE data store **130** to the decode logic **121**.

If the required macro-instructions in the appropriate ICACHE block are not present on the current physical page (i.e., the physical page corresponding to the logical page value of the current value of the program counter) which is stored in the ICACHE data store **130** (i.e., a Tag match does not occur) or if the validity bit is not set, an ICACHE "miss" occurs and the cache block containing the macro-instructions must be referenced from memory. Such ICACHE block memory reference may be to the system cache (SYS CACHE) or to the main memory, if the system cache access also misses. When the accessed ICACHE block is fetched, the desired macro-instructions thereof are written into the ICACHE data store **130** from CPM register **134** and the block is simultaneously routed directly into the decoding logic through bypass path **135**. The ICACHE logic can then continue to prefetch the rest of the macro-instructions from the fetched page as an instruction block thereof, placing each one into the ICACHE data store **130** as they are accessed. The control logic for the ICACHE logic **120** is ICACHE/ICP control logic unit **136**.

The decode logic, shown in more detail in FIG. **103**, includes instruction decode units **140** and **141** for decoding the opcode portion of the macro-instructions. Decode unit **140** is used for decoding the opcodes of the original basic instructions for the system of which the present system is an extension. Thus, in a specific embodiment as discussed above, such basic instructions may be the NOVA and ECLIPSE instructions for Data General Corporation's previous NOVA and ECLIPSE system. Decode unit **141** is used for decoding the opcodes of the extended instruction set, e.g. the "Eagle" macro-instructions mentioned above.

The opcodes are supplied from an instruction decode register (IDR) **142** having three storage register sections, each capable of storing a word and identified as IDR A, IDR B and IDR C. The opcode of each macro-instruction is stored in the IDR A section while displacements are stored in the IDR B and C sections. An IDR shifter unit **143** is used to shift the desired opcode portion of the instruction accessed from the ICACHE data store **130** into the IDR A section of IDR **142** and to shift the appropriate displacement words of the instruction, if any, to the IDR B and IDR C sections thereof. The control logic for the IDR and the IDR shifter units is IDR/shifter control unit **137**, shown in FIG. **102**.

When the macro-instruction has been routed to the decode logic, the decode units **140** or **141**, as required, decode the opcode portion thereof to provide opcode description (OPCD DSCR) information, including the length of the instruction (i.e., whether the instruction comprises a single, or double or triple word). When the entire instruction has been supplied to the decode logic (from ICACHE data store **130**) a SET IDR VLD signal is generated to produce an IDR VLD signal at IDR/shifter control **137** (FIG. **102**). Following the decoding process, the starting micro-address is loaded into the STμAD register **144** from either decode PROM **140** or **141** depending on whether the macro-instruction is a basic or an extended instruction. Control of the loading of STμAD register **64** resides in STμAD load control unit **145**.

The displacement word or words, if any, are generally present in IDR B or C (for certain NOVA instructions a byte displacement may be extracted from IDRA, although generally for almost all other instructions displacements are extracted from IDRB and IDR), being extracted from the displacement logic **146**, as shown in FIG. **104**. The displacements are sign or zero extended, as necessary, and are clocked into a displacement register thereof so as to be made available either directly to the logical address (LA) bus or to the CPD bus for use at the ALU unit, as discussed below.

When the starting micro-address has been clocked into STμAD register **144**, and UPDATE signal is issued by the IP status logic unit **138** (FIG. **102**) to inform the IDR/shifter control **143** that the decoded information has been used and can be shifted out of IDR **140/141**. The decoding of subsequent macro-instructions continues until a discontinuity in the straight-line decoding operation occurs. When a jump in the straight-line operation occurs the micro-sequencer issues an IPSTRT signal to the program counter register **147** of the instruction processor (FIG. **20**) so that a new program counter address (LA 4-31) can be placed in the program counter register from the logical address bus. The starting micro-address register **144** is reset and the starting micro-address of an appropriate wait routine, for example, is placed therein until the decoding process for the instruction associated with the new program counter can begin.

In some situations the sequence of macro-instructions which are being decoded are present on more than one physical page. Under such conditions when the ICACHE control detects the end of the page which is stored in the ICACHE data store **130**, a special routine must be invoked in order to fetch the next page into the ICACHE store **130** so as to continue the prefetching operation on the new page. Thus, when the last instruction of a particular page has been decoded and the decode pipeline is effectively empty, the starting micro-address register is loaded with the starting micro-address of a suitable page control routine which accesses the required new page and permits the next page to be loaded into ICACHE store **130** via physical page register **134** so that the instruction processor can continue with the decoding of the macro-instructions thereon.

If a macro-instruction is not on the page contained in the ICACHE store **130**, the correct page must be accessed from either the system cache or main memory because of an ICACHE "miss" in the instruction processor. Access to the system cache is provided at the same system cache input port as that used by the address translation unit (ATU). In the system of the invention, however, the ICACHE is given a lower priority than the ATU so that if the ATU wishes to access the system cache the instruction processor must hold its access request until the ATU has completed its access.

The use of ICACHE logic as described herein becomes extremely advantageous in programs which utilize a short branch backwards. If a macro-instruction branch displacement is less than the number of words in the ICACHE data store there is a good chance that the

required macro-instructions will still be stored locally in the ICACHE data store and no additional system cache or main memory references are required.

In a particular embodiment, for example, the overall ICACHE logic **120** may comprise a single set, direct mapped array of **256** double words in data store **130** plus Tag and Validity bits in Tag Store **131** and Validity Store **132**. Data is entered into the data store as aligned double words and the ICACHE data store is addressed with the eight bits which include bits $\overline{\text{ICP 23-27}}$ from the instruction cache pointer (ICP) unit **150** shown in FIG. **105** and bits ADR **28,29,30** from unit **139**.

A copy of the Tag store **131** of the instructor processor's ICACHE unit is also kept in the system cache, the latter cache needing such information so that it can inform the instruction processor when data has been written into the ICACHE.

The validity store **132** is arranged, for example, in a particular embodiment, as 64 double words by four validity bits in order to indicate the validity of each double word in the ICACHE data store. Each initial fetch into a new block of instruction words will set the corresponding validity bit for the double words and reset the remaining three validity bits. During a prefetch operation into the same block, the corresponding validity bit for the prefetch double word is set while the remaining three validity bits remain the same. The prefetching operation stops when the last double word in the block has been prefetched in order to avoid unnecessary system cache faults.

If the ICACHE operation is such that the end of a physical page is reached and it is necessary to obtain the next physical page address for the next logical page of the program counter (PC bits **4–21**), the ICACHE control logic unit **136** (FIG. **102**) asserts a signal (identified as the $\overline{\text{ICAT}}$ signal) which is supplied to the STμAD load control logic **145** (FIG. **103**). When the last macro-instruction at the end of the current page has been decoded, the STμAD control logic **145** supplies the starting micro-address for the $\overline{\text{ICAT}}$ micro-code routine which thereupon performs the necessary address translation operation for a transfer of the next physical page address for the ICACHE data store **130**.

The instruction processor utilizes two pointers to the instruction stream. The first pointer is the program counter register **147** (FIG. **104**) which holds the logical address of the instruction which is being executed, and the second pointer is the instruction cache pointer (ICP) **150** (FIG. **106**) which holds the logical address of the next macro-instruction which is needed for the decode logic. A separate register PICP **152** (physical instruction cache pointer) holds the physical page address of the logical page referred to by bits **4–21** of the instruction cache pointer (ICP). Thus the ICP **150** functions as the prefetch logical address pointer and the PICP functions as the prefetch physical address pointer. The program counter **147** and the ICP **150** are loaded from the logical address bus at the start of an instruction processor operation. The ICP is incremented ahead of the program counter as the decoding pipeline operation is filled. On an ICACHE fault, or miss, the PICP physical address is used to reference the memory and the ICP address is used as a pointer to the next logical page address for address translations when the end of the correct page has been reached.

In accordance with the instruction processor operation the optimum performance is achieved when the instructions are locally available in the ICACHE, such instructions thereby becoming substantially immediately available when the micro-sequencer requests them. Instructions which are not locally available in the ICACHE take an amount of time which is dependent on system cache access operation and page fault routine operations.

The macro-instruction decoding logic utilizes three 16-bit fields indentified as the IDR A, IDR B, and IDR C fields, as mentioned above. The "A" field contains the opcode while the "B" and "C" contain either the displacement(s) for the instruction in the "A" field or one or more fields of the macro-instruction which follows in the instruction stream. The instruction decode register, IDR **142**, is arranged to keep all three fields full, if possible, by sending word requests to the ICACHE (ICP control unit **136**) when any of the three IDR fields is empty. As mentioned above, if the ICACHE word request results in an ICACHE "miss" a system cache fetch is initiated.

The "A" field of the instruction decode register **142** is used by the decode logic PROMs **140** or **141** to decode the opcode of the macro-instruction and, also to provide the starting address of the macro-instruction which is required. The "B" and "C" fields determine the displacements, if any, that are required. Each field is one word in length and therefore the longest instruction that the instruction processor can decode and canonicalize the displacement for has a maximum length of three words.

When the A field of the instruction decode register is full, the decode PROMs **140** or **141** decode the opcode of the instruction. If the entire instruction, including opcode plus displacement, is in the instruction decode register, a signal IDR VLD is asserted by the IDR shifter control logic **137** to inform the IP status logic **138** that an entire instruction is ready to be decoded so as to provide a starting micro-address for STμAD register **144**. The displacement logic **146** which extracts the displacement, either sign or zero extends it, as necessary, and then loads it into a displacement register. If the displacement index is on the ALU board the displacement is latched onto the CPD bus via latch unit **153** for supply thereto. If the displacement index is the PC register **147**, the displacement is added to the PC bits at adder **148** and supplied to the logical address bus via latches **149**, as shown in FIG. **104**.

During the above loading processes the instruction decode register **142** is shifted by the length of the instruction that has been decoded so as to be ready to receive the next instruction, i.e., a shift of one, two or three words. The IDR shifter unit **143** serves to provide such shift of the contents of the instruction decode register **142**. A shift of three words, for example, completely empties the instruction decode register which is then ready to receive the next instruction from the ICACHE (or directly from memory on an ICACHE "miss"). The shifter, for example, allows either word in a double-word instruction which has been accessed from the ICACHE to be directly loaded anywhere into the instruction decode register. The placement in IDR **142** is detemined by examination of the validity bits in the IDR. Thus if the "A" field is invalid, the incoming instruction data would be loaded into the "A" field. Whenever any of the three fields in the instruction decode register **142** are empty, a word request is made of the ICACHE via ICACHE control logic **136** for accessing the next instruction as determined by the ICACHE pointer (ICP) **150**, bits **23–27** of which uniquely

determine which double-word in the ICACHE is to be accessed. If the instruction is a single word instruction, the ICP bits **28–30** and the ICPX bits **28–30** obtained from the fetch request control logic **151** (FIG. **105**) uniquely determine which word of the double word is to be used as the instruction as shown at word pointer logic **139** (FIG. **102**).

If the instruction decode register **142** has at least two fields empty and a word pointer points to an even double word, then the double word would be loaded into two empty fields of the IDR. After loading, the ICACHE pointer **150** would be incremented so that it points to the next double word. If the IDR has only one empty field and a word pointer points to an even double word, then the first word would be loaded into the IDR and the word pointer would be sent to point to the second word of the double word and the ICACHE pointer remains the same. When the word pointer points to the second word, only one word can be accessed from the ICACHE and loaded into the instruction decode register.

The decode logic utilizes predecode logic **154** (FIG. **103**) which is used to select the location in one of the two sets of decode PROMs **140** and **141**. As mentioned above, one set of PROMs **140** holds a basic set of instructions (e.g., NOVA/ECLIPSE instructions) while the second set of PROMs **141** holds the extended instructions (e.g., EAGLE instructions). The decoding process for the basic set of decode PROMs **140** is performed in two stages, the first level being performed in the predecode logic **154** at the output of the shifter which is used to place the basic macro-instructions into the correct form so that the decode logic **140** can decode the opcode and be ready with the displacement information in the correct form and sequence. Such logic is shown in more detail in FIG. **122**. The instructions for the extended set are already in the desired form and need not be predecoded before being supplied to the decode PROMs **141**. In either case each incoming macro-instruction maps into at least one location of a selected one of the decode PROMs **140** or **141** to produce the required opcode descriptors and the required starting micro-address for supply to the micro-sequencer.

The decision to select the output of decode PROM **140** (e.g., NOVA/ECLIPSE) or decode PROM **141** (e.g., EAGLE) is determined by examining selected bits (e.g., bits **Ø**, **12–15** as discussed above) of IDR A. As described above, the selection of the decode PROM is not determined by a separately designated "mode" bit as in previous systems, which prior process causes the decode operation to be mutually exclusive. In contrast, the present system in selecting the appropriate decode operation performs such operation on an instruction by instruction basis since each instruction inherently carries with it the information required to determine such decode selection.

Specific logic circuitry for implementing the block diagram of the instruction processor to provide the operation discussed above with reference to FIGS. **101–106** is shown in FIGS. **107–136**. ICACHE data store **130** and the ICACHE data store address input logic are shown in FIGS. **107** and **108**, respectively, while CPM register **134** supplying cache block words from memory being shown in FIG. **109** and **109**A. ICACHE tag store **131** is also depicted in FIG. **109**B and **109**C and ICACHE validity store **132**, together with the validity store address input is shown in FIGS. **110** and **111**, respectively. Comparator **133** and logic for providing the SET IDR VLD signal are shown in FIG. **112**.

FIG. **113** shows IDR shifter **143**, the IDR shifter control logic **137** being shown in FIG. **114**. The instruction decode register (IDR) unit **142** is depicted in FIG. **115** and include IDR sections A, B and C as shown.

With reference to the ICACHE logic circuitry the ICACHE pointer (ICP) logic **150** and the ICP logical address driver logic of FIG. **106** is shown in more detail in FIGS. **116** and **117**, respectively. The ICACHE pointer pre-fetch request control logic **151** and the physical ICP translation register **152** of FIG. **105** is depicted in more detail in FIGS. **118** and **119**, respectively. Other general ICACHE control logic is further depicted in FIG. **120**.

The driver logic which provides inputs FASA**Ø**-**15** from the CPD bus to IDR A as shown in FIG. **103** is depicted in FIG. **121**, while the instruction pre-decode logic and control therefor is shown in FIG. **122**. Decode PROMS **140** and **141** which effectively include the STμAD register **144**, together with the IP status logic **138** are shown in FIG. **123**. The starting microaddress control logic **145** is depicted in detail in FIG. **124**.

With reference to the displacement and program counter portion of the instruction processor, the displacement logic **146** is shown in FIG. **125**, the displacement multiplexer associated therewith being depicted in FIG. **126**. The sign extend (SEX) logic is shown in FIG. **127**, while the zero/ones extend logic is shown in FIG. **128**. FIG. **129** shows the displacement increment buffer of FIG. **104** while the displacement latch and drivers **153** are depicted in FIG. **130**. FIG. **131** shows program counter register **147** and the CPD bus driver of FIG. **104**, while adder **148** and the PC+DISP latch and driver units **149** are shown in FIGS. **132** and **133**, respectively. Program counter clock logic is depicted in FIG. **134**.

General instruction processor timing and control logic circuitry is shown in FIG. **135**, while the system cache interface logic required for the instruction processor **12** to interface the system cache **17** is shown in FIG. **136**.

MICRO-SEQUENCER

The primary function of the micro-sequencer unit is to generate micro-instructions from the starting micro-address which is supplied to a random-access-memory (RAM) storage unit on the micro-sequencer board. An overall block diagram of the micro-sequencer board for the particular embodiment of the system of the invention described herein is shown in FIGS. **137–138**. As can be seen, the RAM storage unit is identified as the micro-control store unit **170** and is capable of storing up to 4-K 80 bit (79 bits plus 1 parity bit) micro instructions and is sufficient to store all of the micro-instructions required for the system being described. The micro-instructions can be appropriately loaded into store unit **170** initially (i.e., prior to the use of the system) through a suitable console via appropriate console interface logic unit **171**. Once the entire micro-instruction set has been loaded into the micro-control store unit **170**, the console interface logic need no longer be used, unless a micro-instruction is changed or additional micro-instructions are to be stored. Addresses for the micro-instructions are supplied at the RA input to the micro-sequencer board.

Once the entire micro-instruction set has been loaded into the micro-control store **170**, the system is ready for performing the micro-instructions, as determined by the instruction processor unit **12** which, as discussed above, supplies the starting micro-address (STμAD) for a micro-instruction routine. As can be seen in FIG. **137**, the starting micro-address (STμAD) is supplied via buffer **172** and AND circuitry **173** to the address input of the micro-control store **170**. The starting micro-address selects the starting micro-instruction at the appropriate location in the micro-control store and supplies the control signals associated with said instruction via buffer **174** to the appropriate locations within the overall data processing system which are involved in the operations required for such instruction in a manner similar to that which would occur in supplying instructions to any data processing system.

The micro-sequencer must then determine the next address required for the next sequential micro-instruction (if any) via appropriate decoding of the "next address control" field (NACØ-**19**) of the current micro-instruction. This field in the particular embodiment described is a 20-bit field of the 80-bit micro-instruction obtained from the micro-control store. The NAC field is suitably decoded by the NAC decode logic **175** to provide the necessary control signals (some of which are identified) required to obtain the next micro-address. The decoding process can in one mode be a conditional one, i.e., wherein the NAC field decoding is conditioned upon one of a plurality of possible conditions which must be appropriately tested to determine which, if any, condition is TRUE. In the particular embodiment described, for example, there are eight test signals (TEST Ø-7) each test representing 8 conditions, for a total of **64** conditions which can be tested. Alternatively, in another mode the selection of the next micro-address may not be conditioned on any of the 64 conditions involved. After appropriate testing the address is selected from one of four sources, as determined by the decoding and condition test logic **182**, for supply to the micro-control store **170** via ADDR multiplexer unit **176**. Decoding and condition test logic **182** is shown in further detail in FIG. **138**.

Thus, the address multiplexer output can be selected from the next sequential program counter address (μPC **4-15**) which represents the previous micro-address incremented by one as obtained from the (μPC+1) unit **177** and increment logic **178** which accepts the previous micro-instruction (RA **4-15**), increments it by one and supplies it to an input of the address multiplexer unit **176**.

Alternatively, the next micro-address may be obtained from a temporary storage of a plurality of micro-addresses for a particular micro-code routine which addresses have been stored in a stack RAM storage unit **179**, the next address being supplied directly as the address at the top of the stack (TOS **4-15**) via a top of the stack (TOS) register **180**. Alternatively, the address at the top of the stack may already have been accessed (popped) from the stack and saved in a previous operation in the Save TOS register **181** (particularly used in restoring the overall context after in interrupt process) so that the next micro-instruction address may alternatively be obtained from the top of the stack data (STOS **4-15**) which has previously been saved in the STOS register.

A further source of the next micro-address for the address multiplexer may be an absolute address from decode and condition test logic **182**, shown more specifically in FIG. **138**, which address is specified by the micro-instruction word itself or an absolute address which may be identified by bits from another source external to the micro-sequencer board which other sources dispatch such address to the micro-sequencer, i.e., from the address translation unit (ATU) or from the arithmetic logic unit (ALU) selected bits of which can be suitably concatenated with absolute address bits from the current micro-instruction to form the next micro-address. As see in FIG. **138**, the latter bits may be received via suitable registers **183** and **184** (see FIG. **138**) from the ATU at the ATU dispatch (ATUD) register **183** or from the ALU on the CPD bus at the CPD register **184**. Thus, as seen best in FIG. **138**, such bits (ATUD **13-14** and CPD **20-31**) can be concatenated with bits from the micro-instruction itself, identified by NAC bits Ø-2, **8-19**, to form five possible micro-addresses by concatenation logic unit **185**. One of five concatenated addresses is capable of being selected at Dispatch Multiplexer unit **186** and thereupon supplied to Address Multiplexer **176**.

In order to obtain the desired stack data for the next possible micro-address (TOS **4-15** or STOS **4-15**) suitable stack pointer logic **187** and stack control logic **188** are used with the stack RAM unit **179**. The stack addresses which are supplied via stack pointer logic **187** determine the locations of the sequence of micro-instruction addresses which are required for micro-code routines, which sequence has been previously supplied to the stack via stack multiplexer unit **189**, the inputs of which are obtained either as absolute addresses (AA **4-15**) from the micro-instruction which is currently being processed or as addresses obtained from the micro-program counter **177** (μPC+1), from a dispatched ALU source (CPD **20-31**) via the CPD bus, or from an address which has been previously saved (AD **4-15**) in save register **190**.

When a micro-code routine which has been stored in the stack RAM is completed, the stack is then empty and a STKMT signal from the stack pointer logic **187** produces an appropriate IPOP OUT signal at the output of IPOP detection and latch logic **191** for supply to the instruction processor to indicate that a new starting micro-address (STμAD) is required to provide the next micro-instruction or sequence thereof.

As a simple example of the operation of the micro-sequencer to illustrate the same, in a conditional jump instruction (CJMP), let it be assumed that the address of the next micro-instruction is to be supplied either as an absolute address from the dispatch multiplexer to which the micro-program must jump if the condition is TRUE or as the next sequential program address from the micro-program counter (PC+1) if the condition is not TRUE. For example, if the present micro-address is at a selected location of the μ-control store **170** (e.g. location "100") the next micro-address is to be either the location signified by the next sequential program counter address (e.g., location "101") if the condition is not TRUE, or a jump to specified absolute address (e.g., at location "500") if the condition is TRUE. In order for the micro-sequencer to determine which of the two locations is be be selected, i.e., the absolute address (AAD **4-15**) or the micro-program counter address (μPC **4-15**), the condition must be tested to determine if it is "TRUE".

If testing of the condition provides a TRUE at the condition out logic **192**, the absolute address (AAD

4-15) will be selected as the correct address from address multiplexer **176**, while if the condition is not TRUE, the next micro-program counter address (μPC 4—15) will be selected. The testing logic **198** is shown in FIG. **138**.

Specific logic circuitry for implementing the micro-sequencer unit **13** as discussed above and shown in the block diagrams of FIGS. **137** and **138** are shown in FIGS. **139-153**. Stack logic circuits, including the stack ram **179**, the stack multiplexer **189**, the stack pointer unit **187** and the top-of-stack unit **180**, are specifically shown in FIG. **139**. The save-top-of-stack unit **181** is shown in FIG. **140**. Address multiplexer **176** is depicted in FIG. **141**, while the address save register is shown in FIG. **142** and the address logic **173** for supplying addresses to the micro-control store **170** is shown in FIG. **143**. FIG. **144** depicts the starting micro-address(ST-μAD) driver unit **172**. The imcremented micro-program counter (μPC+1) unit **177** and increment unit **178** are shown in FIG. **145**.

Micro-control store **170** is specifically depicted in FIG. **146*** and the next address control (NAC) decode logic circuitry **175** is specifically shown in FIG. **147**. Parity logic is shown in FIG. **148**.

*FIG. 146A—F through FIG. 146.7A—F

With reference to the decoding and condition test logic circuitry **182**, shown particularly in FIG. **138**, specific logic circuitry for implementing such circuitry is shown in FIGS. **149-153**. Thus, concatenation logic **185** and dispatch multiplexer **186** are depicted in FIG. **149**, CPD multiplexer **197** is shown in FIG. **150**, 6-bit counter **196** is shown in FIG. **151**, 8 flags unit **193** is shown in FIG. **152**, and test Ø and test 1 multiplexers **194** together with condition multiplexer **195** and the condition output unit **192** are all shown in FIG. **153**.

ARITHMETIC LOGIC UNIT

Before discussing in more detail the format of the micro-instruction word, it is helpful to discuss FIG. **153** which shows a block diagram of a typical arithmetic logic unit generally having a configuration known to those in the art. As can be seen therein, the ALU unit **200**, which performs the arithmetic and logical operations, has two inputs, identified as inputs R and S, which are supplied from a pair of multiplexers **201** and **202**, respectively. The inputs to multiplexer **202** are obtained from the A and B outputs of a register file **203**. A third input may be obtained from a source which supplies zeros to the multiplexer at all 31 bit positions (identified as the "Ø" input) and a fourth input may be obtained from Q register **204**.

Register file **203** contains **16** and **32** bit registers and includes four fixed point registers (ACCØ-3), four floating point registers (FPACØ-3), and eight general registers (GRØ-7). The selection of the appropriate registers for supplying the A and B inputs to ALU **200** is determined by the AREGØ-3 and BREGØ-3 bits of the micro-instruction field, as discussed in more detail below. The inputs to multiplexer **201** are obtained from the A output of the register file, from the D-bus **205** or from an all zeros input, as discussed with reference to multiplexer **202**. The output of ALU **200** is supplied to a multiplexer **206** which selects either the output from ALU **200** or an output directly supplied from the A terminal of register file **203**. The output of multiplexer **206** can be supplied to the logical address bus if the calculation is an address calculation, to the register file **203** for writing back into a selected register therein, to Q register **204** or to a plurality of other units on the arithmetic logic board, significant exemplary ones of which are identified as shifter units **207**, a data store register **208** or directly to the D-bus **205** or to the memory data bus. The shifter outputs are supplied to the D-bus, while the data store register **208** supplies data to the CPD bus or to the D-bus via CPD register **209**. Data supplied to the D-bus can then be used in subsequent arithmetic or logic operations via multiplexer **201**. Other sources of the system may also supply data to D-bus **205**, if desired. The general configuration of the arithmetic logic unit board **11**, as shown in FIG. **154**, is helpful in understanding the micro-instructions which are discussed below.

MICRO-INSTRUCTION FORMAT

As discussed above with reference to the micro-sequencer unit **13**, the micro-control store **170** thereof supplies a micro-instruction of 80 bits, the format thereof being depicted below.

NAC	AREGS	BREGS	CSM	DIST	D2ND	SHFT	ALUS	ALUOP
20	4	4	4	2	4	4	3	3

ALUD	CRYINS	RAND	LAC	CPDS	MEMS	MEMC	FREE	UPAR
3	1	10	2	5	3	2	5	1

The overall format comprises eighteen fields, one field of which has five bits available as reserve bits for future use. The seventeen fields which are utilized are described below.

The Next Address Control Field (NACØ-19)

As discussed above with reference to the micro-sequencer structure and operation, the first 20 bits of the micro-instruction format comprise the field for controlling the selection of the address for the next micro-instruction which address is either a "conditional" address, i.e. an address the selection of which is dependent on whether a specified condition which is tested is either true or false, or an "unconditional" address, i.e., an address which is selected independently of any conditions.

The NAC field of the micro-instruction for selecting a conditional address carries with it a 6 bit test field which identifies which of up to 64 conditions must be tested to determine whether a specified condition is true or false. The basic format of the NAC field for selecting a conditional address is shown below:

UP	POL	TEST	PA
3	1	6	10

The conditions which can be tested may relate to conditions with respect to operations of the arithmetic logic unit, the address translation unit, the instruction processor, the micro-sequencer unit itself or input/output (I/O) conditions. As an example of typical condi-

tions, Appendix C lists **53** conditions which can be tested in the particular system design described herein involving tests relevent to the ALU, ATU, IP and micro-sequencer units, as well as certain I/O tests.

Various types of conditional addresses may be selected as discussed below, it being helpful to consider the following discussion in conjunction with FIGS. **33** and **34** showing broad block designs of the micro-sequencer logic.

A first conditional address may be a conditional absolute address, i.e. an address which uses absolute address bits AA **4-15** appropriately selected and supplied by dispatch multiplexer **186** to the address multiplexer **176**, as seen in FIG. **4**.

The format for such conditional absolute address utilizes the same format shown above for the mode bits, polarity bit and test bits, with the 10 absolute address bits being extended to a full 12 bits by concatenating the most significant bits of the current micro-program counter as the first two bits thereof (sometimes termed the "page bits"). The conditional absolute address may be utilized in 5 different modes as set forth in Appendix D (see "Absolute Address Conditional" therein). An example of one mode such as a "Conditional Jump Code" (CJMP) can be illustratively summarized below.

Mode	MneM.	Explanation	True Action	False Action
000	CJMP	Conditional Jump	PC ← AA(10)	PC ← PC + 1

For such conditional jump mode, if the specified test condition is true the 10 absolute address bits concatenated with the 2 page bits forms the absolute address bits AA **4-15**, which address is then selected at the address multiplexer **176** (FIGS. **33** and **34**). If such specified condition is false, the address which is selected is the current program counter address incremented by **1** (i.e. $\mu PC+1$). Other modes for an "absolute address conditional" format are shown in Appendix D.

Another conditional address is a conditional dispatch address, wherein a portion of the address bits are obtained (or dispatched) from sources external to the micro-sequencer unit (such as the arithmetic logic unit or the address translation unit, for example) which dispatch bits can be concatenated with some or all first eight absolute address bits (AAØ-7) as shown in FIG. **34**. For such conditional dispatched addresses the following format is used:

0 2	3	4 9	10 16	17	18 19
101	POLARITY(1)	TEST(6)	AA(7)	X	DSRC(2)

The source from which the dispatch bits are obtained are identified by the two DSRC bits for **4** different source identifications.

Thus, the address may be formed by direct replacement of the lower 8 bits of the formed absolute address with the lower 8 bits of the CPD bus as shown below.

0 3	4 11
Formed AA bits 0-3	CPD24-CPD31

Alternatively, the address may be formed by direct replacement of the lower 4 bits of the formed absolute address with the lower 4 bits of the CPD bus, as shown below:

0 7	8 11
Formed AA bits 0-7	CPD28-CPD31

As further alternative, the address may be formed by direct replacement of the lower 4 bits of the formed absolute address with a different 4 bits of the CPD bus as shown below:

0 7	8 11
Formed AA bits 0-7	CPD20-CPD23

And as a final alternative, the address can be formed by direct replacement of the lower 3 bits of the formed absolute address with 2 bits from the address translation unit validity dispatch, with a zero in the least significant bit position, as shown below:

0 8	9 10	11
Formed AA bits 0-8	ATU VAL(2)	0

Certain addresses may require the use either of the incremented program counter address or the top of the stack address (with the top of the stack being appropriately popped, or removed, when the address is used) and for such purposes the lower 12 bits (NAC-**19**) need not be involved in the address generation process. Accordingly, such 8 bits are available for other purposes as desired. The format therefor is shown below:

0 2	3	4 9	10 11	12 19
110	POLARITY(1)	TEST(6)	Mod2(2)	OTHER(8)

An explanation of such three special condition address selections are shown in more detail in Appendix D, identified as LCNT, CPOP and LOOP.

Certain addresses may be selected in conjunction with the setting of the 8 flags that are involved and such flag control commands can be identified by the NAC field in accordance with the following format:

0 2	3	4 9	10 12	13	14 15	16 17	18 19
110	POLARITY(1)	TEST(6)	110	POP(1)	SET(2)	CNTL1 (2)	CNTL2 (2)

As seen in Appendix D (see Flag Controls set forth therein) such instructions can be divided into two sets each set being identified by the POP bit and each set having four different instructions identified by the two SET bits. Each instruction involves the setting of two flags, each flag being set in accordance with the CNTL1 or CNTL2 fields as follows:

CNTL1 or CNTL2	Action
00	no change
01	set it FALSE
10	set it TRUE
11	Toggle it

In each of the above flag control cases if the test condition which is specified is determined to be "True" the incremented micro-program counter address is used ($\mu PC+1$) while if the condition is "false" the top of the stack address is utilized and the stack is appropriately popped. As mentioned above, a summary of the flag controls is set forth in Appendix D.

Two of the instructions of the NAC field allow the conditional use of the stack without popping it (as opposed to the use and popping thereof discussed above) in accordance with the following format:

0 2	3 4	9 10 12	13 14	15 16 17	18 19
110	POLARITY(1)	TEST(6)	111	Mod3 (2)	FS(1)	CNTL1 (2)	CNTL2 (2)

Two instructions are involved, flag control being provided for either the set of flags **Ø** and **1** or the set of flags **2** and **3**. A summary of such instructions, identified as the SPLIT instructions is shown in Appendix D. As can be seen therein, if the condition is "false" the top of the stack address is utilized but the address remains at the top of the stack (i.e. the top of the stack is not popped). The final conditional instruction is a context restore instruction. Such instruction may be used, for example, after a fault routine has been implemented and it is desired to restore the machine to its previous state. In accordance therewith, not only is the machine state restored but a decision is made as to the next micro-address which should be utilized, depending on whether the condition which is tested is true or false. The context restore instruction format is shown below:

0 2	3 4	9 10 12	13 14	15 19
110	POLARITY(1)	TEST(6)	111	Mod3(2)	XXXXX

A summary of the two instructions involved is shown in Appendix D identified as Context Restore Instruction.

In addition to the conditional address instructions discussed above, in a particular embodiment of the system discussed, there are also unconditional address instructions (one particular embodiment utilizing eight unconditional instructions are set forth in Appendix D identified as Unconditional Instructions). In accordance with the format thereof there are no conditions to be tested so that for each mode of operation only a single action is specified and no selected choice need be made.

A summary of the unconditional address instructions, which can be divided into unconditional instructions utilizing the 12-bit absolute address or unconditional instructions utilizing the combinations of certain absolute address bits and dispatch source bits (Unconditional Dispatches) is shown in Appendix D.

AREG, BREG Fields

The 8 bits in these two fields identify which register of the register file in the arithmetic logic unit is to be used to provide the A and B inputs of the arithmetic logic unit **200**. Thus the register file is capable of selecting one of sixteen registers, namely, the accumulators AC **Ø-3**, the floating point registers FPAC **Ø-3** or other general registers GR **Ø-7** in accordance with the following select codes.

Mnem	Value
AC0	0
AC1	1
AC2	2
AC3	3
FPAC0	4
FPAC1	5
FPAC2	6
FPAC3	7
GR0	8
GR1	9
GR2	A
GR3	B
GR4	C
GR5	D
@ACSR	E
@ACDR	F

In the above table the coded value is in hexadecimal notation and in the specific case of coding ACSR or ACDR, the register file control comes from a register that specifies a source accumulator or from a register that specifies a destination accumulator. When the source accumulator ACSR **Ø-3** or the destination accumulator ACDR **Ø-3** equals hex E the general register GR6 will be selected. When ACSR **Ø-3** or ACDR **Ø-3** equal hex F then the general register GR7 will be selected.

The Control Store Mode

The control store mode 4-bit field defines the functionality of six of the other micro-instruction fields, namely, the ALUS, ALUOP, ALUD, DIST. CRYIN, and RAND fields. The following table summarizes the 16 control modes for the control store mode field.

		Half-cycle 1			Half-cycle 2			DIST		RAND
Mnem	Value	ALUS	ALUOP	ALUD	ALUS	ALUOP	ALUD	Type	CRYIN	Type
SMATH	0	uI	uI	#	DZ	OR	uI	Math	Type0	Math
SFIXP	1	uI	uI	#	DZ	OR	uI	Gen	Type1	Fixp
SGEN	2	uI	uI	#	DZ	OR	uI	Gen	Type0	Gen
SATU	3	uI	uI	#	DZ	OR	uI	Gen	Type0	Atu
FMATH	4	uI	uI	#	uI	uI	uI	Math	Type0	Math
FFIXP	5	uI	uI	#	uI	uI	uI	Gen	Type1	Fixp
FGEN	6	uI	uI	#	uI	uI	uI	Gen	Type0	Gen
FATU	7	uI	uI	#	uI	uI	uI	Gen	Type0	Atu
MPY	8	@	@	#	@	@	#	Math	Type2	Math
DIV	9	uI	@	uI	uI	@	uI	Math	Type3	Math
BOUT	A	uI	uI	#	ZB	OR	uI	Gen	Type0	Gen
NORM	B	uI	uI	#	DZ	OR	uI	Math	Type0	Math

-continued

Mnem	Value	Half-cycle 1 ALUS	ALUOP	ALUD	Half-cycle 2 ALUS	ALUOP	ALUD	DIST Type	CRYIN	RAND Type
QDEC	C	ZQ	SUB	GREG	uI	uI	uI	Gen	*Type0	Gen
QINC	D	ZQ	ADD	GREG	uI	uI	uI	Gen	*Type0	Gen
QADD	E	DQ	ADD	GREG	uI	uI	uI	Gen	*Type0	Gen
PRESC	F	@	uI	#	DZ	OR	uI	Math	Type0	Math

In the above table the following abbreviations are used:
uI—Represents the u-order from the appropriate field of the specified u-instruction.
#—No clock takes place.
@—The u-order will deter to a predecoded or "Forced" value. See notes below for further information.
*The CRYIN is forced to a zero the first half cycle in modes QDEC and QADD, and to a one during the first half of mode QINC.

As can be seen, operations can occur in either half of the operating time cycle of the system, for example, operations with respect to the CPU occurring in one-half of the cycle and operations with respect to I/O devices occurring in the other half of the cycle. The above table shows that the control modes for the control store mode field must be defined in accordance with the half-cycle which is occurring. Thus certain fields in the over-all micro-instruction format will change depending on which half of the cycle is occurring and the CSM field defines how each of such fields is affected during each of the half-cycles involved.

The ALU source inputs (R and S), the ALU operation and the ALU destination as determined by their respective fields are discussed below, the above table providing a definition for the functionality thereof as explained by the above noted abbreviations. The source for the D-bus (see ALU in FIG. **53**) for the first half cycle is discussed below under the DIST field. The CRYIN definition determines the type of usage for the carry input select field as discussed below and the random field (RAND) type is also defined as discussed below with respect to such field. A more detailed description of the multiply (MPY), divide (DIV), prescaled mantissa (PRESC) and NORM modes is shown in Appendix E.

The DIST Field

This 2-bit field defines the source for the 31 bits which are placed on the D-bus **205** of the arithmetic logic unit (see FIG. **53**) during the first half cycle. The functionality of this field is dependent on what is coded in the CSM field as discussed above. For the two types (identified as MATH or GEN) the following sources are defined depending on the value of the DIST field.

Mnem	Value	Description
Type Math		
MREG	0	D<0-31> = MREG<0-31>
MACC	1	D<0-31> = MACC<0-31>
CPDR	2	D<0-31> = CPDR<0-31>
AAR	3	D<0-23> = zero D<24-31> = AAR<24-31>
Type Gen		
MREG	0	D<0-31> = MREG<0-31>
CPDB	1	D<0-31> = CPD<0-31>
CPDR	2	D<0-31> = CPDR<0-31>
AAR	3	D<0-23> = zero D<24-31> = AAR<24-31>

D2ND Field

The four bits for this field define the source of the 31 bits to be placed on the D-bus during the second half cycle in accordance with the following definitions.

D<**0-31**> source during second half cycle.

Mnem	Value	Description
	0	Unassigned
CPDB	1	D<0-31> = CPDB<0-31>
CPDR	2	D<0-31> = CPDR<0-31>
AAR	3	D<0-23> = zero D<24-31> = AAR<24-31>
CREG	4	D<0-31> = MREG<0-31>
MACC	5	D<0-31> = MACC<0-31>
	6	Unassigned
	7	Unassigned
NSHR	8	Right Nipple shifts. See SHFT field
NSHL	9	Left Nipple shifts. See SHFT field
PASS	A	D<0-31> = TLCH<0-31>
	B	Unassigned
PMD	C	Processor memory data. See note below.
	D	Unassigned
ASR	E	D<0-15> = ASR<0-15>
	F	Unassigned

The SHFT Field

The four bits of the SHFT field define two basic functions, namely, a control of the inputs for bit shifts into the Q-register or the B-register of the arithmetic logic unit (FIG. **53**) and a control of a 4-bit shift (a "nibble" shift) at the Shifter **207** of the ALU. The latter shift is controlled by the D2ND field to occur only when such field is coded to produce a right nibble shift (NSHR) or a left nibble shift (NSHL) as indicated above. The bit shift occurs with respect either to the data that is present in the Q-register or to the data which is being placed into the B-register, only if the D2ND field contains something other than a NSHR or NSHL code. The charts in Appendix F explain more completely how the nibble shift and bit shift hardware are controlled by the SHIFT field.

The ALUS Field, The ALUOP Field and The ALUD Field

The 3 bits of the ALUS field determines which bus is supplied to the R and S input of the arithmetic logic circuit **200** (FIG. **53**) in accordance with the following chart.

ALUS FIELD	(R,S)
AQ	0
AB	1
ZQ	2
ZB	3
ZA	4
DA	5
DQ	6
DZ	7

In the above chart, A represents the A output of the register file, B represents the B output of the register file, Q represents the Q output from the Q register, Z is

the all zeros input and D is the D-bus in FIG. **53**. Thus, for an ALUS field of zero, for example, the R input is from the Q register, and so forth.

The three bits of the ALUOP field define the operation which is to be performed by the arithmetic logic circuit **200** in accordance with the following chart.

ALUOP FIELD		
ADD	0	(R + S)
SUB	1	(S − R)
RSB	2	(R − S)
OR	3	(R or S)
AND	4	(R * S)
ANC	5	(R' * S)
XOR	6	(R xor S)
XNR	7	(R xnr S)'

The 3 bits of the ALUD field defines the destination for the output of the arithmetic logic circuit **200** (i.e. where the result of the arithmetic or logical operation will be placed) in accordance with the following chart.

ALUD FIELD		
Mnem	Value	Description
NLD	0	No load; Y<0-31> = ALU<0-31>
GREG	1	Load GREG only; Y<0-31> = ALU<0-31>
BREG	2	Load BREG only; Y<0-31> = ALU<0-31>
AOUT	3	Load BREG only; Y<0-31> = AREG<0-31> If FLAG0 = 0, Y<0-15> = ALU<0-31>, Y<16-31> = AREG<16-31>
RSHB	4	Load BREG with ALU shifted right one bit; LINK register: = ALU31; Y<0-31> = ALU<0-31>
RSQB	5	Load BREG with ALU shifted right one bit; Shift QREG right; Y<0-31> = ALU<0-31> LINK register: = ALU31
LSHB	6	Load BREG with ALU shifted left one bit; Y<0-31> = ALU<0-31> LINK gets ALU16, ALUO for FLAG0 = 0,1 respectively.
LSQB	7	Load BREG with ALU shifted left one bit; Shift QREG left; Y<0-31> = ALU<0-31> LINK gets ALU16, ALUO for FLAG0 = 0,1 respectively.

The CRYINS Field

This field represents the arithmetic logic unit carry input select field and determines what kind of carry is used. There are 4 types of usage for this field (identified as Types **Ø-3**), the use thereof being governed by the CSM field discussed above and the RAND field discussed below. The charts in Appendix G for each type summarize the determinations to be made by the CRYINS field.

The Rand Field

The 10-bit random field is a multi-functional field and is controlled as discussed above by the CSM field. There are 4 types of usage thereof, identified as MATH, FIXP, GEN, and ATU.

The MATH type of usage has the following format:

Type Math

ROUND	FPOP	MISC
1	4	5

which includes 1 bit for controlling the rounding off of the floating point computation and the 4 FPOP bits for defining the floating point operation with regard to the exponent, multiplication and truncation utilized. The remaining 5 bits are available for other arithmetic logic unit operations, if desired. The MATH type usage for the random field is specified in the summary set forth in Appendix H.

The fixed point type usage (FIXP) has the following format:

Type Fixp

CEXT	MISC1	MISC2
1	4	5

As can be seen the first bit of the field in this type of usage combines with the CRYINS field Type **1** to form certain micro-orders as set forth below:

CRYINS Mnem	CRYINS Value	CEXT Mnem	CEST (RAND<O>) Value	Description
Z	0	N	0	CRYIN = 0
H	1	N	0	CRYIN = 0
Z,C	0	Carry	1	CRYIN = CARRY
H,B	1	Carry	1	CRYIN = CARRY

The remaining bits relate to miscellaneous operations, the first 4 miscellaneous bits (MISC **1**) relating to ALU loading control and the second 5 miscellaneous bits (MISC **2**) relating to various random operations with respect to carry, overflow and status operations, and set forth in Appendix I.

The general type of usage (GEN) utilizes the following format:

Type Gen

REGS	SPAR	SPAC
4	2	4

The first 4 bits (REGS) deal with general source and destination accumulator operations set forth in Appendix J. The 2 SPAR scratch pad bits deal with operations set forth in Appendix J. The 4 SPAD scratch pad bits deal with various scratch pad operations specified in Appendix J.

The final usage type for the random field is identified as ATU usage dealing with various address translation unit operations and has the following format.

Type ATU

ATUO	ATU1	AT02
5	2	3

The first 5 bits (ATU **0**) deal with the address translation unit operations, the next 2 ATU bits (ATU **1**) define further ATU operations, and the final 3 ATU bits (ATU **2**) define general operations, all as set forth in Appendix K.

The LAC Field

This 2 bit logical address control field controls the data that will be placed on the logical address bus, i.e. the field specifies the source for LA bits **1-31**, in accordance with the following chart:

Specifies the source of LA<1-31>.

Mnem	Value	Description
DSN	0	LA<0-31>: = WDLCH<0-31> or BYLCH<0-31>
DS	1	LA<0-31> & LAR<0-31>: = WDLCH<0-31> or BYLCH<0-31>
SP	2	LA = Scratch Pad; LAR: = Scratch Pad
IP	3	LA = PC + DISP; LAR = PC + DISP exception: when ICAT coded in ATUO, LA = ICP; LAR = ICP

The CPDS Field

This 5-bit CPD source select field determines what is placed on the CPD bus, i.e. the source for the CPD 0-31 bits. This field also controls the loading of the CPDR register on the arithmetic logic unit.

An NCPDR random field (see GEN Type random field) overrides the loading of the CPDR register and prevents such loading. The source select and other control operations for the CPDR field are specified in accordance with the chart shown in Appendix L.

The MEMS Field

This 3-bit field defines the type of operating cycle which will be started for the memory (e.g. read cycle, a write cycle, a read-modify-write cycle) in accordance with the following chart:

Mnem	Value	Description
NOP	0	
RW	1	Start a read cycle for a word.
RD	2	Start a read cycle for a double-word.
RB	3	Start a read cycle for a byte.
S@	4	Start per MEMS field of previous non LAT start. During EFA routines, the IP supplies the control.
WW	5	Start a write or rmod cycle for a word.
WD	6	Start a write or rmod cycle for a double word. See below.
WB	7	Start a write or rmod cycle for a byte.

The MEMC Field

This 2-bit field defines the completion of a memory operation in accordance with the following chart:

Mnem	Value	Description
N	0	
R	1	Read or Rmod operation.
W	2	Write operation. PMD<0-31> = DS<0-31>
A	3	Abort operation

The UPAR Field

This single bit field contains the odd parity of the micro-word. If an even parity error is detected the overall operation will stop at the current micro-location incremented by +1.

The above discussion summarizes each of the fields of the micro-instruction format in accordance with the invention. It is helpful also to describe below the usage of the 8 flags which can be defined.

Flag 0 is the width flag and defines either a narrow (16 bit) arithmetic logic unit operation or a wide (32 bit) arithmetic logic unit operation. Flag 1 is an address flag and defines whether the logical address is to be driven as a basic instruction address (e.g. for NOVA/ECLIPSE operation) in which case only bits 17-31 of the logical address are driven by the logical address latch on the arithmetic logic unit, the address translation unit or the instruction processor unit. If the flag indicates an instruction expended address than all bits 0-31 of the extended logical address are so driven.

Flags 2-7 are general purpose flags and can be used as desired by the general micro-code in sequencing. For example, flag 4 has been used as a "shift indirect" flag and, when NSH is coded in the SHFT field of the micro-instruction format (see the discussion thereof above), a shift is made either to the left or to the right depending on the setting of flag 4. Further, flag 5 has been used to define whether or not a floating point operation requires a double precision operation.

UNIQUE MACRO-INSTRUCTIONS

In accordance with the unique extended processor system of the invention, as described above, certain operations are performed by the system which operations are in themselves uniquely indigenous to the overall operating capabilities of the system. Such operations are described in more detail below and can be best understood in conjunction with the system instruction set reproduced in Appendix B.

The first operation to be considered involves an interruption of a currently executing program by a peripheral device, for example, and the need to transfer control of the system to the appropriate interrupt operating sequence. One such unique interruption operation is related to the instruction designated as "EAGLE Vector on Interrupting Device" (having the abbreviated mnemonic description XVCT) in Appendix B (the instructions in the instruction set of Appendix B are listed in alphabetical order in accordance with their abbreviated mnemonic designations). An understanding of the XVCT interrupt operation can be obtained with the help of the diagrammatic representation of the memory locations shown in FIG. 155.

Interrupt requests are examined and identified in between the decoding of macroinstructions of a currently executing program and, if an interrupt request occurs, the contents of the stack registers for the current program are first saved in selected locations provided for such purpose in the current ring of execution (e.g. selected locations in Page 0 of the current ring).

Since ring 0 is the ring reserved for special operations, e.g., interrupt operations, the systems must then cross to ring 0 (change the CRE bits 1-3 of the SBA's to identify ring 0) and load the now empty stack registers with the contents, relating to interrupt procedures, of selected locations in ring 0. Further, a selected location of ring 0, e.g., location 0, for example, is examined to determine if the interrupt is a "base level" interrupt, i.e., an interrupt condition in which no other prior interrupts are being processed, or as a "higher level" interrupt in which one or more other interrupts are already pending. If pending location 0 indicates that the interrupt is a base level interrupt (e.g, location 0 is a "zero"), as seen, for example, in FIG. 155, then the interrupt code examines a selected location (e.g., location 1) of ring 0 to determine if such location contains the XVCT code (the first 16 bits of such location 1 corresponds to the first 16 bits of the XCVT code specified in Appendix B). If the interrupt is an XVCT interrupt, the stack registers are then loaded with the XVCT information to

set up a XVCT stack, i.e., an XVCT stack "PUSH" as seen in FIG. **156**.

The displacement bits **17–31** of location **1** (corresponding to the displacement bits **17–31** of the XVCT instruction shown in Appendix B) then represent an address which points to a selected location in a preloaded XVCT table in the main memory (see FIG. **155**). The "device code" information (a 16 bit offset code unique to each I/O device from which an interrupt request can be received) is received from the particular device which has requested the interrupt and offsets to a selected address which points to a particular device control table (DCT) in main memory associated with that particular device (e.g., DCT associated with device N identified in XVCT table). The device control table contains the address which points to macroinstructions in main memory which are required in order to perform the interrupt routine requested by the interrupting device.

The DCT also contains a coded word ("MASK") which identifies which other device can be "masked out" (i.e., prevented from performing an interrupt while the interrupt is pending for the particular device in question). Certain other devices which have higher interrupt priority than the device in question will not be so masked.

The DCT further defines the state of the system by a PSR (processor status register) word which is loaded into the PSR of the system and determines whether or not a fixed point overflow condition is to be enabled.

Once the macroinstructions for the particular interrupt routine requested by the particular device in question have been performed, the previously stored contents of the system stack registers relating to the program currently being executed by the system prior to the interrupt are restored to the system stack registers and such program continues its execution. The overall operation is shown diagrammatically in FIG. **156**.

Another operation unique to the system described herein involves the loading of the segment base registers (SBR) of the system and related to the LSBRA instruction described in the instruction set of Appendix B. As explained above, the SBR's of the systems are not located in main memory but are more readily available on the ATU board of the system. The eight segment base registers of the system each contain a double word of a block of eight double words. The operation described here relates to the loading of such SBR's with an eight double-word block from memory, the starting address of which is contained in a selected accumulator of the system (e.g., ACØ). The LSBRA operation then loads such block into the SBR's in the manner shown by the table designated in connection with the LSBRA instruction in Appendix B.

In another operation indigenous to the system described here the 31-bit value contained in the program counter (PC), as discussed with reference to the instruction processor unit (FIG. **20**), is added to the value of the displacement contained in a particular instruction word and the result is placed in the program counter, as shown with reference to address **148** and PC register **147** of FIG. **20**. The displacement is contained in the instruction designated as WBR (Wide Branch) in the instruction set in Appendix B. Such operation is in effect a program counter "relative jump" and involves a 16-bit EAGLE address (PC) and an 8-bit offset, the latter contained as bits **1–8** of the WBR instruction.

In connection with EAGLE operation in the extended system of the invention, operations are performed to extend (i.e., to validate), 16-bit data to 32 bits. Such operations will involve either zero-extending (ZEX) or sign-extending (SEX) the 16-bit data, as shown in the ZEX or SEX instruction in Appendix B. Thus, for a zero extend operation the 16-bit integer which is contained in the source accumulator (ACS) identified by bits **1**, **2** of the instruction, is zero-extended to 32 bits and the result is loaded into the destination accumulator (ACD), identified by bits **3**, **4** of the instruction, with the contents of ACS remaining unchanged, unless such accumulators are the same accumulator. For a sign extend operation the 16-bit integer in the ACS is sign extended and placed in the ACD as above.

A further operation unique to the extended system of the invention involves an operation in which the signed 16-bit integer in bits **16–31** of the ACD is multiplied by the signed 16-bit integer in bits **16–31** of the ACS. Such operation is associated with the Narrow Multiply (NMUL) instruction in Appendix B. Since the system utilizes 32-bit accumulators, when multiplication of 16-bit words (i.e. "narrow" words) is required it is necessary to use only 16 bits of the 32-bit accumulator contents. An overflow occurs if the answer is larger than 16 bits, so that if the overflow bit "OVK" is in a selected state (e.g. OVK is a 1) an overflow indication occurs and the machine operation is stopped (a "trap" occurs) and an overflow handling routine must be invoked.

The above discussed unique operations of the system of the invention are all indigenous to the design and operation thereof and represent operations not required or suggested by other previously known data processing systems.

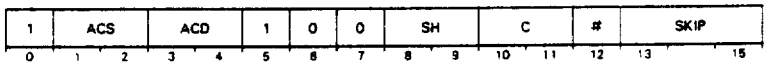
APPENDIX A

DATA GENERAL CORPORATION MANUAL NO.	TITLE
015-000 009	HOW TO USE THE NOVA COMPUTER
014-000 092	ECLIPSE M/600 PRINCIPLES OF OPERATION
014-000 629	INTERFACE DESIGN'S REFERENCE NOVA AND ECLIPSE LINE COMPUTERS
014-000 617	PROGRAMMER'S REFERENCE NOVA 4

APPENDIX B

Add Complement

ADC[*c*][*sh*][#] *acs,acd*[,*skip*]



Adds the logical complement of an unsigned integer to another unsigned integer.

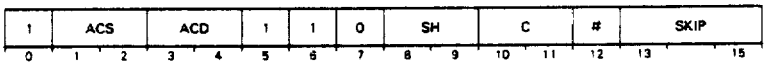
Initializes carry to the specified value, adds the logical complement of the unsigned, 16-bit number in bits 16-31 of ACS to the unsigned, 16-bit number in bits 16-31 of ACD, and places the result in the shifter. The instruction then performs the specified shift operation, and loads the result of the shift into bits 16-31 of ACD if the no-load bit is 0. If the skip condition is true, the next sequential word is skipped. For this instruction, *overflow* is 0.

If the load option is specified, bits 0-15 of ACD are undefined.

NOTE: *If the sum of the two numbers being added is greater than 65.535 the instruction complements carry.*

Add

ADD[*c*][*sh*][#] *acs,acd*[,*skip*]



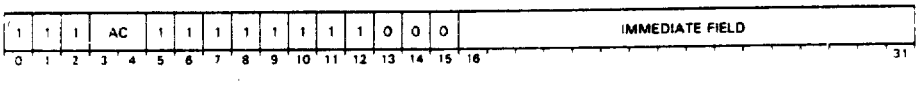
Performs unsigned integer addition and complements carry if appropriate.

Initializes carry to the specified value, adds the unsigned, 16-bit number in bits 16-31 of ACS to the unsigned, 16-bit number in bits 16-31 of ACD, and places the result in the shifter. The instruction then performs the specified shift operation and places the result of the shift in bits 16-31 of ACD if the no-load bit is 0. If the skip condition is true, the next sequential word is skipped. For this instruction, *overflow* is 0.
If the load option is specified, bits 0-15 of ACD are undefined.

NOTE: *If the sum of the two numbers being added is greater than 65.535, the instruction complements carry.*

Extended Add Immediate

ADDI *i,ac*



Adds a signed integer in the range -32,768 to +32,767 to the contents of an accumulator.

Treats the contents of the immediate field as a signed, 16-bit, two's complement number and adds it to the signed, 16-bit, two's complement number contained in bits 16-31 of the specified accumulator, placing the result in bits 16-31 of the same accumulator. Carry remains unchanged and *overflow* is 0.

Bits 0-15 of the modified accumulator are undefined after completion of this instruction.

Add Immediate

ADI *n,ac*

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

Adds an unsigned integer in the range 1–4 to the contents of an accumulator.

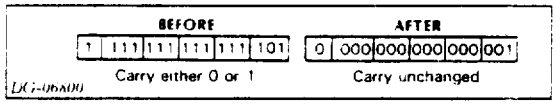
Adds the contents of the immediate field *N*, plus 1, to the unsigned, 16-bit number contained in bits 16-31 of the specified accumulator, placing the result in bits 16-31 of the same accumulator. Carry remains unchanged and *overflow* is 0.

Bits 0-15 of the modified accumulator are undefined after completion of this instruction.

> **NOTE:** *The assembler takes the coded value of n and subtracts 1 from it before placing it in the immediate field. Therefore, the programmer should code the exact value that he wishes to add.*

Example

Assume that AC2 contains 177775_8. After the instruction **ADI 4,2** is executed, AC2 contains 000001_8 and carry is unchanged.



AND With Complemented Source

ANC *acs,acd*

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

Forms the logical AND of the logical complement of the contents of bits 16-31 of ACS and the contents of bits 16-31 of ACD and places the result in bits 16-31 of ACD. The instruction sets a bit position in the result to 1 if the corresponding bit position in ACS contains 0. The contents of carry and ACS remain unchanged. *Overflow* is 0.

Bits 0-15 of the modified accumulator are undefined after completion of this instruction.

AND

AND*[c][sh][#]* *acs,acd[,skip]*

1	ACS	ACD	1	1	1	SH	C	#	SKIP
0	1 2	3 4	5	6	7	8 9	10 11	12	13 15

Forms the logical AND of the contents of two accumulators.

Initializes the carry bit to the specified value. Places the logical AND of bits 16-31 of ACS and bits 16-31 of ACD in the shifter. Each bit placed in the shifter is 1 only if the corresponding bit in both ACS and ACD is one; otherwise the resulting bit is 0. The instruction then performs the specified shift operation and places the result in bits 16-31 of ACD if the no-load bit is 0. If the skip condition is true, the next sequential word is skipped. *Overflow* is 0.

If the load option is specified, bits 0-15 of ACD are undefined.

AND Immediate

ANDI *i,ac*

1	1	0	AC	1	1	1	1	1	1	1	1	0	0	0	IMMEDIATE FIELD
0	1	2	3 4	5	6	7	8	9	10	11	12	13	14	15	16 31

Places the logical AND of the contents of the immediate field and the contents of bits 16-31 of the specified accumulator in bits 16-31 of the specified accumulator. Carry is unchanged and *overflow* is 0.

Bits 0-15 of the modified accumulator are undefined after completion of this instruction.

Block Add and Move

BAM

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

Moves memory words from one location to another, adding a constant to each one. Moves words sequentially from one memory location to another, treating them as unsigned, 16-bit integers. After fetching a word from the source location, the instruction adds the unsigned, 16-bit integer in bits 16-31 of AC0 to it. If the addition produces a carry of 1 out of the high-order bit, no indication is given.

Bits 17–31 of AC2 contain the address of the source location. Bits 17–31 of AC3 contain the address of the destination location. The address in bits 17–31 of AC2 or AC3 is an indirect address if bit 16 of that accumulator is 1. In that case, the instruction follows the indirection chain before placing the resultant effective address in the accumulator.

The unsigned, 16-bit number in bits 16-31 of AC1 is equal to the number of words moved. This number must be greater than 0 and less than or equal to 32,768. If the number in AC1 is outside these bounds, no data is moved and the contents of the accumulators remain unchanged.

AC	Contents
0	Addend
1	Number of words to be moved
2	Source address
3	Destination address

For each word moved, the count in AC1 is decremented by one and the source and destination addresses in AC2 and AC3 are incremented by one. Upon completion of the instruction, AC1 contains zeroes, and AC2 and AC3 point to the word following the last word in their respective fields. The contents of carry and AC0 remain unchanged. *Overflow* is 0.

The 32-bit effectibe address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

Words are moved in consecutive, ascending order according to their addresses. The next address after 77777_8 is 0 for both fields. The fields may overlap in any way.

> NOTE: *Because of the potentially long time that may be required to perform this instruction it is interruptable. If a* Block Add and Move *instruction is interrupted, the program counter is decremented by one before it is placed in location 0 so that it points to the interrupted instruction. Because the addresses and the word count are updated after every word stored, any interrupt service routine that returns control to the interrupted program via the address stored in memory location 0 will correctly restart the* Block Add and Move *instruction.*

When updating the source and destination addresses, the *Block Add And Move* instruction forces bit 0 of the result to 0. This ensures that upon return from an interrupt, the *Block Add And Move* instruction will not try to resolve an indirect address in either AC2 or AC3.

Breakpoint

BKPT

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

Pushes a wide return block onto the present stack.
The value of the PC in the return block is the address of this instruction. After pushing the block, the instruction checks for stack overflow. If no overflow occurred, the instruction sets the PSR to zero and performs a wide jump indirect through locations $10–11_8$ in page zero of the current segment. If overflow occurred, a stack fault occurs and AC1 contains the code 0; after the fault is handled, the PSR is set to zero and the jump indirect occurs. Carry remains unchanged by this instruction.

Block Move

BLM

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

Moves memory words from one location to another.

The *Block Move* instruction is the same as the *Block Add And Move* instruction in all respects except that no addition is performed and AC0 is not used. Carry remains unchanged and *overflow* is 0.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

> NOTE: *The* Block Move *instruction is interruptable in the same manner as the* Block Add And Move *instruction.*

Set Bit To One

BTO *acs,acd*

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

Sets the specified bit to 1.

Forms a 32-bit bit pointer from the contents of bits 16-31 of both ACS and ACD. Bits 16-31 of ACS contains the high-order 16 bits and bits 16-31 of ACD contains the low-order 16 bits of the bit pointer. If ACS and ACD are specified as the same accumulator, the instruction treats the accumulator contents as the low-order 16-bits of the bit pointer and assumes the high-order 16 bits are 0. Carry remains unchanged and *overflow* is 0.

The instruction then sets the addressed bit in memory to 1, leaving the contents of ACS and ACD unchanged.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

NOTE: *The bit pointer contained in ACS and ACD must not make indirect memory references.*

Set Bit To Zero

BTZ *acs,acd*

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

Sets the addressed bit to 0.

Forms a 32-bit bit pointer from the contents of bits 16-31 of both ACS and ACD. Bits 16-31 of ACS contains the high-order 16 bits and bits 16-31 of ACD contains the low-order 16 bits of the bit pointer. If ACS and ACD are specified as the same accumulator, the instruction treats the accumulator contents as the low-order 16 bits of the bit pointer and assumes the high-order 16 bits are 0. Carry remains unchanged and *overflow* is 0.

The instruction then sets the addressed bit in memory to 0, leaving the contents of ACS and ACD unchanged.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

NOTE: *The bit pointer contained in ACS and ACD must not make indirect memory references.*

Compare To Limits

CLM *acs,acd*

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

Compares a signed integer with two other integers and skips if the first integer is between the other two. The accumulators determine the location of the three integers.

Compares the 16-bit, signed, two's complement integer in bits 16-31 of ACS to two 16-bit, signed, two's complement limit values, *L* and *H*. If the number in bits 16-31 of ACS is greater than or equal to *L* and less than or equal to *H*, the next sequential word is skipped. If the number in bits 16-31 of ACS is less than *L* or greater than *H*, the next sequential word is executed.

If ACS and ACD are specified as different accumulators, the address of the limit value *L* is contained in bits 16–31 of ACD. The limit value *H* is contained in the word following *L*. Bits 0-15 of ACD are ignored.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

If ACS and ACD are specified as the same accumulator, then the integer to be compared must be in that accumulator and the limit values *L* and *H* must be in the two words following the instruction. *L* is the first word and *H* is the second word. The next sequential word is the third word following the instruction.

When *L* and *H* are in line, this instruction can be placed anywhere in the 32-bit address space.

This instruction leaves carry unchanged; *overflow* is 0.

Character Compare

CMP

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

Under control of the four accumulators, compares two strings of bytes and returns a code in AC1 reflecting the results of the comparison.

The instruction compares the strings one byte at a time. Each byte is treated as an unsigned 8-bit binary quantity in the range $0–255_{10}$. If two bytes are not equal, the string whose byte has the smaller numerical value is, by definition, the *lower valued* string. Both strings remain unchanged. The four accumulators contain parameters passed to the instruction. Two accumulators specify the starting address, the number of bytes, and the direction of processing (ascending or descending addressed) for each string.

Bits 16-31 of AC0 specify the length and direction of comparison for string 2. If the string is to be compared from its lowest memory location to the highest, bits 16-31 of AC0 contain the unsigned value of the number of bytes in string 2. If the string is to be compared from its highest memory location to the lowest, bits 16-31 of AC0 contain the two's complement of the number of bytes in string 2.

Bits 16-31 of AC1 specify the length and direction of comparison for string 1. If the string is to be compared from its lowest memory location to the highest, bits 16-31 of AC0 contain the unsigned value of the number of bytes in string 1. If the string is to be compared from its highest memory location to the lowest, bits 16-31 of AC1 contain the two's complement of the number of bytes in string 1.

Bits 16-31 of AC2 contain a byte pointer to the first byte compared in string 2. When the string is compared in ascending order, AC2 points to the lowest byte. When the string is compared in descending order, AC2 points to the highest byte.

Bits 16-31 of AC3 contain a byte pointer to the first byte compared in string 1. When the string is compared in ascending order, AC3 points to the lowest byte. When the string is compared in descending order, AC3 points to the highest byte.

Code	Comparison Result
- 1	string 1 < string 2
0	string 1 = string 2
+ 1	string 1 > string 2

The strings may overlap in any way. Overlap will not effect the results of the comparison.

Upon completion, bits 16-31 of AC0 contain the number of bytes left to compare in string 2. AC1 contains the return code as shown in the table above. Bits 16-31 of AC2 contains a byte pointer either to the failing byte in string 2 (if an inequality were found), or to the byte following string 2 (if string 2 were exhausted). Bits 16-31 of AC3 contains a byte pointer either to the failing byte in string 1 (if an inequality were found), or to the byte following string 1 (if string 1 were exhausted). Carry remains unchanged. *Overflow* is 0.

If AC0 and AC1 both contain 0 (both string 1 and string 2 have length zero), the instruction compares no bytes and returns 0 in AC1. If the two strings are of unequal length, the instruction pads the shorter string with space characters $<040_8>$ and continues the comparison.

The 32-bit effective address generated by this instruction is constrained to be within the first 64 Kbyte of the current segment.

> NOTE: *The original contents of AC2 and AC3 must be valid byte pointers to an area in the user's address space. If the pointers are invalid a protection fault occurs, even if no bytes are to be compared. AC1 contains the code 2.*

Character Move Until True
CMT

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

Under control of the four accumulators, moves a string of bytes from one area of memory to another until either a table-specified delimiter character is moved or the source string is exhausted.

The instruction copies the string one byte at a time. Before it moves a byte, the instruction uses that byte's value to determine if it is a delimiter. It treats the byte as an unsigned 8-bit binary integer (in the range $0–255_{10}$) and uses it as a bit index into a 256-bit delimiter table. If the indexed bit in the delimiter table is zero, the byte pending is not a delimiter, and the instruction copies it from the source string to the destination string. If the indexed bit in the delimiter table is 1, the byte pending is a delimiter; the instruction does not copy it, and the instruction terminates.

The instruction processes both strings in the same direction, either from lowest memory locations to highest (*ascending order*), or from highest memory locations to lowest (*descending order*). Processing continues until there is a delimiter or the source string is exhausted. The four accumulators contain parameters passed to the instruction.

Bits 16-31 of AC0 contain the address (word address), possibly indirect, of the start of the 256-bit (16-word) delimiter table.

Bits 16-31 of AC1 specify the length of the strings and the direction of processing. If the source string is to be moved to the destination field in ascending order, bits 16-31 of AC1 contain the unsigned value of the number of bytes in the source string. If the source string is to be moved to the destination field in descending order, bits 16-31 of AC1 contain the two's complement of the number of bytes in the source string.

Bits 16-31 of AC2 contain a byte pointer to the first byte to be written in the destination field. When the process is performed in ascending order, bits 16-31 of AC2 point to the lowest byte in the destination field. When the process is performed in descending order, bits 16-31 of AC2 point to the highest byte in the destination field.

Bits 16-31 of AC3 contain a byte pointer to the first byte to be processed in the source string. When the process is performed in ascending order, bits 16-31 of AC3 point to the lowest byte in the source string. When the process is performed in descending order, bits 16-31 of AC3 point to the highest byte in the source string.

The fields may overlap in any way. However, the instruction moves bytes one at a time, so certain types of overlap may produce unusual side effects.

Upon completion, bits 16-31 of AC0 contain the resolved address of the translation table and AC1 contain the number of bytes that were not moved. Bits 16-31 of AC2 contain a byte pointer to the byte following the last byte written in the destination field. Bits 16-31 of AC3 contain a byte pointer either to the delimiter or to the first byte following the source string. Carry remains unchanged. *Overflow* is 0.

The 32-bit effective address generated by this instruction is constrained to be within the first 64 Kword of the current segment.

> **NOTE:** *If AC1 contains the number 0 at the beginning of this instruction, no bytes are fetched and none are stored. The instruction becomes a No-Op.*

> **NOTE:** *The original contents of AC0, AC2, and AC3 must be valid pointers to some area in the user's address space. If they are invalid a protection fault occurs, even if no bytes are to be moved. AC1 contains the code 2.*

Character Move
CMV

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

Under control of the four accumulators, moves a string of bytes from one area of memory to another and returns a value in carry reflecting the relative lengths of source and destination strings.

The instruction copies the source string to the destination field, one byte at a time. The four accumulators contain parameters passed to the instruction. Two accumulators specify the starting address, number of bytes to be copied, and the direction of processing (ascending or descending addresses) for each field.

Bits 16-31 of AC0 specify the length and direction of processing for the destination field. If the field is to be processed from its lowest memory location to the highest, bits 16-31 of AC0 contain the unsigned value of the number of bytes in the destination field. If the field is to be processed from its highest memory location to the lowest, bits 16-31 of AC0 contain the two's complement of the number of bytes in the destination field.

Bits 16-31 of AC1 specify the length and direction of processing for the source string. If the string is to be processed from its lowest memory location to the highest, bits 16-31 of AC1 contain the unsigned value of the number of bytes in the source string. If the field is to be processed from its highest memory location to the lowest, bits 16-31 of AC1 contain the two's complement of the number of bytes in the source string.

Bits 16-31 of AC2 contain a byte pointer to the first byte to be written in the destination field. When the field is written in ascending order, bits 16-31 of AC2 point to the lowest byte. When the field is written in descending order, bits 16-31 of AC2 point to the highest byte.

Bits 16-31 of AC3 contain a byte pointer to the first byte copied in the source string. When the field is copied in ascending order, bits 16-31 of AC3 point to the lowest byte. When the field is copied in descending order, bits 16-31 of AC3 point to the highest byte. The fields may overlap in any way. However, the instruction moves bytes one at a time, so certain types of overlap may produce unusual side effects.

Upon completion, AC0 contains 0 and bits 16-31 of AC1 contain the number of bytes left to fetch from the source field. Bits 16-31 of AC2 contain a byte pointer to the byte following the destination field; bits 16-31 of AC3 contain a byte pointer to the byte following the last byte fetched from the source field. *Overflow* is 0.

The 32-bit effective address generated by this instruction is constrained to be within the first 64Kbyte of the current segment.

> NOTE: *If AC0 contains the number 0 at the beginning of this instruction, no bytes are fetched and none are stored. If AC1 is 0 at the beginning of this instruction, the destination field is filled with space characters.*

> NOTE: *The original contents of AC2 and AC3 must be valid pointers to some area in the user's address space. If they are invalid a protection fault occurs, even if no bytes are to be moved. AC1 contains the code 2.*

If the source field is longer than the destination field, the instruction terminates when the destination field is filled and sets carry to 1. In any other case, the instruction sets carry to 0.

If the source field is shorter than the destination field, the instruction pads the destination field with space characters $<040_8>$.

Count Bits

COB *acs,acd*

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

Adds a number equal to the number of ones in bits 16-31 of ACS to the signed, 16-bit, two's complement number in bits 16-31 of ACD. The instruction leaves the contents of ACS and the state of carry unchanged. *Overflow* is 0.

Bits 0-15 of the modified accumulator are undefined after completion of this instruction.

> NOTE: *If ACS and ACD are the same accumulator, the instruction functions as described above, except the contents of ACS will be changed.*

Complement

COM[*c*][*sh*][#] *acs,acd*[,*skip*]

1	ACS	ACD	0	0	0	SH	C	#	SKIP
0	1 2	3 4	5	6	7	8 9	10 11	12	13 15

Forms the logical complement of the contents of an accumulator.

Initializes carry to the specified value, forms the logical complement of the number in bits 16-31 of ACS, and performs the specified shift operation. The instruction then places the result in bits 16-31 of ACD if the no-load bit is 0. If the skip condition is true, the next sequential word is skipped.
If the load option is specified, bits 0-15 of ACD are undefined.

For this instruction, *overflow* is 0.

Complement Carry

CRYTC

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

Complements the value of carry. *Overflow* is 0.

Set Carry to One

CRYTO

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

Unconditionally sets the value of carry to 1. *Overflow* is 0.

Set Carry to Zero

CRYTZ

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

Unconditionally sets the value of carry to 0. *Overflow* is 0.

Character Translate

CTR

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

Under control of the four accumulators, translates a string of bytes from one data representation to another and either moves it to another area of memory or compares it to a second translated string.

The instruction operates in two modes; translate and move, and translate and compare.

When operating in translate and move mode, the instruction translates each byte in string 1, and places it in a corresponding position in string 2. Translation is performed by using each byte as an 8-bit index into a 256-byte translation table. The byte addressed by the index then becomes the translated value.

When operating in translate and compare mode, the instruction translates each byte in string 1 and string 2 as described above, and compares the translated values. Each translated byte is treated as an unsigned 8-bit binary quantity in the range $0\text{-}255_{10}$. If two translated bytes are not equal, the string whose byte has the smaller numerical value is, by definition the *lower valued* string. Both strings remain unchanged.

Bits 16-31 of AC0 specify the address, either direct or indirect, of a word which contains a byte pointer to the first byte in the 256-byte translation table.

Bits 16-31 of AC1 specify the length of the two strings and the mode of processing. If string 1 is to be processed in translate and move mode, bits 16-31 of AC1 contain the two's complement of the number of bytes in the strings. If the strings are to be processed in translate and compare mode, bits 16-31 of AC1 contain the unsigned value of the number of bytes in the strings. Both strings are processed from lowest memory address to highest.

Bits 16-31 of AC2 contain a byte pointer to the first byte in string 2.

Bits 16-31 of AC3 contain a byte pointer to the first byte in string 1.

Upon completion of a translate and move operation, bits 16-31 of AC0 contain the address of the word which contains the byte pointer to the translation table and AC1 contains 0. Bits 16-31 of AC2 contain a byte pointer to the byte following string 2 and bits 16-31 of AC3 contain a byte pointer to the byte following string 1. Carry remains unchanged. *Overflow* is 0.

Upon completion of a translate and compare operation, bits 16-31 of AC0 contain the address of the word which contains the byte pointer to the translation table. AC1 contains a return code as calculated in the table below. Bits 16-31 of AC2 contain a byte pointer to either the failing byte in string 2 (if an inequality was found) or the byte following string 2 if the strings were identical. Bits 16-31 of AC3 contain a byte pointer to either the failing byte in string 1 (if an inequality was found) or the byte following string 1 if the strings were identical. Carry contains an indeterminate value. *Overflow* is 0.

The 32-bit effective address generated by this instruction is constrained to be within the first 64 Kbyte of the current segment.

Code	Result
-1	Translated value of string 1 < Translated value of string 2
0	Translated value of string 1 = Translated value of string 2
+1	Translated value of string 1 > Translated value of string 2

If the length of both string 1 and string 2 is zero, the compare option returns a 0 in AC1.

The fields may overlap in any way. However, processing is done one character at a time, so unusual side effects may be produced by certain types of overlap.

NOTE: *The original contents of AC0, AC2, and AC3 must be valid byte pointers to some area in the user's address space. If they are invalid a protection fault occurs, even if no bytes are to be moved or compared. AC1 contains the code 2.*

Convert to 16-Bit Integer

CVWN *ac*

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

Converts a 32-bit integer to a 16-bit integer.

The instruction converts the 32-bit contents of the specified accumulator to a 16-bit integer by extending bit 17 into bits 0–16. If the 17 most significant bits do not contain the same value (i.e., all 1's or all 0's) before conversion takes place, then this instruction sets *overflow* to 1 before performing the conversion. Carry is unchanged.

Decimal Add

DAD *acs,acd*

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

Performs decimal addition on 4-bit binary coded decimal (BCD) numbers and uses carry for a decimal carry.

Adds the unsigned decimal digit contained in bits 28–31 of ACS to the unsigned decimal digit contained in bits 28–31 of ACD. Carry is added to this result. The instruction then places the decimal units' position of the final result in bits 28–31 of ACD, and the decimal carry in carry. The contents of ACS and bits 0–27 of ACD remain unchanged. *Overflow* is 0.

NOTE: *No validation of the input digits is performed. Therefore, if bits 28–31 of either ACS or ACD contain a number greater than 9, the results will be unpredictable.*

Example

Assume that bits 28–31 of AC2 contain 9; bits 28–31 of AC3 contain 7; and the carry bit is 0. After the instruction **DAD 2,3** is executed, AC2 remains the same; bits 28–31 of AC3 contain 6; and carry is 1, indicating a decimal carry from this *Decimal Add.*

	BEFORE	AFTER
AC2	0 000 000 000 001 001	0 000 000 000 001 001
AC3	0 000 000 000 000 111	0 000 000 000 000 110
	Carry = 0	Carry = 1

DG-06798

Double Hex Shift Left

DHXL *n,ac*

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

Shifts the 32-bit number contained in bits 16-31 of AC and bits 16-31 of AC+1 left a number of hex digits depending upon the immediate field *N*. The number of digits shifted is equal to *N*+1. Bits shifted out are lost and the vacated bit positions are filled with zeroes. Carry remains unchanged and *overflow* is 0.

Bits 0-15 of the modified accumulator are undefined after completion of this instruction.

NOTE: *If AC is specified as AC3, then AC+1 is AC0.*

NOTE: *The assembler takes the coded value of* n *and subtracts one from it before placing it in the immediate field. Therefore, the programmer should code the exact number of hex digits that he wishes to shift.*

If N *is equal to 3, the contents of AC+1 are placed in AC and AC+1 is filled with zeroes.*

Double Hex Shift Right

DHXR *n,ac*

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

Shifts the 32-bit number contained in bits 16-31 of AC and bits 16-31 of AC+1 right a number of hex digits depending upon the immediate field *N*. The number of digits shifted is equal to *N*+1. Bits shifted out are lost and the vacated bit positions are filled with zeroes. Carry remains unchanged and *overflow* is 0.

Bits 0-15 of the modified accumulator are undefined after completion of this instruction.

NOTE: *If AC is specified as AC3, then AC+1 is AC0.*

NOTE: *The assembler takes the coded value of* n *and subtracts one from it before placing it in the immediate field. Therefore, the programmer should code the exact number of hex digits that he wishes to shift.*

If N *is equal to 3, the contents of AC are placed in AC+1 and AC is filled with zeroes.*

Unsigned Divide

DIV

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

Divides the unsigned 32-bit integer in bits 16-31 of two accumulators by the unsigned contents of a third accumulator. The quotient and remainder each occupy one accumulator.

Divides the unsigned 32-bit number contained in bits 16-31 of AC0 and bits 16-31 of AC1 by the unsigned, 16-bit number in bits 16-31 of AC2. The quotient and remainder are unsigned, 16-bit numbers and are placed in bits 16-31 of AC1 and AC0, respectively. Carry is set to 0. The contents of AC2 remain unchanged. *Overflow* is 0.

Bits 0-15 of the modified accumulator are undefined after completion of this instruction.

NOTE: *Before the divide operation takes place, the number in bits 16-31 of AC0 is compared to the number in bits 16-31 of AC2. If the contents of bits 16-31 of AC0 are greater than or equal to the contents of bits 16-31 of AC2, an overflow condition is indicated. Carry is set to 1, and the operation is terminated. All operands remain unchanged.*

Signed Divide

DIVS

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

Divides the signed 32-bit integer in bits 16-31 of two accumulators by the signed contents of a third accumulator. The quotient and remainder each occupy one accumulator.

The signed, 32-bit two's complement number contained in bits 16-31 of AC0 and bits 16-31 of AC1 is divided by the signed, 16-bit two's complement number in bits 16-31 of AC2. The quotient and remainder are signed, 16-bit numbers and occupy bits 16-31 of AC1 and AC0, respectively. The sign of the quotient is determined by the rules of algebra. The sign of the remainder is always the same as the sign of the dividend, except that a zero quotient or a zero remainder is always positive. Carry is set to 0. The contents of AC2 remain unchanged. *Overflow* is 0.

Bits 0-15 of the modified accumulator are undefined after completion of this instruction.

NOTE: *If the magnitude of the quotient is such that it will not fit into bits 16-31 of AC1, an overflow condition is indicated. Carry is set to 1, and the operation is terminated. The contents of AC0 and AC1 are unpredictable.*

Sign Extend and Divide

DIVX

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

Extends the sign of one accumulator into a second accumulator and performs a *Signed Divide* on the result.

Extends the sign of the 16-bit number in bits 16-31 of AC1 into bits 16-31 of AC0 by placing a copy of bit 16 of AC1 in bits 16-31 of of AC0. After extending the sign, the instruction performs a *Signed Divide* operation. *Overflow* is 0.

Bits 0-15 of the modified accumulator are undefined after completion of this instruction.

Double Logical Shift

DLSH *acs,acd*

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

Shifts the 32-bit number contained in bits 16-31 of ACD and bits 16-31 of ACD+1 either left or right depending on the number contained in bits 24–31 of ACS. The signed, 8-bit two's complement number contained in bits 24–31 of ACS determines the direction of the shift and the number of bits to be shifted. If the number in bits 24–31 of ACS is positive, shifting is to the left; if the number in bits 24–31 of ACS is negative, shifting is to the right. If the number in bits 24–31 of ACS is zero, no shifting is performed. Bits 0–23 of ACS are ignored.

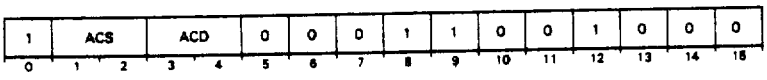
AC3+1 is AC0. The number of bits shifted is equal to the magnitude of the number in bits 24–31 of ACS. Bits shifted out are lost, and the vacated bit positions are filled with zeroes. Carry and the contents of ACS remain unchanged. *Overflow* is 0.

Bits 0-15 of the modified accumulator are undefined after completion of this instruction.

NOTE: *If the magnitude of the number in bits 24–31 of ACS is greater than 31_{10}, bits 16-31 of ACD are set to 0. Carry and the contents of ACS remain unchanged.*

Decimal Subtract

DSB *acs,acd*

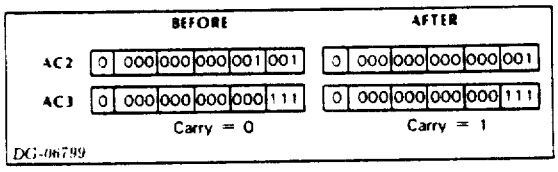


Performs decimal subtraction on 4-bit binary coded decimal (BCD) numbers and uses carry as a decimal borrow.

Subtracts the unsigned decimal digit contained in ACS bits 28–31 from the unsigned decimal digit contained in ACD bits 28–31. Subtracts the complement of carry from this result. Places the decimal units' position of the final result in ACD bits 28–31 and the complement of the decimal borrow in carry. In other words, if the final result is negative, the instruction indicates a borrow and sets carry to 0. If the final result is positive, the instruction indicates no borrow and sets carry to 1. The contents of ACS and bits 0–27 of ACD remain unchanged. *Overflow* is 0.

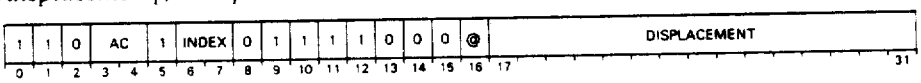
Example

Assume that bits 28–31 of AC2 contain 9; bits 28–31 of AC3 contain 7; and carry contains 0. After the instruction **DSB 3,2** is executed, AC3 remains the same; bits 28–31 of AC2 contain 1; and carry is set to 1, indicating no borrow from this *Decimal Subtract*.



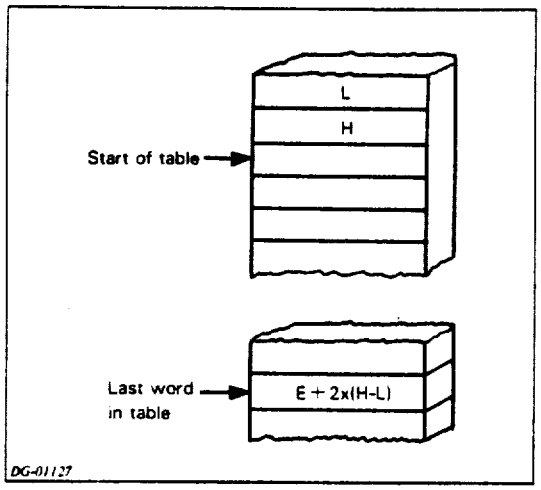
Dispatch

DSPA *ac,[@]displacement[,index]*



Conditionally transfers control to an address selected from a table.

Computes the effective address *E*. This is the address of a *dispatch table*. The dispatch table consists of a table of addresses. Immediately before the table are two 16-bit, signed, two's complement limit words, *L* and *H*. The last word of the table is in location *E+H-L*.



Compares the signed, two's complement number contained in bits 16-31 of the specified accumulator to the limit words. If the number in the accumulator is less than *L* or greater than *H*, sequential operation continues with the instruction immediately after the *Dispatch* instruction.

If the number in bits 16-31 of the specified accumulator is greater than or equal to *L* and less than or equal to *H*, the instruction fetches the word at location *E-L+number*. If the fetched word is equal to 177777_8, sequential operation continues with the instruction immediately after the *Dispatch* instruction. If the fetched word is not equal to 177777_8, the instruction treats this word as the intermediate address in the effective address calculation. After the indirection chain, if any, has been followed, the instruction places the effective address in the program counter and sequential operation continues with the word addressed by the updated value of the program counter.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

This instruction sets *overflow* to 0 and carry to 0.

Decrement And Skip If Zero

DSZ *[@]displacement[,index]*

0	0	0	1	1	@	INDEX	DISPLACEMENT
0	1	2	3	4	5	6 7	8 15

Decrements the addressed word, then skips if the decremented value is zero. Decrements by one the word addressed by *E* and writes the result back into that location. If the updated value of the location is zero, the instruction skips the next sequential word. *Overflow* is 0 and carry remains unchanged.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

Decrement the Word Addressed by WSP and Skip if Zero

DSZTS

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

Uses the contents of WSP (the wide stack pointer) as the address of a double word. Decrements the contents of the word addressed by WSP. If the decremented value is equal to zero, the instruction skips the next word. Carry is unchanged and *overflow* is 0.

NOTE: *The operation performed by this instruction is not indivisible.*

Load CPU Identification

ECLID

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

Loads a double word into AC0.

The double word has the format:

MODEL NUMBER	MICROCODE REV		MEM SIZE
0 ... 15	16 ... 23	24 25	26 ... 31

where

model # is the binary representation of the machine's model number,
microcode rev indicates the microcode revision currently in use on this machine,
mem size indicates the amount of physical memory on this machine. A zero in this field indicates 256 Kbytes of memory; a one indicates 512 Kbytes, and so on.

This instruction leaves carry unchanged. *Overflow* is 0.

NOTE: *When the C/350 MAP is enabled on the MV/8000 this instruction is used to identify the machine. The processor assumes AC0 to be 32 bits long for this instruction. If an interrupt occurs while* ECLID *is executing, however, the processor saves only bits 16–31 of AC0.*

Edit

EDIT

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

Converts a decimal number from either packed or unpacked form to a string of bytes under the control of an edit sub-program. This sub-program can perform many different operations on the number and its destination field, including leading zero suppression, leading or trailing signs, floating fill characters, punctuation control, and insertion of text into the destination field. The instruction also performs operations on alphanumeric data if data type 4 is specified.

The instruction maintains two flags and three indicators or pointers.

The flags are the significance Trigger (*T*) and the Sign flag (*S*). *T* is set to 1 when the first non-zero digit is processed unless otherwise specified by an edit op-code. At the beginning of an *Edit* instruction, *T* is set to 0. *S* is set to reflect the sign of the number being processed. If the number is positive, *S* is set to 0. If the number is negative, *S* is set to 1.

The three indicators are the Source Indicator (SI), the Destination Indicator (DI), and the op-code Pointer (P). Each is 16 bits wide and contains a byte pointer to the *current* byte in each respective area. At the beginning of an *Edit* instruction, SI is set to the value contained in bits 16-31 of AC3. DI is set to the value contained in bits 16-31 of AC2, and P is set to the value contained in bits 16-31 of AC0. Also at this time the sign of the source number is checked for validity.

The sub-program is made up of 8-bit op-codes followed by one or more 8-bit operands. P, a byte pointer, acts as the *program counter* for the *Edit* sub-program. The sub-program proceeds sequentially until a branching operation occurs - much the same way programs are processed. Unless instructed to do otherwise, the *Edit* instruction updates P after each operation to point to the next sequential op-code. The instruction continues to process 8-bit op-codes until directed to stop by the **DEND** op-code.

The sub-program can test and modify *S* and *T* as well as modify SI, DI and P.

Upon entry to **EDIT** bits 16-31 of AC0 contain a byte pointer to the first op-code of the *Edit* sub-program.

Bits 16-31 of AC1 contain the data-type indicator describing the number to be processed.

Bits 16-31 of AC2 contain a byte pointer to the the first byte of the destination field.

Bits 16-31 of AC3 contain a byte pointer to the first byte of the source field.

The fields may overlap in any way. However the instruction processes characters one at a time, so unusual side effects may be produced by certain types of overlap.

Upon successful termination, carry contains the significance Trigger; bits 16-31 of AC0 contain a byte pointer (P) to the next op-code to be processed; AC1 is undefined; bits 16-31 of AC2 contain a byte pointer (DI) to the next destination byte; and bits 16-31 of AC3 contain a byte pointer (SI) to the next source byte. *Overflow* is 0.

The 32-bit effective address generated by this instruction is constrained to be within the first 64 Kbyte of the current segment.

> NOTES: *If SI is moved outside the area occupied by the source number, zeros will be supplied for numeric moves, even if SI is later moved back inside the source area.*
>
> *Some op-codes perform movement of characters from one string to another. For those op-codes which move numeric data, special actions may be performed. For those which move non-numeric data, characters are copied exactly to the destination.*
>
> *The* Edit *instruction places information on the stack. Therefore, the stack must be set up and have at least 9 words available for use.*
>
> *If the* Edit *instruction is interrupted, it places restart information on the stack and places* 177777_8 *in AC0.*
>
> *If the initial contents of AC0 are equal to* 177777_8 *the* Edit *instruction assumes it is restarting from an interrupt; therefore do not allow this to occur under any other circumstances.*

In the description of some of the *Edit* op-codes we use the symbol *j* to denote how many characters a certain operation should process. When the high order bit of *j* is 1, *j* has a different meaning, it is a pointer into the stack to a word that denotes the number of characters the instruction should process. So, in those cases where the high order bit of *j* is 1, the instructions interpret *j* as an 8-bit two's complement number pointing into the stack. The number on the stack is at address:

$$\textit{stack pointer} + 1 + j\ ).$$

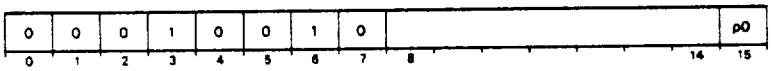
The operation uses the number at this address as a character count instead of *j*.

An *Edit* operation that processes numeric data (e.g., DMVN) skips a leading or trailing sign code it encounters; similarly, such an operation converts a high-order or low-order sign to its correct numeric equivalent.

The edit operations are as follows.

Add To DI

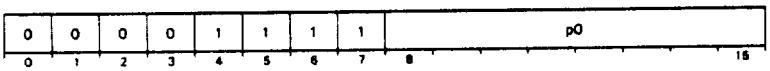
DADI *p0*



Adds the 8-bit two's complement integer specified by *p0* to the Destination Indicator (DI).

Add To P Depending On S

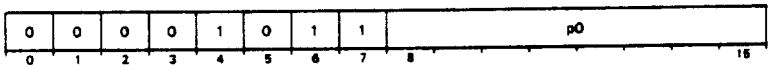
DAPS *p0*



If *S* is 0, the instruction adds the 8-bit two's complement integer specified by *p0* to the op-code Pointer (P). Before the add is performed, P is pointing to the byte containing the DAPS op-code.

Add To P Depending On T

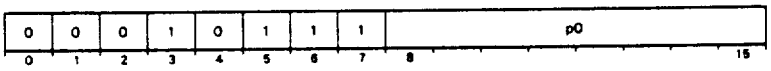
DAPT *p0*



If *T* is one, the instruction adds the 8-bit two's complement integer specified by *p0* to the op-code Pointer (P). Before the add is performed, P is pointing to the byte containing the DAPT op-code.

Add To P

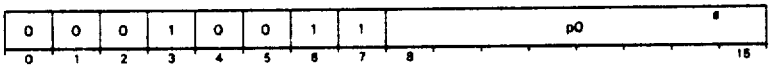
DAPU *p0*



Adds the 8-bit two's complement integer specified by *p0* to the op-code Pointer (P). Before the add is performed, P is pointing to the byte containing the DAPU op-code.

Add To SI

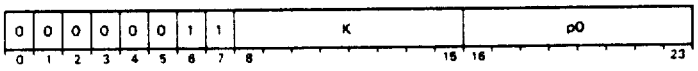
DASI *p0*



Adds the 8-bit two's complement integer specified by *p0* to the Source Indicator (SI).

Decrement and Jump If Non-Zero

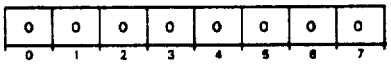
DDTK *k,p0*



Decrements a word in the stack by one. If the decremented value of the word is non-zero, the instruction adds the 8-bit two's complement integer specified by *p0* to the op-code Pointer (P). Before the add is performed, P is pointing to the byte containing the DDTK op-code. If the 8-bit two's complement integer specified by *k* is negative, the word decremented is at the address *stack pointer*+1+*k*. If *k* is positive, the word decremented is at the address *frame pointer*+1+*k*.

End Edit

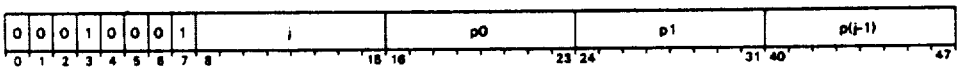
DEND



Terminates the EDIT sub-program.

Insert Characters Immediate

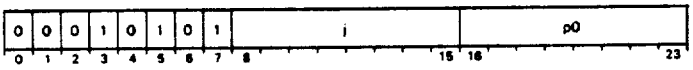
DICI *j,p0,p1,....,p(j-1)*



Inserts *j* characters from the op-code stream into the destination field beginning at the position specified by DI. Increases P by *j*+2, and increases DI by *j*.

Insert Character J Times

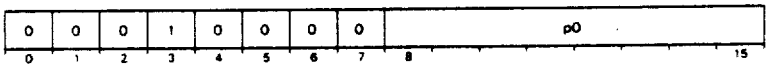
DIMC *j,p0*



Inserts the character specified by *p0* into the destination field a number of times equal to *j* beginning at the position specified by DI. Increases DI by *j*.

Insert Character Once

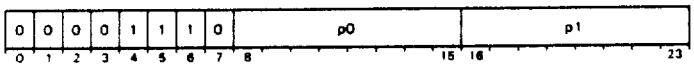
DINC *p0*



Inserts the character specified by *p0* in the destination field at the position specified by DI. Increments DI by 1.

Insert Sign

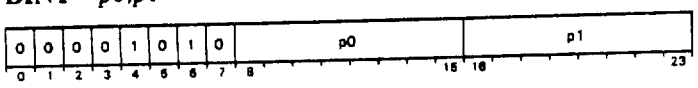
DINS *p0,p1*



If the Sign flag (*S*) is 0, the instruction inserts the character specified by *p0* in the destination field at the position specified by DI. If *S* is 1, the instruction inserts the character specified by *p1* in the destination field at the position specified by DI. Increments DI by one.

Insert Character Suppress

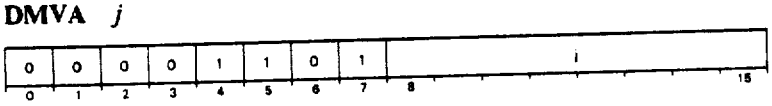
DINT *p0,p1*



If the significance Trigger (T) is 0, the instruction inserts the character specified by *p0* in the destination field at the position specified by DI. If T is 1, the instruction inserts the character specified by *p1* in the destination field at the position specified by DI. Increments DI by one.

Move Alphabetics

DMVA *j*

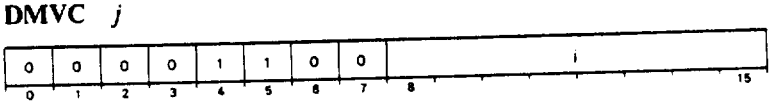


Moves j characters from the source field (beginning at the position specified by SI) to the destination field (beginning at the position specified by DI). Increases both SI and DI by j. Sets T to 1.

Initiates a commercial fault if the attribute specifier word indicates that the source field is data type 5 (packed). Initiates a commercial fault if any of the characters moved is not an alphabetic (A-Z, a-z, or space).

Move Characters

DMVC *j*

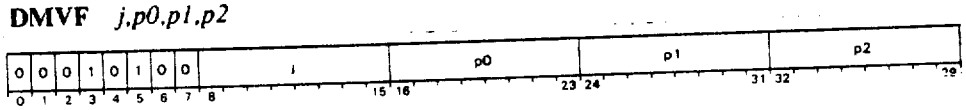


Increments SI if the source data type is 3 and $j>0$. The instruction then moves j characters from the source field beginning at the position specified by SI to the destination field beginning at the position specified by DI. Increases both SI and DI by j. Sets T to 1.

Initiates a commercial fault if the attribute specifier word indicates that the source is data type 5 (packed). Performs no validation of the characters.

Move Float

DMVF *j,p0,p1,p2*



If the source data type is 3, $j>0$, and SI points to the sign of the source number, the instruction increments SI. Then for j characters, the instruction either places a digit substitute in the destination field beginning at the position specified by DI, or it moves a digit from the source field beginning at the position specified by SI to the destination field beginning at the position specified by DI. When T changes from 0 to 1, the instruction places both the digit substitute and the digit in the destination field, and increases SI by j. If T does not change from 0 to 1, the instruction increases DI by j. If T does change from 0 to 1, the instruction increases DI by $j+1$.

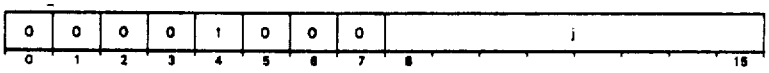
If the source data type is 2, the state of SI is undefined after the least significant digit has been processed.

If *T* is 1, the instruction moves each digit processed from the source field to the destination field. If *T* is 0 and the digit is a zero or space, the instruction places *p0* in the destination field. If *T* is 0 and the digit is a non-zero, the instruction sets *T* to 1 and the characters placed in the destination field depend on *S*. If *S* is 0, the instruction places *p1* in the destination field followed by the digit. If *S* is 1, the instruction places *p2* in the destination field followed by the digit.

The instruction initiates a commercial fault if any of the digits processed is not valid for the specified data type.

Move Numerics

DMVN *j*



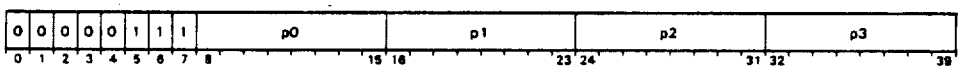
Increments SI if the source data type is 3 and $j>0$. The instruction then moves *j* characters from the source field beginning at the position specified by SI to the destination field beginning at the position specifed by DI. Increases both SI and DI by *j*. Sets *T* to 1.

Initiates a commercial fault if any of the characters moved is not valid for the specified data type.

For data type 2, the state of SI is undefined after the least significant digit has been processed.

Move Digit With Overpunch

DMVO *p0,p1,p2,p3*



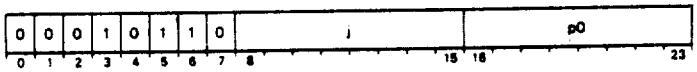
Increments SI if the source data type is 3 and SI points to the sign of the source number. The instruction then either places a digit substitute in the destination field (at the position specified by DI), or it moves a digit plus overpunch from the source field (at the position specified by SI) to the destination field (at the position specified by DI). Increases both SI and DI by 1.

If the source data type is 2, the state of the SI is undefined after the least significant digit has been processed.
If the digit is a zero or space and *S* is 0, then the instruction places *p0* in the destination field. If the digit is a zero or space and *S* is 1, then the instruction places *p1* in the destination field. If the digit is a non-zero and *S* is 0, the instruction adds *p2* to the digit and places the result in the destination field. If the digit is a non-zero and *S* is 1, the instruction adds *p3* to the digit and places the result in the destination field. If the digit is a non-zero, the instruction sets *T* to 1. The instructions assumes *p2* and *p3* are ASCII characters.

The instruction initiates a commercial fault if the character is not valid for the specified data type.

Move Numeric With Zero Suppression

DMVS *j,p0*



Increments SI if the source data type is 3, $j>0$, and SI points to the sign of the source number. The instruction then moves j characters from the source field (beginning at the position specified by SI) to the destination field (beginning at the position specified by DI). Moves the digit from the source to the destination if T is 1. Replaces all zeros and spaces with *p0* as long as T is 0. Sets T to 1 when the first non-zero digit is encountered. Increases both SI and DI by j.

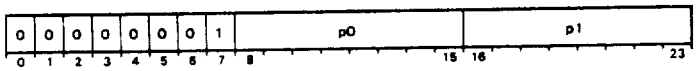
If the source data type is 2, the state of the SI is undefined after the least significant digit has been processed.

This op-code destroys the data type specifier.

Initiates a commercial fault if any of the characters moved is not a numeric (0-9 or space).

End Float

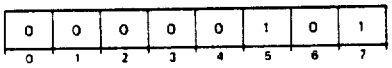
DNDF *p0,p1*



If T is 1, the instruction places nothing in the destination field and leaves DI unchanged. If T is 0 and S is 0, the instruction places *p0* in the destination field at the position specified by DI. If T is 0 and S is 1, the instruction places *p1* in the destination field at the position specified by DI. It increases DI by 1, and sets T to one.

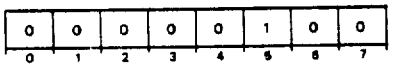
Set S To One

DSSO



Set S To Zero

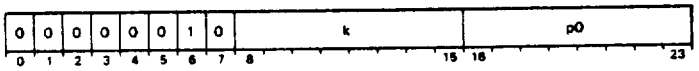
DSSZ



Sets the Sign flag (S) to 0.

Store In Stack

DSTK *k,p0*



Stores the byte specified by *p0* in bits 8–15 of a word in the stack. Sets bits 0–7 of the word that receives *p0* to 0. If the 8-bit two's complement integer specified by k is negative, the instruction addresses the word receiving *p0* by *stack pointer*$+1+k$. If k is positive, then the instruction stores *p0* at the address *frame pointer*$+1+k$.

Set T To One

DSTO

0	0	0	0	1	0	0	1
0	1	2	3	4	5	6	7

Sets the significance Trigger (*T*) to 1.

Set T To Zero

DSTZ

0	0	0	0	0	1	1	0
0	1	2	3	4	5	6	7

Sets the significance Trigger (*T*) to 0.

Extended Decrement and Skip if Zero

EDSZ *[@]displacement[,index]*

1	0	0	1	1	1	INDEX	0	0	1	1	1	0	0	0	@	DISPLACEMENT
0	1	2	3	4	5	6 7	8	9	10	11	12	13	14	15	16	17 31

Decrements the addressed word, then skips if the decremented value is zero. Computes the effective address, *E*. Decrements by one the contents of the location addressed by *E* and writes the result back into that location. If the updated value of the word is zero, the instruction skips the next sequential word.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

This instruction leaves carry unchanged. *Overflow* is 0.

Extended Increment And Skip If Zero

EISZ *[@]displacement[,index]*

1	0	0	1	0	1	INDEX	0	0	1	1	1	0	0	0	@	DISPLACEMENT
0	1	2	3	4	5	6 7	8	9	10	11	12	13	14	15	16	17 31

Increments the addressed word, then skips if the incremented value is zero.

Computes the effective address, *E* . Increments by one the contents of the location specified by *E*, and writes the new value back into memory at the same address. If the updated value of the location is zero, the instruction skips the next sequential word.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

This instruction leaves carry unchanged. *Overflow* is 0.

Extended Jump

EJMP *[@]displacement[,index]*

1	0	0	0	0	1	INDEX	0	0	1	1	1	0	0	0	@	DISPLACEMENT
0	1	2	3	4	5	6 7	8	9	10	11	12	13	14	15	16	17 31

Computes the effective address, *E*, and places it in the program counter. Sequential operation continues with the word addressed by the updated value of the program counter.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

Carry is unchanged and *overflow* is 0.

Extended Jump To Subroutine

EJSR *[@]displacement[,index]*

1	0	0	0	1	1	INDEX	0	0	1	1	1	0	0	0	@	DISPLACEMENT
0	1	2	3	4	5	6 7	8	9	10	11	12	13	14	15	16	17 31

Increments and stores the value of the program counter in AC3, then places a new address in the program counter.
Computes the effective address, *E*. The instruction then places the address of the next sequential instruction (the instruction following the **EJSR** instruction) in AC3. Places *E* in the program counter. Sequential operation continues with the word addressed by the updated value of the program counter.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

Overflow is 0 and carry is unchanged.

NOTE: *The instruction computes* E *before it places the incremented program counter in AC3.*

Extended Load Accumulator

ELDA *ac,[@]displacement[,index]*

1	0	1	AC	1	INDEX	0	0	1	1	1	0	0	0	@	DISPLACEMENT
0	1	2	3 4	5	6 7	8	9	10	11	12	13	14	15	16	17 31

Moves a copy of the contents of a memory word into bits 16-31 of the specified accumulator.

Calculates the effective address, *E*. Places the contents of the location addressed by *E* in bits 16-31 of the specified accumulator. The contents of the location addressed by *E* remain unchanged.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

Carry remains unchanged and *overflow* is 0.

Extended Load Byte

ELDB *ac,displacement[,index]*

1	0	0	AC	1	INDEX	0	1	1	1	1	0	0	0		DISPLACEMENT
0	1	2	3 4	5	6 7	8	9	10	11	12	13	14	15	16	17 31

Copies a byte from memory into an accumulator.

Forms a byte pointer from the displacement in the following way: shifts the 16-bit number contained in the displacement field to the right one bit, producing a 15-bit

address and a 1-bit byte indicator. Uses the value of the index bits to determine an offset value. Adds the offset value to the 15-bit address produced from the displacement to give a memory address. The byte indicator designates which byte of the addressed word will be loaded into bits 24–31 of the specified accumulator. The instruction sets bits 16–23 of the specified accumulator to 0.

The 32-bit effective address generated by this instruction is constrained to be within the first 64 Kbyte of the current segment.

Carry is unchanged and *overflow* is 0.
The instruction destroys the previous contents of bits 16-31 of the specified accumulator, but it does not alter either the index value or the displacement.

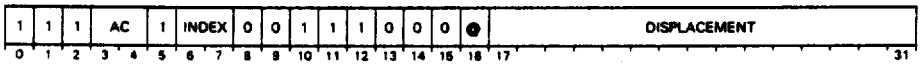
The argument *index* selects the source of the index value. It may have values in the range of 0–3. The meaning of each value is shown below:

Index Bits	Index Value
00	0
01	Address of the displacement field (Word 2 of this instruction)
10	Contents of bits 16-31 of AC2
11	Contents of bits 16-31 of AC3

This instruction sets *overflow* to 0 and carry to 0.

Load Effective Address

ELEF *ac,[@]displacement[,index]*

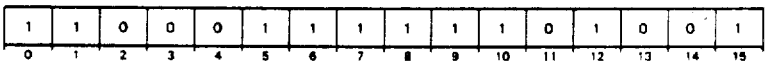


Places a 32-bit effective address constrained to be with the first 32 Kword of the current segment in an accumulator.

Sets bit 0 of the accumulator to 0. *Overflow* is 0 and carry is unchanged.

Enqueue Towards the Head

ENQH



Enqueues a data element.

AC0 contains the effective address of a queue descriptor.

AC1 contains an effective address of a data element in the queue defined by AC0.

AC2 contains the effective address of the data element to be added to the queue.

The instruction checks the page or pages that contain the element for valid read and write access privileges. If the privileges are invalid, the appropriate protection fault occurs and the queue remains unchanged.

If the privileges are valid, the instruction checks the queue descriptor addressed by AC0. If the queue descriptor indicates an empty queue, the instruction ignores the contents of AC1, places the data element addressed by AC2 in the queue, and updates the queue descriptor. The next sequential word is executed.

If the descriptor indicates a queue that contains data elements, the instruction prepares to enqueue a new data element; the instruction enqueues the data element addressed by AC2 *before* the data element addressed by AC1. If the new data element becomes the head of the queue, the instruction updates the queue descriptor appropriately. The next sequential word is skipped.

> NOTE: *If the processor references a page containing a link, the instruction completely updates that link before the processor references another link or descriptor. This means the instruction executes correctly even if only one page is resident in memory.*

The instruction checks all reads and writes of links in data elements and queue descriptors against the current ring. Ring numbers of the link addresses must be greater than or equal to the current ring.

The enqueue operation is not interruptable. The entire operation completes before any interrupts are enabled.

The instruction leaves the contents of AC0, AC1, AC2, and AC3 unchanged. Carry is unchanged and *overflow* is 0.

Enqueue Towards the Tail

ENQT

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

Enqueues a data element.

AC0 contains the effective address of a queue descriptor.

AC1 contains an effective address of a data element in the queue defined by AC0.

AC2 contains the effective address of the data element to be added to the queue.

The instruction checks the page or pages that contain the element for valid read and write access privileges. If the privileges are invalid, the appropriate protection fault occurs and the queue remains unchanged.

If the privileges are valid, the instruction checks the queue descriptor addressed by AC0. If the queue descriptor indicates an empty queue, the instruction ignores the contents of AC1 and enqueues the data element addressed by AC2. The instruction updates the queue descriptor if necessary, then the next sequential word is executed.

If the descriptor indicates a queue that contains data elements, the instruction prepares to enqueue a new data element; the instruction enqueues the data element addressed by AC2 *after* the data element addressed by AC1. If the new data element becomes the tail of the queue, then the instruction updates the queue descriptor appropriately. The next sequential word is skipped.

> NOTE: *If the processor references a page containing a link, the instruction completely updates that link before the processor references another link or descriptor. This means that this instruction will execute correctly even if only one page is resident in memory.*

The instruction checks all reads and writes of links in data elements and queue descriptors against the current ring. Ring numbers of the link addresses must be greater than or equal to the current ring.

The enqueue operation is not interruptable. The entire operation completes before any interrupts are enabled.

The instruction leaves the contents of AC0, AC1, AC2, and AC3 unchanged. Carry is unchanged and *overflow* is 0.

Extended Store Accumulator

ESTA *ac,[@]displacement[,index]*

1	1	0	AC	1	INDEX	0	0	1	1	1	0	0	0	@	DISPLACEMENT
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15 16 17 ... 31

Stores the contents of bits 16-31 of an accumulator into a memory location.

The contents of bits 16-31 of the specified accumulator are placed in the word addressed by the effective address, *E*. The previous contents of the location addressed by *E* are lost. The contents of the specified accumulator and carry remain unchanged.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kwords of the current segment.

Overflow is 0.

Extended Store Byte

ESTB *ac,displacement[,index]*

1	0	1	AC	1	INDEX	0	1	1	1	1	0	0	0		DISPLACEMENT
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15 16 17 ... 31

Copies into memory the byte contained in bits 24-31 of an accumulator.

Forms a byte pointer from the displacement as follows: shifts the 16-bit number contained in the displacement field to the right one bit, producing a 15-bit address and a 1-bit byte indicator. Uses the value of the index bits to determine an offset value. Adds the offset value to the 15-bit address produced from the displacement field to give a memory address. The byte indicator determines which byte of the addressed location will receive bits 24–31 of the specified accumulator.

The 32-bit effective address generated by this instruction is constrained to be within the first 64 Kbyte of the current segment.

The argument *index* selects the source of the index value. It may have values in the range of 0–3; the meaning of each value is shown below:
The instruction checks all reads and writes of links in data elements and queue descriptors against the current ring. Ring numbers of the link addresses must be greater than or equal to the current ring.

The enqueue operation is not interruptable. The entire operation completes before any interrupts are enabled.

The instruction leaves the contents of AC0, AC1, AC2, and AC3 unchanged. Carry is unchanged and *overflow* is 0.

Extended Store Accumulator

ESTA *ac,[@]displacement[,index]*

1	1	0	AC	1	INDEX	0	0	1	1	1	0	0	0	@	DISPLACEMENT
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15 16 17 ... 31

Stores the contents of bits 16-31 of an accumulator into a memory location.

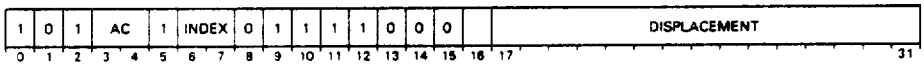
The contents of bits 16-31 of the specified accumulator are placed in the word addressed by the effective address, *E*. The previous contents of the location addressed by *E* are lost. The contents of the specified accumulator and carry remain unchanged.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kwords of the current segment.

Overflow is 0.

Extended Store Byte

ESTB *ac,displacement[,index]*



Copies into memory the byte contained in bits 24-31 of an accumulator.

Forms a byte pointer from the displacement as follows: shifts the 16-bit number contained in the displacement field to the right one bit, producing a 15-bit address and a 1-bit byte indicator. Uses the value of the index bits to determine an offset value. Adds the offset value to the 15-bit address produced from the displacement field to give a memory address. The byte indicator determines which byte of the addressed location will receive bits 24–31 of the specified accumulator.

The 32-bit effective address generated by this instruction is constrained to be within the first 64 Kbyte of the current segment.

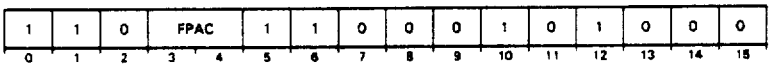
The argument *index* selects the source of the index value. It may have values in the range of 0–3; the meaning of each value is shown below:

Index Bits	Index Value
00	0
01	Address of the displacement field (Word 2 of this instruction)
10	Contents of bits 16-31 of AC2
11	Contents of bits 16-31 of AC3

This instruction leaves carry unchanged; *overflow* is 0.

Absolute Value

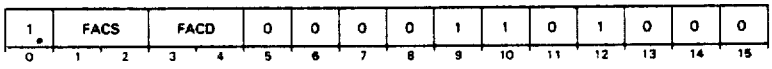
FAB *fpac*



Sets the sign bit of FPAC to 0. Updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

Add Double (FPAC to FPAC)

FAD *facs,facd*



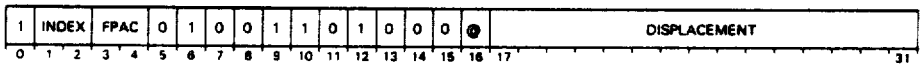
Adds the 64-bit floating point number in FACS to the 64-bit floating point number in FACD.

Adds the 64-bit floating point number in FACS to the 64-bit floating point number in FACD. Places the normalized result in FACD. Leaves the contents of FACS unchanged and updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FACD.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Add Double (Memory to FPAC)

FAMD *fpac,[@]displacement[,index]*



Adds the 64-bit floating point number in the source location to the 64-bit floating point number in FPAC and places the normalized result in FPAC.

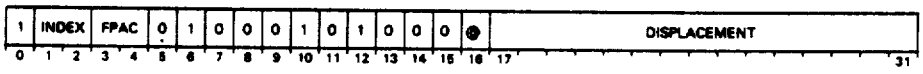
Computes the effective address, *E*. Uses *E* to address a double precision (four word) operand. Adds this 64-bit floating point number to the floating point number in the specified FPAC. Places the normalized result in the specified FPAC. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kwords of the current segment.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Add Single (Memory to FPAC)

FAMS *fpac,[@]displacement[,index]*



Adds the 32-bit floating point number in the source location to the 32-bit floating point number in FPAC and places the normalized result in FPAC.

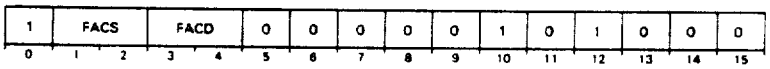
Computes the effective address, *E*. Uses *E* to address a single precision (double word) operand. Adds this 32-bit floating point number to the floating point number in bits 0-31 of the specified FPAC. Places the normalized result in the specified FPAC. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kwords of the current segment.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Add Single (FPAC to FPAC)

FAS *facs,facd*



Adds the 32-bit floating point number in bits 0-31 of FACS to the 32-bit floating point number in bits 0-31 of FACD.

Adds the 32-bit floating point number in bits 0-31 of ACS to the 32-bit floating point number in bits 0-31 of FACD. Places the normalized result in FACD. Leaves the contents of FACS unchanged. Sets bits 32-63 of FACD to 0 and updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FACD.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Clear Errors

FCLE

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

Sets bits 0–4 of the floating point status register to 0.

> NOTES: *Since this instruction sets the* ANY *bit of the FPSR to 0, the FPPC field is undefined.*
>
> *The* IORST *instruction and the system reset function will also set these bits to 0.*

Compare Floating Point

FCMP *facs,facd*

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

Compares two floating point numbers and sets the *Z* and *N* flags in the floating point status register accordingly.

Algebraically compares the floating point numbers in FACS and FACD to each other. Updates the *Z* and *N* flags in the floating point status register to reflect the result. The contents of FACS and FACD remain unchanged. The results of the compare and the corresponding flag settings are shown in the table below.

Z	N	Result
1	0	FACS=FACD
0	1	FACS>FACD
0	0	FACS<FACD

Divide Double (FPAC by FPAC)

FDD *facs,facd*

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

Divides the floating point number in FACD by the floating point number in FACS and places the normalized result in FACD. Destroys the previous contents of FACD. Updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FACD. The contents of FACS remain unchanged.

Checks the floating point number contained in FACS for a zero mantissa. If the mantissa is zero, sets the *DVZ* bit in the floating point status register and terminates. The number in FACD remains unchanged.

If the mantissa is nonzero, compares the mantissas of the numbers contained in FACS and FACD. If the mantissa of the number in FACD is greater than or equal to the mantissa of the number in FACS, the instruction shifts the mantissa of the number in FACD right one hex digit and adds one to the exponent of the number in FACD.

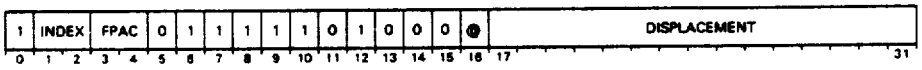
After aligning the mantissas, the instruction divides the mantissa in FACD by the mantissa in FACS. The mantissa of the quotient becomes the mantissa of the intermediate result. The two operands and the rules of algebra determine the sign of the intermediate result. To calculate the exponent of the intermediate result in excess 64 representation, the instruction subtracts the exponent in FACS from the exponent in FACD and adds 64 to this result. The result is already normalized by the shifting algorithm described in the paragraph above, so the instruction places the result in FACD unaltered. Updates the *Z* and *N* flags to reflect the new contents of FACD.

If the exponent processing produces either overflow or underflow, the instruction sets the corresponding bit in the floating point status register. If overflow occurs, the sign and the mantissa in FACD are correct but the exponent is 128 too small. If underflow occurs, the sign and the mantissa in FACD are correct but the exponent is 128 too large.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Divide Double (FPAC by Memory)

FDMD *fpac,[@]displacement[,index]*



Divides the 64-bit floating point number in FPAC by the 64-bit floating point number in the source location and places the normalized result in FPAC.

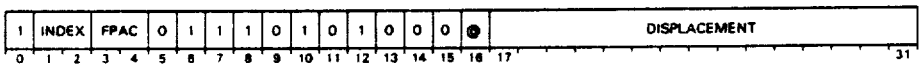
Computes the effective address, *E*. Uses *E* to address a double precision (four word) operand. Divides the floating point number in the specified FPAC by this 64-bit floating point number. Places the normalized result in the specified FPAC. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Divide Single (FPAC by Memory)

FDMS *fpac,[@]displacement[,index]*



Divides the 32-bit floating point number in bits 0-31 of FPAC by the 32-bit floating point number in the source location and places the normalized result in FPAC.

Computes the effective address, *E*. Uses *E* to address a single precision (double word) operand. Divides the floating point number in bits 0-31 of the specified FPAC by this 32-bit floating point number. Places the normalized result in the specified FPAC. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Divide Single (FPAC by FPAC)

FDS *facs,facd*

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

Divides the floating point number in bits 0–31 of FACD by the floating point number in bits 0–31 of FACS and places the normalized result in FACD. Destroys the previous contents of FACD. Updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FACD. The contents of FACS remain unchanged.

Checks the floating point number contained in FACS for a zero mantissa. If the mantissa is zero, sets the *DVZ* bit in the floating point status register and terminates. The number in FACD remains unchanged.

If the mantissa is nonzero, compares the mantissas of the numbers contained in FACS and FACD. If the mantissa of the number in FACD is greater than or equal to the mantissa of the number in FACS, the instruction shifts the mantissa of the number in FACD right one hex digit and adds one to the exponent of the number in FACD.

After aligning the mantissas, the instruction divides the mantissa in FACD by the mantissa in FACS. The mantissa of the quotient becomes the mantissa of the intermediate result. The two operands and the rules of algebra determine the sign of the intermediate result. To calculate the exponent of the intermediate result in excess 64 representation, the instruction subtracts the exponent in FACS from the exponent in FACD and adds 64 to this result. The result is already normalized by the shifting algorithm described in the paragraph above, so the instruction places the result in FACD unaltered. Updates the *Z* and *N* flags to reflect the new contents of FACD.

If the exponent processing produces either overflow or underflow, the instruction sets the corresponding bit in the floating point status register. If overflow occurs, the sign and the mantissa in FACD are correct but the exponent is 128 too small. If underflow occurs, the sign and the mantissa in FACD are correct but the exponent is 128 too large.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Load Exponent

FEXP *fpac*

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

Loads an exponent into bits 1–7 of an accumulator.

The instruction places bits 17–23 of AC0 in bits 1–7 of the specified FPAC. Ignores bits 0–16 and 24–31 of AC0. Changes the *Z* and *N* flags in the floating point status register to reflect the contents of FPAC. AC0 and bits 0 and 8–63 of FPAC remain unchanged. If FPAC contains true zero, the instruction does not load bits 1–7 of FPAC.

Fix To AC

FFAS *ac,fpac*

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

Converts the integer portion of the floating point number contained in the specified FPAC to a signed two's complement integer. Places the result in the specified accumulator.

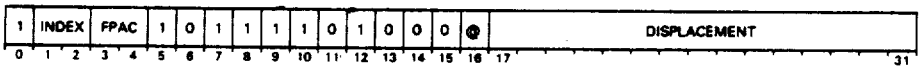
If the integer portion of the number contained in FPAC is less than -32,769 or greater than +32,768, the instruction sets *MOF* in the FPSR to 1. Takes the absolute value of the integer portion of the number contained in the FPAC. Takes the 15 least significant bits of the absolute value and appends a 0 onto the leftmost bit to giver a 16-bit number. If the sign of the number is negative, formes the two's complement of the 16-bit result. Places the 16-bit integer in bits 16-31 of the specified accumulator.

If the integer portion is within the range of -32,768 to +32,767 inclusive, the instruction places the 16-bit, two's complement of the integer portion of the number contained in the FPAC in bits 16-31 of the specified accumulator.

The instruction leaves the FPAC and the *Z* and *N* flags of the FPSR unchanged.

Fix To Memory

FFMD *fpac,[@]displacement[,index]*



Converts the integer portion of the floating point number contained in the specified FPAC to a signed two's complement integer. Places the result in a memory location.

Calculates the effective address, *E*. If the integer portion of the number contained in FPAC is less than -2,147,483,649 or greater than +2,147,483,648, the instruction sets *MOF* in the FPSR to 1. Takes the absolute value of the integer portion of the number contained in the FPAC. Takes the 31 least significant bits of the absolute value and appends a 0 onto the leftmost bit to give a 32-bit number. If the sign of the number is negative, forms the two's complement of the 32-bit result. Places the 32-bit integer in the memory locations specified by *E*.

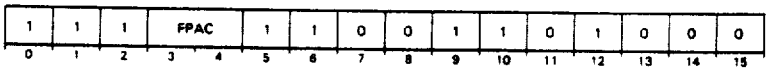
If the integer portion is within the range of -2,147,483,648 to +2,147,483,647 inclusive, the instruction places the 32-bit, two's complement of the integer portion of the number contained in the FPAC in the memory locations specified by *E*.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

The instruction leaves the FPAC and the *Z* and *N* flags of the FPSR unchanged.

Halve

FHLV *fpac*



Divides the floating point number in FPAC by 2.

Shifts the mantissa contained in FPAC right one bit position. Fills the vacated bit position with a zero and places the bit shifted out in the guard digit. Normalizes the number and places the result in FPAC. Updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

If underflow occurs, sets the *UNF* flag in the floating point status register to 1. In this case, the mantissa and sign in FPAC are correct, but the exponent is 128 too large.

This instruction does rounding.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Integerize

FINT

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

Zeros the fractional portion (if any) of the number contained in the specified FPAC. Normalizes the result. Updates the *Z* and *N* flags in the floating point status register to reflect the new contents of the specified FPAC.

NOTE: *If the absolute value of the number contained in the specified FPAC is less than 1, the specified FPAC is set to true zero.*

This instruction truncates towards zero, and does not do rounding.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Float From AC

FLAS *ac,fpac*

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

Converts a two's complement number in the range of -32,768 to +32,767 inclusive to floating point format.

Converts the signed two's complement number contained in bits 16–31 of the specified accumulator to a single precision floating point number. Places the result in the high-order 32 bits of the specified FPAC. Sets the low-order 32 bits of the FPAC to 0. Updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC. The contents of the specified accumulator remain unchanged.

Load Floating Point Double

FLDD *fpac,[@]displacement[,index]*

1	INDEX	FPAC	1	0	0	0	1	1	0	1	0	0	0	@	DISPLACEMENT
0	1 2	3 4	5	6	7	8	9	10	11	12	13	14	15	16	17 ... 31

Moves four words out of memory and into a specified FPAC.

Computes the effective address, *E*. Fetches the double precision floating point number at the address specified by *E* and places it in FPAC. Updates the *Z* and *N* flags in the FPSR to reflect the new contents of FPAC.

The 32-bit effective address generated by this instruction is contrained to be within the first 32 Kword of the current segment.

NOTE: *This instruction will move unnormalized data without change.*

Load Floating Point Single

FLDS *fpac,[@]displacement[,index]*

1	INDEX	FPAC	1	0	0	0	0	1	0	1	0	0	0	@	DISPLACEMENT
0	1 2	3 4	5	6	7	8	9	10	11	12	13	14	15	16	17 ... 31

Moves two words out of memory into a specified FPAC.

Computes the effective address, *E*. Fetches the single precision floating point number at the address specified by *E*. Places the number in the high-order bits of FPAC. Sets the low-order 32 bits of FPAC to 0. Updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

NOTE: *This instruction will move unnormalized or illegal data without change.*

Float From Memory

FLMD *fpac,[@]displacement[,index]*

1	INDEX	FPAC	1	0	1	0	1	1	0	1	0	0	0	@	DISPLACEMENT
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15 16 17 ... 31

Converts the contents of two 16-bit memory locations to floating point format and places the result in a specified FPAC.

Computes the effective address, *E*. Converts the 32-bit, signed, two's complement number addressed by *E* to a double precision floating point number. Places the result in the specified FPAC. Updates the *Z* and *N* flags in the floating point status register to reflect the new contents of the FPAC.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.
The range of numbers that you can convert is -2,147,483,648 to +2,147,483,647 inclusive.

Load Floating Point Status

FLST *[@]displacement[,index]*

1	0	1	INDEX	1	1	0	1	1	1	0	1	0	0	0	@	DISPLACEMENT
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16 17 ... 31

Moves two words out of memory into the floating point status register.

Computes the effective address, *E*. Places the 32-bit operand addressed by *E* in the floating point status register as follows:

- Places bits 0-15 of the operand in bits 0-15 of the FPSR. Sets bits 16-32 of the FPSR to 0.
- If *ANY* is 0, bits 33-63 of the FPSR (the FPPC) are undefined.
- If *ANY* is 1, the instruction places the value of the current segment in bits 33-35 of the FPSR, zeroes in bits 36-48, and bits 17-31 of the operand in bits 49-63 of the FPSR.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

NOTES: *This instruction does not set the* ANY *flag from memory. If any of bits 1-4 are loaded as 1,* ANY *is set to 1; otherwise,* ANY *is 0.*

Bits 12-15 of the FPSR are not set from memory. These bits are the floating point identification code an are read protected. In the MV/8000 they are set to 0111.

This instruction initiates a floating point trap if ANY *and* TE *are both 1 after the FPPC is loaded.*

See Chapter 8 and Appendix G for more information about floating point manipulation.

Multiply Double (FPAC by FPAC)

FMD *facs,facd*

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

Multiplies the floating point number in FACD by the floating point number in FACS and places the normalized result in FACD. Updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FACD. The contents of FACS remain unchanged.

The instruction multiplies the mantissas of the two numbers together. The result is the mantissa of the intermediate result. The two operands and the rules of algebra determine the sign of the intermediate result. Adds the exponents of the two numbers together and subtracts 64 from the result to maintain excess 64 notation. This value becomes the exponent of the intermediate result. Normalizes the intermediate result if necessary and loads the result into FACD. Updates the *Z* and *N* flags in the floating point status register.
If the exponent processing produces either overflow or underflow, the result is held until normalization, as that procedure may correct the condition. If normalization does not correct the condition, the instruction sets the corresponding flag in the floating point status register to 1. The mantissa and sign of the number will be correct, but the exponent will be 128 too small if overflow occurred, or 128 too large if underflow occurred.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Multiply Double (FPAC by Memory)

FMMD *fpac,[@]displacement[,index]*

1	INDEX	FPAC	0	1	1	0	1	1	0	1	0	0	0	@	DISPLACEMENT
0	1 2	3 4	5	6	7	8	9	10	11	12	13	14	15	16	17 31

Multiplies the 64-bit floating point number in the source location by the 64-bit floating point number in FPAC and places the normalized result in FPAC.

Computes the effective address, *E*. Uses *E* to address a double precision (four word) operand. Multiplies this 64-bit floating point number by the floating point number in the specified FPAC. Places the normalized result in the specified FPAC. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

The 32-bit effective address generated by this instruction is constrained to be within the first 64 Kword of the current segment.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Multiply Single (FPAC by Memory)

FMMS *fpac,[@]displacement[,index]*

1	INDEX	FPAC	0	1	1	0	0	1	0	1	0	0	0	@	DISPLACEMENT
0	1 2	3 4	5	6	7	8	9	10	11	12	13	14	15	16	17 31

Multiplies the 32-bit floating point number in the source location by the 32-bit floating point number in bits 0-31 of FPAC and places the normalized result in FPAC.

Computes the effective address, *E*. Uses *E* to address a single precision (double word) operand. Multiplies this 32-bit floating point number by the floating point number in bits 0-31 of the specified FPAC. Places the normalized result in bits 0-31 of the specified FPAC. Sets bits 32-63 of FPAC to 0. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Move Floating Point

FMOV *facs,facd*

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

Moves the contents of one FPAC to another FPAC.

Places the contents of FACS in FACD. Updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FACD. The contents of FACS remain unchanged.

Multiply Single (FPAC by FPAC)

FMS *facs,facd*

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

Multiplies the 32-bit floating point number in bits 0-31 of FACS by the 32-bit floating point number in bits 0-31 of FACD.

Multiplies the 32-bit floating point number in bits 0-31 of ACS by the 32-bit floating point number in bits 0-31 of FACD. Places the normalized result in FACD. Leaves the contents of FACS unchanged. Sets bits 32-63 of FACD to 0 and updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FACD.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Negate

FNEG *fpac*

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

Inverts the sign bit of FPAC. Leaves bits 1–63 of FPAC unchanged. Updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

If FPAC contains true zero, leaves the sign bit unchanged.

Normalize

FNOM *fpac*

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

Normalizes the floating point numbers in FPAC. Sets a true zero in FPAC if all the bits of the mantissa are zero. Updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

If an exponent underflow occurs, sets the *UNF* flag in the floating point status register. In this case, the mantissa and the sign of the number in FPAC are correct, but the exponent is 128 too large.

NOTE: *This instruction does not do rounding.*

See Chapter 8 and Appendix G for more information about floating point manipulation.

No Skip

FNS

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

The next sequential word is executed.

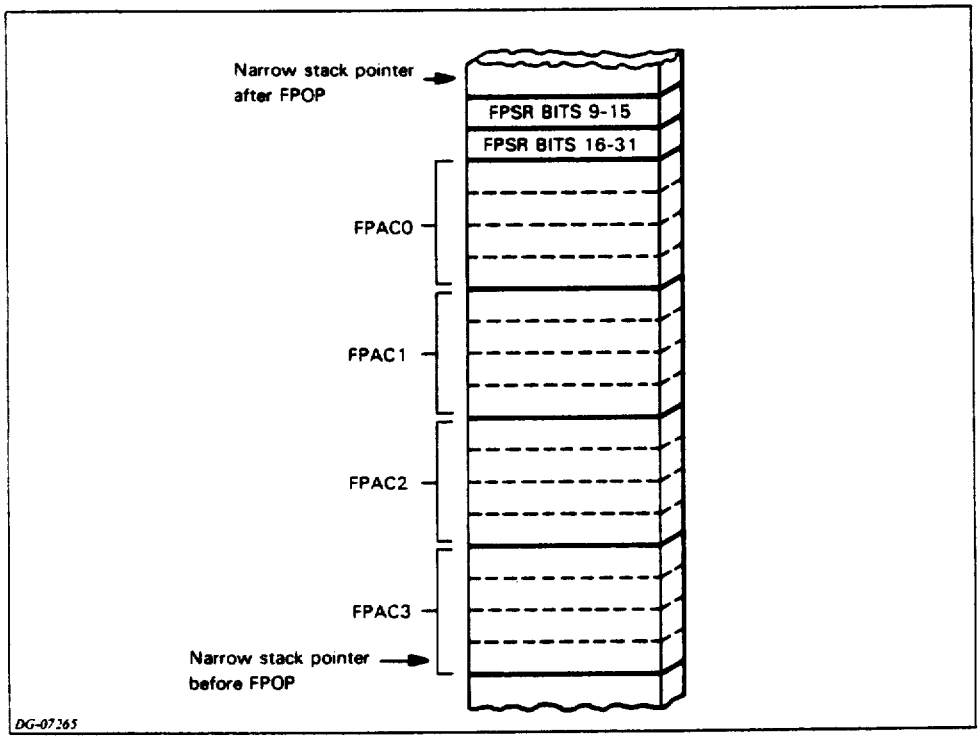
Pop Floating Point State

FPOP

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

Pops the state of the floating point unit off the narrow stack.

Pops an 18-word block off the narrow stack and loads the contents into the FPSR and the four FPACs. The format of the 18-word block is shown below.



The instruction pops the first 32-bit operand on the stack and places it in the FPSR as follows:

- Places bits 0-15 of the operand in bits 0-15 of the FPSR.
- If *ANY* is 0, bits 16-31 of the FPSR are undefined.
- If *ANY* is 1, the instruction places bits 16-31 of the popped operand into bits 16-31 of the FPSR.

The rest of the stack words are popped in the usual way. See Chapter 8 for more information.

The 32-bit effective address generated by this instruction is constrainted to be within the first 32 Kword of the current segment.

NOTES: *This instruction moves unnormalized data without change.*

This instruction does not set the ANY *flag from memory. If any of bits 1-4 are loaded as 1,* ANY *is set to 1; otherwise,* ANY *is 0.*

Bits 12-15 of the FPSR are not set from memory. These bits are the floating point identification code an are read protected. In the EAGLE they are set to 0111.

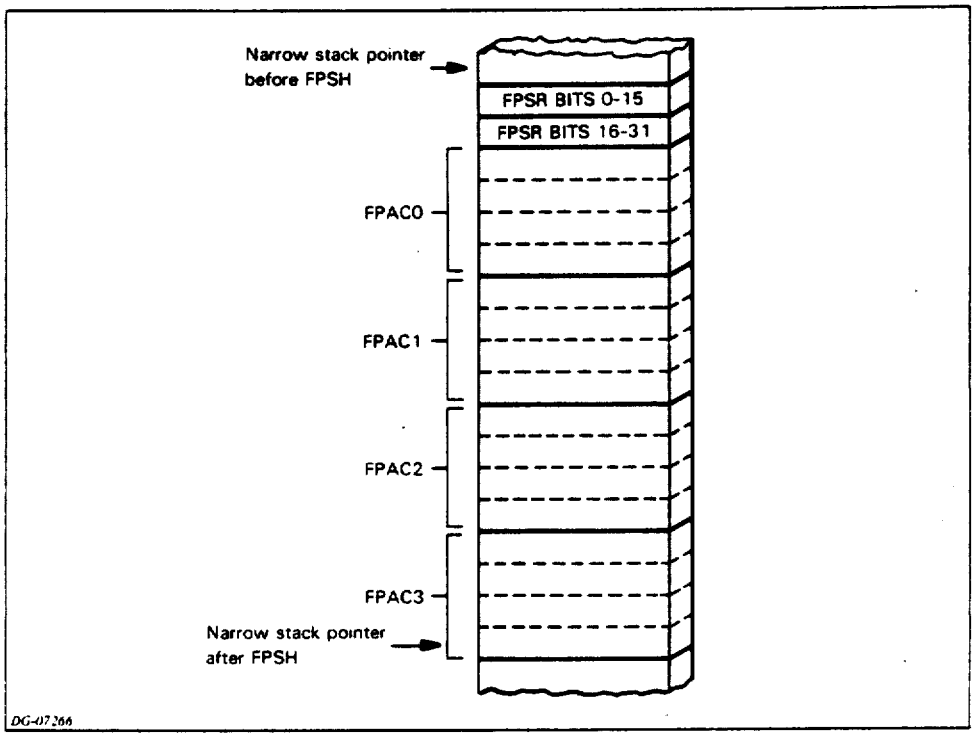
This instruction does not initiates a floating point trap under any conditions of the FPSR.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Push Floating Point State
FPSH

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

Pushes an 18-word floating point return block onto the narrow stack, leaving the contents of the floating point accumulators and the floating point status register unchanged. The format of the 18 words pushed is as follows:



The instruction pushes the contents of the FPSR as follows:

- Stores bits 0-15 of the FPSR in the first memory word.
- If *ANY* is 0, the contents of the second memory word are undefined.
- If *ANY* is 1, the instruction stores bits 16-31 of the FPSR into the second memory word.

The rest of the block is pushed after the FPSR has been pushed.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

NOTES: *This instruction moves unnormalized data without change.*

This instruction does not initiate a floating point trap under any conditions of the FPSR.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Read High Word

FRH *fpac*

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

Places the high-order 16 bits of FPAC in bits 16–31 of AC0. FPAC and the *Z* and *N* flags in the floating point status register remain unchanged.

NOTE: *This instruction moves unnormalized data without change.*

Skip Always

FSA

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

Skips the next sequential word.

Scale

FSCAL *fpac*

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

Shifts the mantissa of the floating point number in FPAC either right or left, depending upon the contents of bits 17–23 of AC0. Leaves the contents of AC0 unchanged.

Bits 17–23 of AC0 contain an exponent.

The instruction subtracts the exponent of the number contained in FPAC from the exponent in AC0. The difference between the exponents specifies *D*, a number of hex digits.

If *D* is zero, the instruction updates the *Z* and *N* flags, and stops.

If *D* is positive, the instruction shifts the mantissa of the number contained in FPAC to the right by *D* digits.

If *D* is negative, the instruction shifts the mantissa of the number contained in FPAC to the left by *D* digits. Sets the *MOF* flag in the floating point status register.

After the right or left shift, the instruction loads the contents of bits 17–23 of AC0 into the exponent field of FPAC. Bits shifted out of either end of the mantissa are lost. Updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

NOTE: *This instruction does not do rounding.*

See Chapter 8 and Appendix G for more information about floating point manipulation.

Subtract Double (FPAC from FPAC)

FSD *facs,facd*

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

Subtracts the 64-bit floating point number in FACS from the 64-bit floating point number in FACD. Places the normalized result in FACD. Updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FACD. The contents of FACS remain unchanged.

Refer to Chapter 8 and Appendix G for more information about floating point manipulation.

Skip On Zero

FSEQ

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

Skips the next sequential word if the *Z* flag of the floating point status register is 1.

Skip On Greater Than Or Equal To Zero

FSGE

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

Skips the next sequential word if the *N* flag of the floating point status register is 0.

Skip On Greater Than Zero

FSGT

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

Skips the next sequential word if both the *Z* and *N* flags of the floating point status register are 0.

Skip On Less Than Or Equal To Zero

FSLE

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

Skips the next sequential instruction if either the *Z* flag or the *N* flag of the floating point status register is 1.

Skip On Less Than Zero

FSLT

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

Skips the next sequential word if the *N* flag of the floating point status register is 1.

Subtract Double (Memory from FPAC)

FSMD *fpac,[@]displacement[,index]*

1	INDEX	FPAC	0	1	0	1	1	1	0	1	0	0	0	●	DISPLACEMENT
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17 ... 31

Subtracts the 64-bit floating point number in the source location from the 64-bit floating point number in FPAC and places the normalized result in FPAC.

Computes the effective address, *E*. Uses *E* to address a double precision (four word) operand. Subtracts this 64-bit floating point number from the floating point number in the specified FPAC. Places the normalized result in the specified FPAC. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Subtract Single (Memory from FPAC)

FSMS *fpac,[@]displacement[,index]*

1	INDEX	FPAC	0	1	0	1	0	1	0	1	0	0	0	●	DISPLACEMENT
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17 ... 31

Subtracts the 32-bit floating point number in the source location from the 32-bit floating point number in bits 0-31 of FPAC and places the normalized result in FPAC.

Computes the effective address, *E*. Uses *E* to address a single precision (double word) operand. Subtracts this 32-bit floating point number from the floating point number in bits 0-31 of the specified FPAC. Places the normalized result in the specified FPAC. Sets bits 32-63 of FPAC to 0. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.
See Chapter 8 and Appendix G for more information about floating point manipulation.

Skip On No Zero Divide

FSND

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

Skips the next sequential word if the *DVZ* flag of the floating point status register is 0.

Skip On Non-Zero

FSNE

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

Skips the next sequential word if the *Z* flag of the floating point status register is 0.

Skip On No Error

FSNER

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

Skips the next sequential word if bits 1–4 of the floating point status register are all 0.

Skip On No Mantissa Overflow

FSNM

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

Skips the next sequential word if the *MOF* flag of the floating point status register is 0.

Skip On No Overflow

FSNO

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

Skips the next sequential word if the *OVF* flag of the floating point status register is 0.

Skip On No Overflow and No Zero Divide

FSNOD

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

Skips the next sequential word if both the *OVF* flag and the *DVZ* flag of the floating point status register are 0.

Skip On No Underflow

FSNU

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

Skips the next sequential word if the *UNF* flag of the floating point status register is 0.

Skip On No Underflow And No Zero Divide

FSNUD

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

Skips the next sequential word if both the *UNF* flag and the *DVZ* flag of the floating point status register are 0.

Skip On No Underflow And No Overflow

FSNUO

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

Skips the next sequential word if both the *UNF* flag and the *OVF* flag of the floating point status register are 0.

Subtract Single (FPAC from FPAC)

FSS *facs facd*

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

Subtracts the 32-bit floating point number in bits 0-31 of FACS from the 32-bit floating point number in bits 0-31 of FACD. Places the normalized result in bits 0-31 of FACD. Sets bits 32-63 of FACD to 0. Updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FACD. The contents of FACS remain unchanged. Refer to Chapter 8 and Appendix G for more information about floating point manipulation.

Store Floating Point Status

FSST *[@]displacement[,index]*

1	0	0	INDEX		1	1	0	1	1	1	0	1	0	0	0	@	DISPLACEMENT
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17 ... 31

Moves the contents of the narrow FPSR into memory.

Computes the effective address, *E*, of two sequential, 16-bit locations in memory. Stores the contents of the FPSR in these locations as follows:

- Stores bits 0-15 of the FPSR in the first memory word.
- If *ANY* is 0, the contents of the second memory word are undefined.
- If *ANY* is 1, the instruction stores bits 48-63 of the FPSR into the second memory word.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

NOTE: *This instruction does not initiate a floating point trap under any conditions of the FPSR.*

• See Chapter 8 and Appendix G for more information about floating point manipulation.

Store Floating Point Double

FSTD *fpac,[@]displacement[,index]*

1	INDEX	FPAC	1	0	0	1	1	1	0	1	0	0	0	@	DISPLACEMENT
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15 16 17 ... 31

Stores the contents of a specified FPAC into a memory location.

Computes the effective address, *E*. Places the floating point number contained in FPAC in memory beginning at the location addressed by *E*. Destroys the previous contents of the addressed memory location. The contents of FPAC and the condition codes in the FPSR remain unchanged.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

Store Floating Point Single

FSTS *fpac,[@]displacement[,index]*

1	INDEX	FPAC	1	0	0	1	0	1	0	1	0	0	0	@	DISPLACEMENT
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15 16 17 ... 31

Stores the contents of a specified FPAC into a memory location.
Computes the effective address *E*. Places the 32 high-order bits of FPAC in memory beginning at the location addressed by *E*. Destroys the previous contents of the addressed memory location. The contents of FPAC and the condition codes in the FPSR remain unchanged.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

Trap Disable

FTD

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

Sets the trap enable (*TE*) bit of the FPSR to 0.

NOTE: *The I/O RESET instruction will also set this bit to 0.*

Trap Enable

FTE

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

Sets the trap enable *TE* bit of the FPSR to 1. If *ANY* is 1 before execution of this instruction, signals a floating point trap. If *ANY* is 0 before execution of this instruction, execution continues normally at the end of this instruction.

NOTES: *When this instruction is used to cause a floating point trap, the FPPC portion of the FPSR will contain the address of the first instruction to cause a fault. Even if another instruction causes a second fault that occurs before the* FTE *instruction executes, the FPPC will still contain the address of the first instruction that caused a fault.*

When a floating point fault occurs and TE *is 1, the processor sets* TE *to 0 before transferring control to the floating point error handler.* TE *should be set to 1 before resuming normal processing.*

Fixed Point Trap Disable

FXTD

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

Unconditionally sets the **OVK** flag to zero. This disables fixed point overflow traps. Carry is unchanged.

Fixed Point Trap Enable

FXTE

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

Unconditionally sets **OVK** to 1 and **OVR** to 0. This enables fixed point overflow traps. Carry is unchanged.

Halve

HLV *ac*

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

Divides the contents of an accumulator by 2 and rounds the result toward zero.

The signed, 16-bit two's complement number contained in bits 16-31 of the specified accumulator is divided by 2 and rounded toward 0. The result is placed in bits 16-31 of the specified accumulator.

Bits 0-15 of the modified accumulator are undefined after completion of this instruction.

This instruction leaves carry unchanged; *overflow* is 0.

Hex Shift Left

HXL *n,ac*

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

Shifts the contents of bits 16-31 of the specified accumulator left a number of hex digits depending upon the immediate field N. The number of digits shifted is equal to $N+1$. Bits shifted out are lost, and the vacated bit positions are filled with zeroes. If N is equal to 3, then bits 16-31 of the specified accumulator are shifted out and are set to 0. Leaves carry unchanged. *Overflow* is 0.

Bits 0-15 of the modified accumulator are undefined after completion of this instruction.

> NOTE: *The assembler takes the coded value of* n *and subtracts one from it before placing it in the immediate field. Therefore, the programmer should code the exact number of hex digits that he wishes to shift.*

Hex Shift Right

HXR *n,ac*

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

Shifts the contents of bits 16-31 of the specified accumulator right a number of hex

digits depending upon the immediate field, N. The number of digits shifted is equal to $N+1$. Bits shifted out are lost, and the vacated bit positions are filled with zeroes. If N is equal to 3, then bits 16-31 of the specified accumulator are shifted out and are set to 0. Leaves carry unchanged. *Overflow* is 0.

Bits 0-15 of the modified accumulator are undefined after completion of this instruction.

NOTE: *The assembler takes the coded value of* n *and subtracts one from it before placing it in the immediate field. Therefore, the programmer should code the exact number of hex digits that he wishes to shift.*

Increment

INC*[c][sh][#]* *acs,acd[,skip]*

1	ACS	ACD	0	1	1	SH	C	#	SKIP
0	1 2	3 4	5	6	7	8 9	10 11	12	13 15

Increments the contents of bits 16-31 of an accumulator.

Initializes carry to the specified value. Increments the unsigned, 16-bit number in bits 16-31 of ACS by one and places the result in the shifter. If the incrementation produces a carry of 1 out of the high order bit, the instruction complements carry. Performs the specified shift operation, and loads the result of the shift into bits 16-31 of ACD if the no-load bit is 0. If the skip condition is true, the next sequential word is skipped.

If the load option is specified, bits 0-15 of ACD are undefined.

NOTE: *If the number in ACS is* 177777_8 *the instruction complements carry.*

For this instruction, *overflow* is 0.

Inclusive OR

IOR *acs,acd*

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

Forms the logical inclusive OR of the contents of bits 16-31 of ACS and the contents of bits 16-31 of ACD and places the result in bits 16-31 of ACD. Sets a bit position in the result to 1 if the corresponding bit position in one or both operands contains a 1; otherwise, the instruction sets the result bit to 0. The contents of ACS remain unchanged. Carry remains unchanged. *Overflow* is 0.

Bits 0-15 of the modified accumulator are undefined after completion of this instruction.

Inclusive OR Immediate

IORI *i,ac*

1	0	0	AC	1	1	1	1	1	1	1	1	0	0	0	IMMEDIATE FIELD
0	1	2	3 4	5	6	7	8	9	10	11	12	13	14	15	16 31

Forms the logical inclusive OR of the contents of the immediate field and the contents of bits 16-31 of the specified accumulator and places the result in bits 16-31 of the specified accumulator. Carry remains unchanged and *overflow* is 0.

Bits 0-15 of the modified accumulator are undefined after completion of this instruction.

Increment And Skip If Zero

ISZ *[@]displacement[,index]*

0	0	0	1	0	@	INDEX	DISPLACEMENT
0	1	2	3	4	5	6 7	8 15

Increments the addressed word, then skips if the incremented value is zero.

Increments the word addressed by *E* and writes the result back into memory at that location. If the updated value of the location is zero, the instruction skips the next sequential word.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

Carry remains unchanged and *overflow* is 0.

Increment the Word Addressed by WSP and Skip if Zero

ISZTS

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

Uses the contents of WSP (the wide stack pointer) as the address of a double word. Increments the contents of the word addressed by WSP. If the incremented value is equal to zero, then the next sequential word is skipped. Carry is unchanged and *overflow* is 0.

NOTE: *The operation performed by this instruction is not indivisible.*

Jump

JMP

0	0	0	0	0	@	INDEX	DISPLACEMENT
0	1	2	3	4	5	6 7	8 15

Computes the effective address, *E*, and places it in the program counter. Sequential operation continues with the word addressed by the updated value of the program counter.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

Carry remains unchanged and *overflow* is 0.

Jump To Subroutine

JSR *[@]displacement[,index]*

0	0	0	0	1	@	INDEX	DISPLACEMENT
0	1	2	3	4	5	6 7	8 15

Increments and stores the value of the program counter in AC3, and then places a new address in the program counter.

Computes the effective address, *E*; then places the address of the next sequential instruction in bits 16-31 of AC3. Places *E* in the program counter. Sequential operation continues with the word addressed by the updated value of the program counter.

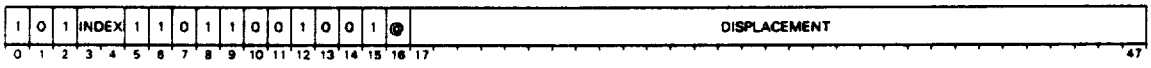
The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

Carry remains unchanged and *overflow* is 0.

NOTE: *The instruction computes* E *before it places the incremented program counter in AC3.*

Call Subroutine (Long Displacement)

LCALL *opcode,argument count,displacement*

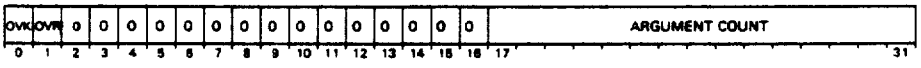


Evaluates the address of a subroutine call.

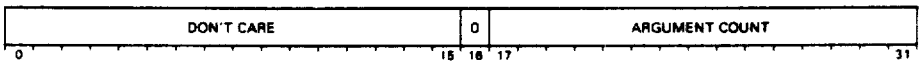
If the target address specifies an outward ring crossing, a protection fault (code=7 in AC1) occurs. Note that the contents of the PC in the return block are undefined.

If the target address specifies an inward ring call, then the instruction assumes the target address has the following format:

The instruction checks the gate field of the above format for a legal gate. If the specified gate is illegal, a protection fault (code=6 in AC1) occurs and no subroutine call is made. Note that the value of the PC in the return block is undefined.
If the specified gate is legal, or if the target address specifies an intra ring crossing, the instruction loads the contents of the PC, plus four, into AC3. The contents of AC3 always references the current ring. If bit 0 of the argument count is 0, the instruction creates a word with the following format:



The instruction pushes this word onto the wide stack. If a stack overflow occurs after this push, a stack fault occurs and no subroutine call is made. Note that the value of the PC in the return block is undefined. If bit 0 of the argument count is 1, then the instruction assumes the top word of the wide stack has the following format:



The instruction modifies this word to include the correct settings of **OVK** and **OVR** in bits 0 and 1.

Regardless of the setting of bit 0 of the argument count, the instruction next unconditionally sets **OVR** to 0 and loads the PC with the target address. Control then transfers to the word referenced by the PC.

Load CPU Identification

LCPID

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

Loads a double word into AC0. Carry is unchanged and *overflow* is 0.

The double word has the format:

MODEL NUMBER	MICROCODE REV	0	0	MEM SIZE
0 … 15	16 … 23	24	25	26 … 31

where

model # is the binary representation of the machine's model number,
microcode rev indicates the microcode revision currently in use on this machine,
mem size indicates the amount of physical memory on this machine. A zero in this field indicates 256 Kbytes of memory; a one indicates 512 Kbytes, and so on.

Load Accumulator

LDA *ac,[@]displacement[,index]*

0	0	1	AC	@	INDEX	DISPLACEMENT
0	1	2	3 … 4	5	6 … 7	8 … 15

Copies a word from memory to an accumulator.
Places the word addressed by the effective address, *E*, in bits 16-31 of the specified accumulator.

Bits 0-15 are undefined.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

The previous contents of the location addressed by *E* and carry remain unchanged. *Overflow* is 0.

Load Accumulator with WFP

LDAFP *ac*

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

Loads the 32-bit contents of WFP (the wide frame pointer) into the specified 32-bit accumulator. Carry is unchanged and *overflow* is 0.

Load Accumulator with WSB

LDASB *ac*

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

Loads the 32-bit contents of WSB (the wide stack base) into the specified 32-bit accumulator. Carry is unchanged and *overflow* is 0.

Load Accumulator with WSL

LDASL *ac*

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

Loads the 32-bit contents of WSL (the wide stack limit) into the specified 32-bit accumulator. Carry is unchanged and *overflow* is 0.

Load Accumulator with WSP

LDASP *ac*

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

Loads the contents of WSP (the wide stack pointer) into the specified accumulator. Carry is unchanged and *overflow* is 0.

Load Accumulator with Double Word

LDATS *ac*

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

Uses the contents of WSP (the wide stack pointer) as the address of a double word. Loads the contents of the addressed double word into the specified accumulator. Carry is unchanged and *overflow* is 0.

Load Byte

LDB *acs,acd*

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

Moves a copy of the contents of a memory byte (as addressed by a byte pointer in one accumulator) into the second accumulator.

Places the 8-bit byte addressed by the byte pointer contained in bits 15-31 of ACS in bits 24–31 of ACD. Sets bits 16–23 of ACD to 0. The contents of ACS remain unchanged unless ACS and ACD are the same accumulator. Carry remains unchanged and *overflow* is 0.

The 32-bit effective address generated by this instruction is constrained to be within the first 64 Kbyte of the current segment.

Load Integer

LDI *fpac*

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

Translates a decimal integer from memory to (normalized) floating point format and places the result in a floating point accumulator.

Under the control of accumulators AC1 and AC3, converts a decimal integer to floating point form, normalizes it, and places it in the specified FPAC. The instruction updates the *Z* and *N* bits in the FPSR to describe the new contents of the specified FPAC. Leaves the decimal number unchanged in memory, and destroys the previous contents of the specified FPAC.

Bits 16-31 of AC1 must contain the data-type indicator describing the number.

Bits 16-31 of AC3 must contain a byte pointer which is the address of the high-order byte of the number in memory.

Numbers of data type 7 are not normalized after loading. By convention, the first byte of a number stored according to data type 7 must contain the sign and exponent of the floating point number. The exponent must be in "excess 64" representation. The instruction copies each byte (following the lead byte) directly to mantissa of the specified FPAC. It then sets to zero each low-order byte in the FPAC that does not receive data from memory.

Upon successful completion, the instruction leaves accumulators AC0 and AC1 unchanged. AC2 contains the original contents of AC3; the contents of AC3 are undefined. Carry remains unchanged and *overflow* is 0.

The 32-bit effective address generated by this instruction is constrained to be within the first 64 Kbyte of the current segment.

NOTE: *An attempt to load a minus 0 sets the specified FPAC to true zero.*

Load Integer Extended
LDIX

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

Distributes a decimal integer of data type 0, 1, 2, 3, 4, or 5 into the four FPACs.

Extends the integer with high-order zeros until it is 32 digits long. Divides the integer into four units of 8 digits each and converts each unit to a floating point number. Places the number obtained from the 8 high-order digits into FAC0, the number obtained from the next 8 digits into FAC1, the number obtained from the next 8 digits into FAC2, and the number obtained from the low-order 8 bits into FAC3. The instruction places the sign of the integer in each FPAC unless that FPAC has received 8 digits of zeros, in which case the instruction sets FPAC to true zero. The *Z* and *N* flags in the floating point status register are unpredictable.

Bits 16-31 of AC1 must contain the data-type indicator describing the integer.

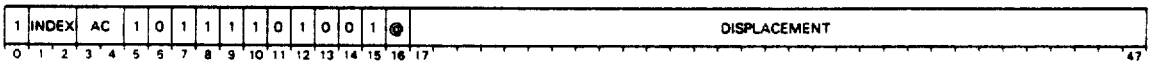
Bits 16-31 of AC3 must contain a byte pointer which is the address of the high-order byte of the integer.

Upon successful termination, the contents of AC0 and AC3 are undefined; the contents of AC1 remain unchanged; and AC2 contains the original contents of AC3. Carry remains unchanged and *overflow* is 0.

The 32-bit effective address generated by this instruction is constrained to be within the first 64 Kbyte of the current segment.

Dispatch (Long Displacement)

LDSP *ac,index,displacement*



Dispatches through a table of 28-bit self-relative addresses indexed by the 31-bit PC.

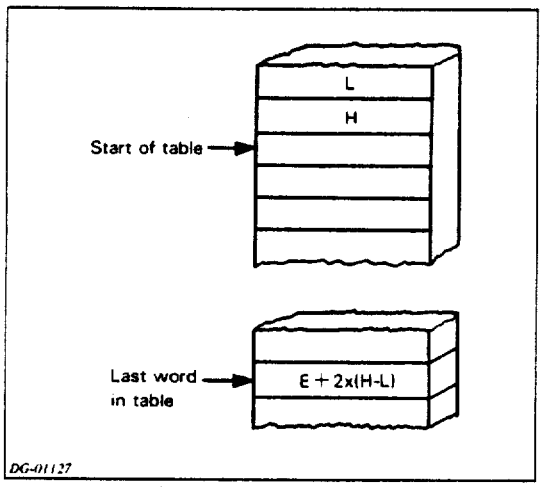
Computes the effective address *E* . This is the address of a *dispatch table*. The dispatch table consists of a table of 28-bit self-relative addresses (bits 0-3 are ignored). Immediately before the table are two signed, two's complement limit words, *L* and *H*. The last word of the table is in location $E + 2 \times (H - L)$. The instruction adds the 28-bit self-relative address in the table entry to the address of the table entry. The ring field of the fetched table entry is ignored.

Compares the signed, two's complement number contained in the accumulator to the limit words. If the number in the accumulator is less than *L* or greater than *H*, sequential operation continues with the instruction immediately after the *Wide Dispatch* instruction.

If the number in AC is greater than or equal to *L* and less than or equal to *H*, the instruction fetches the word at location $E - 2 \times (L - number)$. If the fetched word is equal to 37777777777_8 (all 1's), sequential operation continues with the instruction immediately after the *Wide Dispatch* instruction. If the fetched word is not equal to 37777777777_8, the instruction treats this word as the intermediate address in the effective address calculation. After the indirection chain, if any, has been followed, the instruction places the effective address in the program counter and sequential operation continues with the word addressed by the updated value of the program counter. Carry is unchanged and *overflow* is 0.

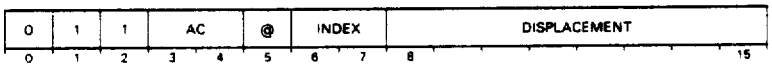
Wraparound occurs within the 28-bit offset. A ring crossing cannot occur. The effective address, *E*, references a table of self-relative addresses in the current segment. Thus, bits 1–3 of *E* and bits 1–3 of any levels of indirection are always interpreted as the current segment.

The structure of the dispatch table is shown in the figure below.



Load Effective Address

LEF *ac.[@]displacement[,index]*



Computes the effective address, *E*, within the current segment and places it in the specified accumulator. Sets bit 0 of the accumulator to 0. The previous contents of the AC are lost.

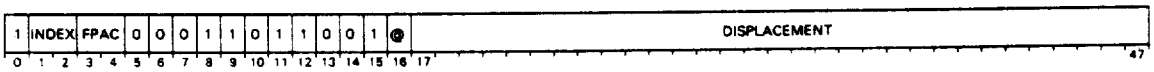
The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

> **NOTES:** *The* LEF *instruction can only be used in a mapped system while in the user mode. With the* LEF *mode bit set to 1, all I/O and* LEF *instructions will be interpreted as* LEF *instructions. With the* LEF *mode bit set to 0, all I/O and* LEF *instructions will be interpreted as I/O instructions.*
>
> *Be sure that I/O protection is enabled or the* LEF *mode bit is set to 1 before using the* LEF *instruction. If you issue a* LEF *instruction in the I/O mode, with protection disabled, the instruction will be interpreted and executed as an I/O instruction, with possibly undesirable results.*

Carry is unchanged and *overflow* is 0.

Add Double (Memory to FPAC) (Long Displacement)

LFAMD *fpac.[@]displacement[,index]*



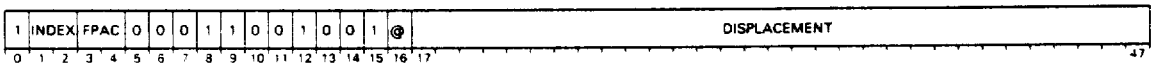
Adds the 64-bit floating point number in the source location to the 64-bit floating poi number in FPAC and places the normalized result in FPAC.

Computes the effective address, *E*. Uses *E* to address a double precision (four word) operand. Adds this 64-bit floating point number to the floating point number in the specified FPAC. Places the normalized result in the specified FPAC. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Add Single (Memory to FPAC) (Long Displacement)

LFAMS *fpac.[@]displacement[,index]*



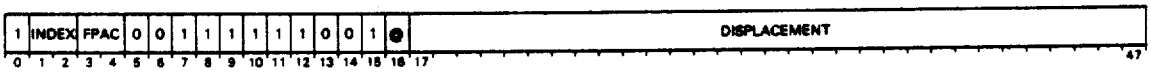
Adds the 32-bit floating point number in the source location to the 32-bit floating point number in FPAC and places the normalized result in FPAC.

Computes the effective address, *E*. Uses *E* to address a single precision (double word) operand. Adds this 32-bit floating point number to the floating point number in bits 0-31 of the specified FPAC. Places the normalized result in the specified FPAC. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Divide Double (FPAC by Memory) (Long Displacement)

LFDMD *fpac,[@]displacement[,index]*



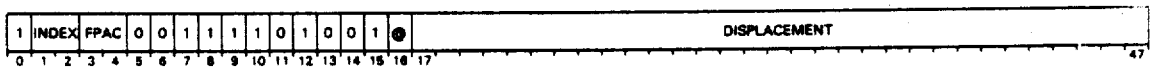
Divides the 64-bit floating point number in FPAC by the 64-bit floating point number in the source location and places the normalized result in FPAC.

Computes the effective address, *E*. Uses *E* to address a double precision (four word) operand. Divides the floating point number in the specified FPAC by this 64-bit floating point number. Places the normalized result in the specified FPAC. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Divide Single (FPAC by Memory) (Long Displacement)

LFDMS *fpac,[@]displacement[,index]*



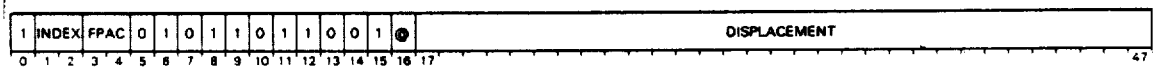
Divides the 32-bit floating point number in bits 32-63 of FPAC by the 32-bit floating point number in the source location and places the normalized result in FPAC.

Computes the effective address, *E*. Uses *E* to address a single precision (double word) operand. Divides the floating point number in bits 0-31 of the specified FPAC by this 32-bit floating point number. Places the normalized result in the specified FPAC. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Load Floating Point Double (Long Displacement)

LFLDD *fpac,[@]displacement[,index]*



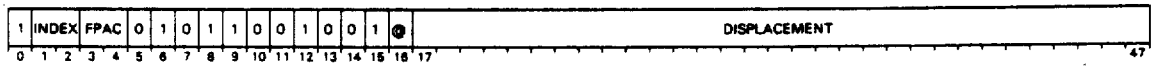
Moves four words out of memory and into a specified FPAC.

Computes the effective address, *E*. Fetches the double precision floating point number at the address specified by *E* and places it in FPAC. Updates the *Z* and *N* flags in the FPSR to reflect the new contents of FPAC.

NOTE: *This instruction will move unnormalized data without change.*

Floating Point Load Single (Long Displacement)

LFLDS *fpac,[@]displacement[,index]*



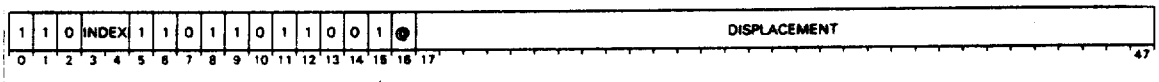
Moves two words out of memory into a specified FPAC.

Computes the effective address *E*. Fetches the single precision floating point number at the address specified by *E*. Places the number in the high-order bits of FPAC. Sets the low-order 32 bits of FPAC to 0. Updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

NOTE: *This instruction will move unnormalized or illegal data without change, but the Z and N flags will be undefined.*

Load Floating Point Status (Long Displacement)

LFLST *[@]displacement[,index]*



Moves the contents of two specified memory locations to the floating point status register.

Computes the effective address, *E*. Places the 32-bit operand addressed by *E* in the floating point status register as follows:

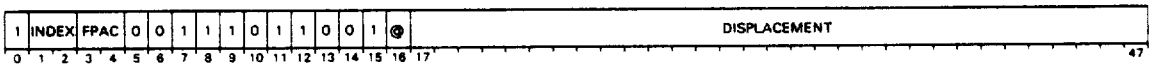
- Places bits 0-15 of the operand in bits 0-15 of the FPSR. Sets bits 16-32 of the FPSR to 0.
- If *ANY* is 0, bits 33-63 of the FPSR are undefined.
- If *ANY* is 1, the instruction places the value of the current segment in bits 33-35 of the FPSR, zeroes in bits 36-48, and bits 17-31 of the operand in bits 49-63 of the FPSR.

> **NOTES:** *This instruction does not set the* ANY *flag from memory. If any of bits 1-4 are loaded as 1,* ANY *is set to 1; otherwise,* ANY *is 0.*
>
> *Bits 12-15 of the FPSR are not set from memory. These bits are the floating point identification code an are read protected. In the MV/8000 they are set to 0111.*
>
> *This instruction initiates a floating point trap if* ANY *and* TE *are both 1 after the FPPC is loaded.*

See Chapter 8 and Appendix G for more information about floating point manipulation.

Multiply Double (FPAC by Memory) (Long Displacement)

LFMMD *fpac,[@]displacement[,index]*



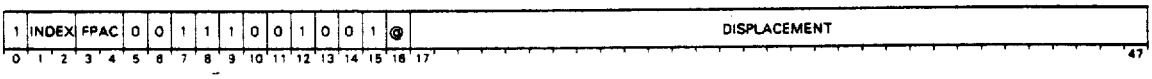
Multiplies the 64-bit floating point number in the source location by the 64-bit floating point number in FPAC and places the normalized result in FPAC.

Computes the effective address, *E*. Uses *E* to address a double precision (four word) operand. Multiplies this 64-bit floating point number by the floating point number in the specified FPAC. Places the normalized result in the specified FPAC. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Multiply Single (FPAC by Memory) (Long Displacement)

LFMMS *fpac,[@]displacement[,index]*



Multiplies the 32-bit floating point number in the source location by the 32-bit floating point number in bits 0-31 of FPAC and places the normalized result in FPAC.

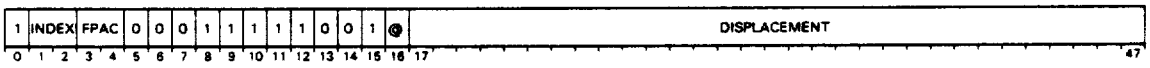
Computes the effective address, *E*. Uses *E* to address a single precision (double word) operand. Multiplies this 32-bit floating point number by the floating point number in

bits 0-31 of the specified FPAC. Places the normalized result in bits 0-31 of the specified FPAC. Sets bits 32-63 of FPAC to 0. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Subtract Double (Memory from FPAC) (Long Displacement)

LFSMD *fpac,[@]displacement[,index]*



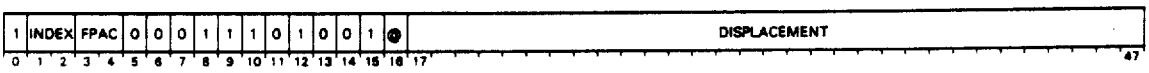
Subtracts the 64-bit floating point number in the source location from the 64-bit floating point number in FPAC and places the normalized result in FPAC.

Computes the effective address, *E*. Uses *E* to address a double precision (four word) operand. Subtracts this 64-bit floating point number from the floating point number in the specified FPAC. Places the normalized result in the specified FPAC. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Subtract Single (Memory from FPAC) (Long Displacement)

LFSMS *fpac,[@]displacement[,index]*



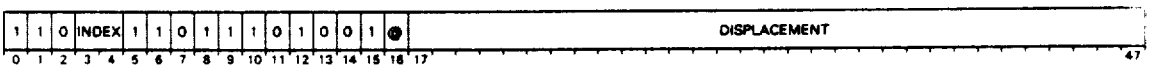
Subtracts the 32-bit floating point number in the source location from the 32-bit floating point number in bits 0-31 of FPAC and places the normalized result in FPAC.

Computes the effective address, *E*. Uses *E* to address a single precision (double word) operand. Subtracts this 32-bit floating point number from the floating point number in bits 0-31 of the specified FPAC. Places the normalized result in the specified FPAC. Sets bits 32-63 of FPAC to 0. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Store Floating Point Status (Long Displacement)

LFSST *[@]displacement[,index]*



Moves the contents of the FPSR to four specified memory locations.

Computes the effective address, *E*, of two sequential, 32-bit locations in memory. Stores the contents of the FPSR in these locations as follows:

- Stores bits 0-15 of the FPSR in the first memory word.
- Sets bits 16-31 of the first memory double word and bit 0 of the second memory double word to 0.

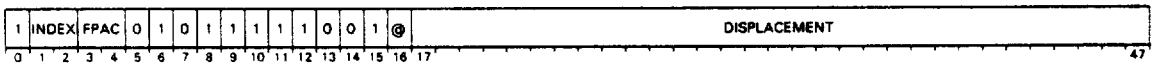
- If *ANY* is 0, the contents of bits 1-31 of the second memory double word are undefined.
- If *ANY* is 1, the instruction stores bits 33-63 of the FPSR into bits 1-31 of the second memory double word.

NOTE: *This instruction does not initiate a floating point trap under any conditions of the FPSR.*

See Chapter 8 and Appendix G for more information about floating point manipulation.

Store Floating Point Double (Long Displacement)

LFSTD *fpac,[@]displacement[,index]*

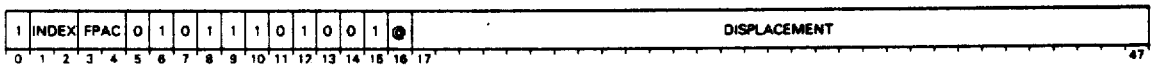


Stores the contents of a specified FPAC into a memory location.
Computes the effective address, *E*. Places the floating point number contained in FPAC in memory beginning at the location addressed by *E*. Destroys the previous contents of the addressed memory location. The contents of FPAC and the condition codes in the FPSR remain unchanged.

NOTE: *This instruction moves unnormalized or illegal data without change.*

Store Floating Point Single (Long Displacement)

LFSTS *fpac,[@]displacement[,index]*



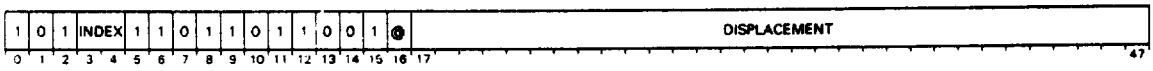
Stores the contents of a specified FPAC into a memory location.

Computes the effective address *E*. Places the 32 high-order bits of FPAC in memory beginning at the location addressed by *E*. Destroys the previous contents of the addressed memory location. The contents of FPAC and the condition codes in the FPSR remain unchanged.

NOTE: *This instruction moves unnormalized or illegal data without change.*

Jump (Long Displacement)

LJMP *index,displacement*

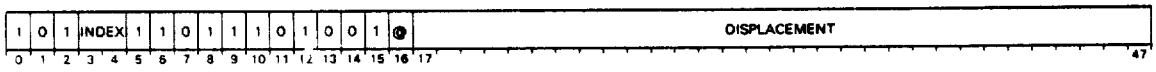


Calculates the effective address *E*. Loads *E* into the PC. Carry is unchanged and *overflow* is 0.

NOTE: *The calculation of* E *is forced to remain within the current segment of execution.*

Jump to Subroutine (Long Displacement)

LJSR *index,displacement*



Loads AC3 with the current 31-bit value of the program counter plus three. Loads the PC with the effective address. Carry is unchanged and *overflow* is 0.

NOTE: *The calculation of* E *is forced to remain within the current segment of execution.*

Load Byte (Long Displacement)

LLDB *ac,index,displacement*

1	INDEX	AC	1	0	0	1	1	0	0	1	0	0	1	DISPLACEMENT
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14 15 16 ... 47

Calculates the effective byte address. Uses the byte address to reference a byte in memory. Loads the addressed byte into bits 24–31 of the specified accumulator, then zero extends the value to 32 bits. Carry is unchanged and *overflow* is 0.

Load Effective Address (Long Displacement)

LLEF *ac,index,displacement*

1	INDEX	AC	0	1	1	1	1	1	0	1	0	0	1	●	DISPLACEMENT
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14 15 16 17	... 47

Calculates the effective address, *E*. Checks for segment crossing violation. If no violation occurs, loads *E* into the specified accumulator. If a violation occurs, issues a protection fault. Carry is unchanged and *overflow* is 0.

Load Effective Byte Address (Long Displacement)

LLEFB *ac,index,displacement*

1	INDEX	AC	1	0	0	1	1	1	0	1	0	0	1	DISPLACEMENT
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14 15 16 ... 47

Calculates a byte address. Checks for segment crossing violation. If no violation occurs, loads the byte address into the specified accumulator. If a violation occurs, issues a protection fault. Carry is unchanged and *overflow* is 0.

Load Modified and Referenced Bits

LMRF

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

Loads the modified and referenced bits of a pageframe into AC0.

AC1 contains a pageframe number in bits 13–31.

The instruction loads the modified and referenced bits of the pageframe specified by AC1 into AC0. The bits are loaded right justified and zero filled. The instruction then resets the referenced bit just accessed to 0. Carry is unchanged and *overflow* is 0.

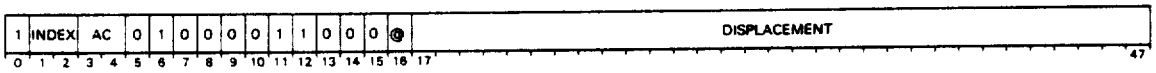
If the ATU is not enabled, undefined results will occur.

Specification of a non-existant pageframe results in an indeterminate data.

NOTE: *This is a privileged instruction.*

Narrow Add Memory Word to Accumulator (Long Displacement)

LNADD *ac,index,displacement*

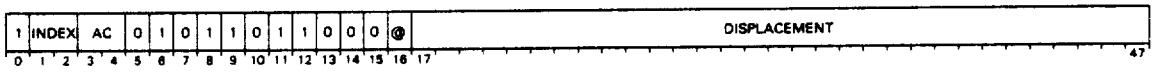


Adds an integer in memory to an integer contained in an accumulator.

The instruction calculates the effective address, *E*. Adds the 16-bit integer contained in the location specified by *E* to the integer contained in bits 16–31 of the specified accumulator. Sign extends the 16-bit result to 32 bits and loads it into the specified accumulator. Sets carry to the value of ALU carry, and *overflow* to 1 if there is an ALU overflow. The contents of the referenced memory location remain unchanged.

Narrow Divide Memory Word (Long Displacement)

LNDIV *ac,index,displacement*

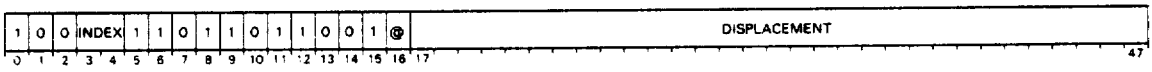


Divides an integer contained in an accumulator by an integer in memory.

The instruction calculates the effective address, *E*. Sign extends the integer contained in bits 16–31 of the specified accumulator to 32 bits and divides it by the 16-bit integer contained in the location specified by *E*. If the quotient is within the range -32,768 to +32,767 inclusive, sign extends the result to 32 bits and loads it into the specified accumulator. If the quotient is outside of this range, or the memory word is zero, the instruction sets *overflow* to 1 and leaves the specified accumulator unchanged. Otherwise, *overflow* is 0. The contents of the referenced memory location and carry remain unchanged.

Narrow Decrement and Skip if Zero (Long Displacement)

LNDSZ *index,displacement*

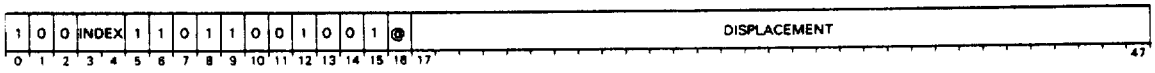


Calculates the effective address *E*. Decrements by one the contents of the 16-bit memory location addressed by *E*. If the result is equal to zero, then the next sequential word is skipped. Carry is unchanged and *overflow* is 0.

NOTE: *This instruction is indivisible.*

Narrow Increment and Skip if Zero (Long Displacement)

LNISZ *index,displacement*



Calculates the effective address, *E*. Increments by one the contents of the 16-bit memory location addressed by *E*. If the result is equal to zero, then the instruction skips the next sequential word. Carry is unchanged and *overflow* is 0.

NOTE: *This instruction is indivisible.*

Narrow Load Accumulator (Long Displacement)

LNLDA *ac,index,displacement*

1	INDEX	AC	0	1	1	1	1	0	0	1	0	0	1	@	DISPLACEMENT
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15 16 17 ... 47

Calculates the effective address, *E*. Fetches the 16-bit fixed point integer contained in the location specified by *E*. Sign extends this integer to 32 bits and loads it into the specified accumulator. Carry is unchanged and *overflow* is 0.

Narrow Multiply Memory Word (Long Displacement)

LNMUL *ac,index,displacement*

1	INDEX	AC	0	1	0	1	0	0	1	1	0	0	0	@	DISPLACEMENT
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15 16 17 ... 47

Multiplies an integer in memory by an integer in an accumulator.

Calculates the effective address, *E*. Multiplies the 16-bit, signed integer contained in the location referenced by *E* by the signed integer contained in bits 16–31 of the specified accumulator. If the result is outside the range of -32,768 to +32,767 inclusive, sets *overflow* to 1; otherwise, *overflow* is 0. Sign extends the result to 32 bits and places the result in the specified accumulator. The contents of the referenced memory location and carry remain unchanged.

Narrow Store Accumulator (Long Displacement)

LNSTA *ac,index,displacement*

1	INDEX	AC	0	1	1	1	1	0	1	1	0	0	1	@	DISPLACEMENT
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15 16 17 ... 47

Computes the effective address, *E*. Stores the low-order 16 bits of the specified accumulator into the location specified by *E*. Carry is unchanged and *overflow* is 0.

Narrow Subtract Memory Word (Long Displacement)

LNSUB *ac,index,displacement*

1	INDEX	AC	0	1	0	0	1	0	1	1	0	0	0	@	DISPLACEMENT
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15 16 17 ... 47

Subtracts an integer in memory from an integer in an accumulator.

Calculates the effective address, *E*. Subtracts the 16-bit integer contained in the location referenced by *E* from the integer contained in bits 16–31 of the specified accumulator. Sign extends the result to 32 bits and stores it in the specified accumulator. Sets carry to the value of ALU carry, and *overflow* to 1 if there is an ALU overflow. The contents of the specified memory location remain unchanged.

Locate Lead Bit

LOB *acs,acd*

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

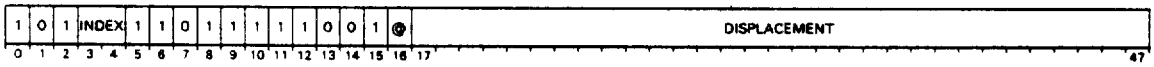
Adds a number equal to the number of high-order zeroes in the contents of bits 16-31 of ACS to the signed, 16-bit, two's complement number contained in bits 16-31 of ACD. The contents of ACS and the state of carry remain unchanged. *Overflow* is 0.

Bits 0-15 of the modified accumulator are undefined after completion of this instruction.

> NOTE: *If ACS and ACD are specified as the same accumulator, the instruction functions as described above, except that since ACS and ACD are the same accumulator, the contents of ACS will be changed.*

Push Address (Long Displacement)

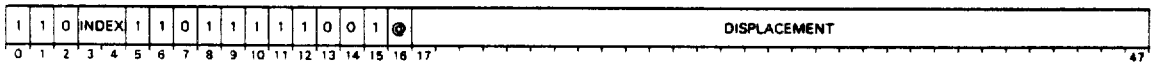
LPEF *index, displacement*



Pushes an effective address onto the wide stack. Carry is unchanged and *overflow* is 0.

Push Byte Address (Long Displacement)

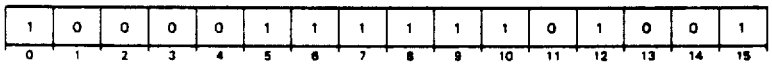
LPEFB *index, displacement*



Calculates an effective byte address. Pushes this byte address onto the wide stack. Carry is unchanged and *overflow* is 0.

Load Physical

LPHY



Translates the logical address contained in AC1 to a physical address.

AC1 contains a logical word address.

If the ATU is disabled, this instruction does nothing. The next word is executed.

If the ATU is enabled, then the actions described below occur.

The instruction compares the ring field of AC1 to the current ring. If AC1's ring field is less than or equal to the current ring field, then a protection fault (AC1 = 4) occurs.

If AC1's ring field is greater than the current ring, then the instruction references the SBR specified by AC1. If the SBR contents are invalid, then the instruction ends and the next instruction is executed. The contents of AC0 will be unchanged.

If the contents of the SBR are valid, the instruction loads AC0 with the last resident PTE. If the PTE indicates no page or validity faults, the instruction loads AC2 with the 32-bit physical word address of the logical address contained in AC1. The next sequential word is skipped.

If the PTE signals a page or validity fault, the contents of AC2 remain unchanged. The next sequential word is executed.

The instruction leaves carry unchanged; *overflow* is 0.

Push Jump (Long Displacement)

LPSHJ *index,displacement*

1	1	0	INDEX	1	1	0	1	1	0	0	1	0	0	1	@	DISPLACEMENT
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16 17 ... 47

Saves a return address on the wide stack and jumps to a specified location.

Pushes the current 31-bit value of the program counter + 3 onto the wide stack. Calculates the effective address, *E*. Loads the PC with *E*. Sequential operation continues with the word addressed by the updated value of the program counter. Carry is unchanged and *overflow* is 0.

NOTE: *The value pushed onto the wide stack will always point to a location in the current ring.*

Load Processor Status Register into AC0

LPSR

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

Loads the contents of the PSR into AC0.

Loads the contents of **OVK, OVR,** and **IRES** into bits 0, 1, and 2 of AC0, respectively. Fills the rest of AC0 with zeroes. The contents of the PSR remain unchanged. Carry is unchanged and *overflow* is 0.

Locate and Reset Lead Bit

LRB *acs,acd*

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

Performs a *Locate Lead Bit* instruction, and sets the lead bit to 0.

Adds a number equal to the number of high-order zeroes in the contents of bits 16-31 of ACS to the signed, 16-bit, two's complement number contained in bits 16-31 of ACD. Sets the leading 1 in bits 16-31 of ACS to 0. Carry remains unchanged and *overflow* is 0.

Bits 0-15 of the modified accumulator are undefined after completion of this instruction.

NOTE: *If ACS and ACD are specified to be the same accumulator, then the instruction sets the leading 1 in that accumulator to 0, and no count is taken.*

Load All Segment Base Registers

LSBRA

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

Loads the SBRs with new values.

AC0 contains the starting address of an 8 double word block.

The instruction loads a copy of the contents of these words into the SBRs as shown in the table below:

Double Word in Block	Destination	Order Moved
1	SBR1	First
2	SBR2	Second
3	SBR3	Third
4	SBR4	Fourth
5	SBR5	Fifth
6	SBR6	Sixth
7	SBR7	Seventh
8	SBR7	Eighth

After loading the SBRs, the instruction purges the ATU. If the ATU was disabled at the beginning of this instruction cycle, the processor enables it now.

If an invalid address is loaded into SBR0, the processor disables the ATU and a protection fault occurs (code = 3 in AC1). This means that logical addresses are identical to physical addresses, and the fault is processed in physical address space.

The instruction leaves AC0 and carry unchanged; *overflow* is 0.

NOTE: *This is a privileged instruction.*

Load Segment Base Registers 1-7
LSBRS

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

Loads SBR1 through SBR7 with new values.

AC0 contains the starting address of a block of seven double words. The instruction loads a copy of the contents of these words into the SBRs as shown in the table below:

Double Word in Block	Destination	Order Moved
1	SBR1	First
2	SBR2	Second
3	SBR3	Third
4	SBR4	Fourth
5	SBR5	Fifth
6	SBR6	Sixth
7	SBR7	Seventh

After loading the SBRs, the instruction purges the ATU. If the ATU was disabled at the beginning of this instruction cycle, the processor enables it now.

If SBR0 contains invalid information, then the processor disables the ATU and a protection fault occurs (code = 3 in AC1). This means that logical addresses are identical to physical addresses, and the fault is processed in physical address space.

The instruction leaves AC0 and carry unchanged; *overflow* is 0.

NOTE: *This is a privileged instruction.*

Logical Shift
LSH *acs,acd*

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

Shifts the contents of bits 16-31 of ACD either left or right depending on the number contained in bits 24–31 of ACS. The signed, 8-bit two's complement number contained in bits 24–31 of ACS determines the direction of the shift and the number of bits to be shifted. If the number in bits 24–31 of ACS is positive, shifting is to the left; if the number in bits 24–31 of ACS is negative, shifting is to the right. If the number in bits 24–31 of ACS is zero, no shifting is performed. Bits 16–23 of ACS are ignored.

The number of bits shifted is equal to the magnitude of the number in bits 24–31 of ACS. Bits shifted out are lost, and the vacated bit positions are filled with zeroes. Carry and the contents of ACS remain unchanged. *Overflow* is 0.

Bits 0-15 of the modified accumulator are undefined after completion of this instruction.

NOTE: *If the magnitude of the number in bits 24–31 of ACS is greater than 15, all bits of ACD are set to 0. Carry and the contents of ACS remain unchanged.*

Load Sign
LSN

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

Under the control of accumulators AC1 and AC3, evaluates a decimal number in memory and returns in AC1 a code that classifies the number as zero or nonzero and identifies its sign. The meaning of the returned code is as follows:

Value of Number	Code
Positive non-zero	+1
Negative non-zero	-1
Positive zero	0
Negative zero	-2

Bits 16-31 of AC1 must contain the data type indicator describing the number.

Bits 16-31 of AC3 must contain a byte pointer which is the address of the high-order byte of the number.

Upon successful termination, the contents of AC0 remain unchanged; AC1 contains the value code; AC2 contains the original contents of AC3; and the contents of AC3 are unpredictable. Carry remains unchanged. The contents of the addressed memory locations remain unchanged. *Overflow* is 0.

The 32-bit effective address generated by this instruction is constrained to be within the first 64 Kbyte of the current segment.

Store Byte (Long Displacement)

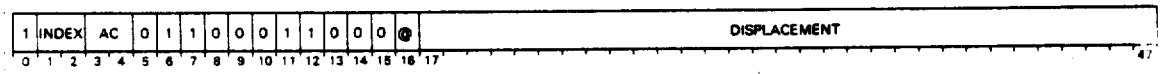
LSTB *ac,index,displacement*

Stores the low-order byte of the specified accumulator in memory.

Calculates the effective byte address. Moves a copy of the contents of bits 24–31 of the specified accumulator into memory at the location specified by the byte address. Carry is unchanged and *overflow* is 0.

Wide Add Memory Word to Accumulator (Long Displacement)

LWADD *ac,index,displacement*

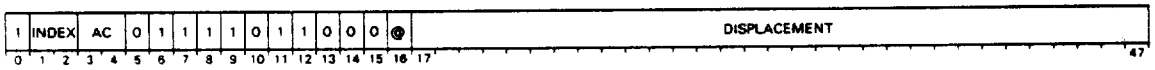


Adds an integer in memory to an integer in an accumulator.

The instruction calculates the effective address, *E*. Adds the 32-bit integer contained in the location specified by *E* to the 32-bit integer contained in the specified accumulator. Loads the result into the specified accumulator. Sets carry to the value of ALU carry, and *overflow* to 1 if there is an ALU overflow. The contents of the referenced memory location remain unchanged.

Wide Divide From Memory (Long Displacement)

LWDIV *ac,index,displacement*



Divides an integer in an accumulator by an integer in memory.

The instruction calculates the effective address, *E*. Sign extends the 32-bit integer contained in the specified accumulator to 64 bits and divides it by the 32-bit integer contained in the location specified by *E*.

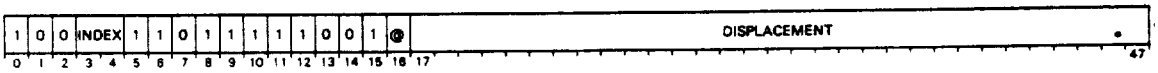
If the quotient is within the range of -2,147,483,648 to +2,147,483,647 inclusive, the instruction loads the quotient into the specified accumulator. *Overflow* is 0.

If the quotient is outside this range, or if the word in memory is zero, the instruction sets *overflow* to 1 and leaves the specified accumulator unchanged.

The contents of the referenced memory location and carry remain unchanged.

Wide Decrement and Skip if Zero (Long Displacement)

LWDSZ *index,displacement*

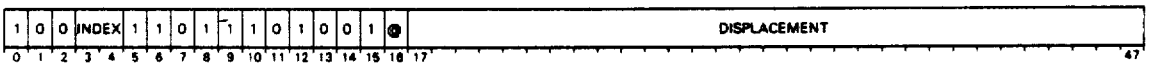


Decrements the contents of a 32-bit memory location by one. If the result is equal to zero, then the instruction skips the next sequential word. Carry is unchanged and *overflow* is 0.

NOTE: *This instruction executes in one indivisible memory cycle if the instruction is located on a double word boundary.*

Wide Increment and Skip if Zero (Long Displacement)

LWISZ *index,displacement*

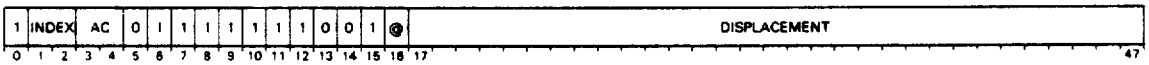


Increments the contents of a 32-bit memory location by one. If the result is equal to zero, then the instruction skips the next sequential word. Carry is unchanged and *overflow* is 0.

NOTE: *This instruction executes in one indivisible memory cycle if the instruction is located on a double word boundary.*

Wide Load Accumulator (Long Displacement)

LWLDA *ac,index,displacement*

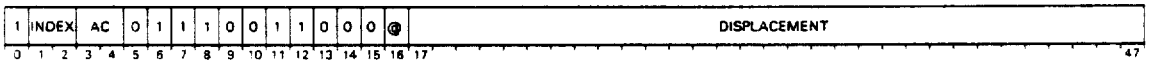


Loads the contents of a memory location into an accumulator.

Calculates the effective address, *E*. Fetches the 32-bit fixed point integer contained in the location specified by *E*. Loads this integer into the specified accumulator. Carry is unchanged and *overflow* is 0.

Wide Multiply From Memory (Long Displacement)

LWMUL *ac,index,displacement*

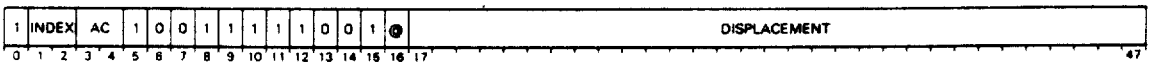


Multiplies an integer in an accumulator by an integer in memory.

The instruction calculates the effective address, *E*. Multiplies the 32-bit, signed integer contained in the location referenced by *E* by the 32-bit, signed integer contained in the specified accumulator. Places the 32 least significant bits of the result in the specified accumulator. The contents of the memory location and carry remain unchanged.
If the result is outside the range of -2,147,483,648 to +2,147,483,647 inclusive, sets *overflow* to 1; otherwise, *overflow* is 1. The specified accumulator will contain the 32 least significant bits of the result.

Wide Store Accumulator (Long Displacement)

LWSTA *ac,index,displacement*



Calculates the effective address, *E*. Stores the 32-bit contents of the specified accumulator in the location specified by *E*. Carry is unchanged and *overflow* is 0.

Wide Subtract Memory Word (Long Displacement)

LWSUB *ac,index,displacement*

1	INDEX	AC	0	1	1	0	1	0	1	1	0	0	0	@	DISPLACEMENT
0	1 2	3 4	5	6	7	8	9	10	11	12	13	14	15	16	17 … 47

Subtracts an integer in memory from an integer in an accumulator.

The instruction calculates the effective address, *E*. Subtracts the 32-bit integer contained in the location referenced by *E* from the 32-bit integer contained in the specified accumulator. Loads the result into the specified accumulator. Sets carry to the value of ALU carry, and *overflow* to 1 if there is an ALU overflow. The contents of the specified memory location remain unchanged.

Move

MOV*[c][sh][#]* *acs,acd[,skip]*

1	ACS	ACD	0	1	0	SH	C	#	SKIP
0	1 2	3 4	5	6	7	8 9	10 11	12	13 … 15

Moves the contents of bits 16-31 of an accumulator into another accumulator.

Initializes carry to the specified value. Places the contents of bits 16-31 of ACS in the shifter. Performs the specified shift operation and loads the result of the shift into bits 16-31 of ACD if the no-load bit is 0. If the skip condition is true, the instruction skips the next sequential word. *Overflow* is 0.

If the load option is specified, bits 0-15 of ACD are undefined.

Modify Stack Pointer

MSP *ac*

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

Changes the value of the stack pointer and tests for potential overflow.

Adds the signed two's-complement number in bits 16-31 of the specified accumulator to the value of the stack pointer and places the result in location 40. The instruction then checks for overflow by comparing the result in location 40 with the value of the stack limit. If the result in location 40 is less than the stack limit, then the instruction ends.

If the result is greater than the stack limit, the instruction changes the value of location 40 back to its original contents before the add. The instruction pushes a standard return block. The program counter in the return block contains the address of the *Modify Stack Pointer* instruction.

After pushing the return block, the program counter contains the address of the stack fault routine. The stack pointer is updated with the value used to push the return block, and control transfers to the stack fault routine. Carry remains unchanged and *overflow* is 0.

Unsigned Multiply

MUL

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

Multiplies the unsigned contents of two accumulators and adds the result to the unsigned contents of a third accumulator. The result is an unsigned 32-bit integer in two accumulators.

The unsigned, 16-bit number in bits 16-31 of AC1 is multiplied by the unsigned, 16-bit number in bits 16-31 of AC2 to yield an unsigned, 32-bit intermediate result. The unsigned, 16-bit number in bits 16-31 of AC0 is added to the intermediate result to produce the final result. The final result is an unsigned, 32-bit number and occupies bits 16-31 of both AC0 and AC1. Bit 16 of AC0 is the high-order bit of the result and bit 31 of AC1 is the low-order bit. The contents of AC2 remain unchanged.

Because the result is a double-length number, overflow cannot occur. Carry remains unchanged and *overflow* is 0.

Bits 0-15 of the modified accumulator are undefined after completion of this instruction.

Signed Multiply

MULS

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

Multiplies the signed contents of two accumulators and adds the result to the signed contents of a third accumulator. The result is a signed 32-bit integer in two accumulators.

The signed, 16-bit two's complement number in bits 16-31 of AC1 is multiplied by the signed, 16-bit two's complement number in bits 16-31 of AC2 to yield a signed, 32-bit two's complement intermediate result. The signed, 16-bit two's complement number in bits 16-31 of AC0 is added to the intermediate result to produce the final result. The final result is a signed, 32-bit two's complement number which occupies bits 16-31 of both AC0 and AC1. Bit 16 of AC0 is the sign bit of the result and bit 31 of AC1 is the low-order bit. The contents of AC2 remain unchanged.

Because the result is a double-length number, overflow cannot occur. Carry remains unchanged and *overflow* is 0.

Bits 0-15 of the modified accumulator are undefined after completion of this instruction.

Narrow Add

NADD *acs,acd*

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

Adds two integers contained in accumulators.

The instruction adds the 16-bit integer contained in bits 16–31 of ACS to the 16-bit integer contained in bits 16–31 of ACD. Stores the result in bits 16–31 of ACD. Sign extends ACD to 32 bits. Sets carry to the value of ALU carry, and sets *overflow* to 1 if there is an ALU overflow.

Narrow Extended Add Immediate

NADDI *n,ac*

1	1	0	AC	1	1	0	0	0	1	1	1	0	0	1	IMMEDIATE FIELD
0	1	2	3 4	5	6	7	8	9	10	11	12	13	14	15	16 31

Adds an integer contained in an immediate field to an integer in an accumulator.

Adds the 16-bit value contained in the immediate field to bits 16-31 of the specified accumulator. Stores the result in the lower 16 bits of ACD. Sign extends ACD to 32 bits. Sets carry to ALU carry (16 bit operation). Sets *overflow* to 1 if there is an ALU overflow (16 bit operation).

Narrow Add Immediate

NADI *n,ac*

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

Adds an integer in the range of 1 to 4 to an integer contained in an accumulator. The instruction adds the value $n+1$ to the 16-bit contents of the specified accumulator, where n is an integer in the range of 0 to 3. Stores the result in the lower 16 bits of the specified accumulator. Sign extends the specified accumulator to 32 bits. Sets carry to the value of ALU carry (16-bit operation). Sets *overflow* to 1 if there is an ALU overflow (16 bit operation).

NOTE: *The assembler takes the coded value of* n *and subtracts 1 from it before placing it in the immediate field. Therefore, the programmer should code the exact value that he wishes to add.*

Search Queue

*<width><direction>*S*<test condition>*

1	1	0	0	0	1	1	1	0	0	0	1	1	0	0	1	RESERVED	W	TEST	D
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16 26	27	28 30	31

Searches a queue for the first data element containing information that meets a specified condition.

AC1 contains the effective address of the first queue data element to search.

AC3 contains a two's complement word offset. The instruction adds the offset to the address of the forward link of each data element to get the address of the location to test (called the test location).

The double word on the top of the wide stack contains a mask word.

Bits 11–15 of the second word of the search instruction specify the conditions of the search. The table below explains the meanings of these bits.

Bits	Name of Field	Encoding	Mnemonic	Meaning
11	Width	0	N	Search field is 16 bits wide.
		1	W	Search field is 32 bits wide.
12-14	Test	000	SS	Some of the bits specified by the mask in the test condition are one.
		001	SC	Some of the bits specified by the mask in the test condition are zero.
		010	AS	All of the bits specified by the mask in the test condition are one.
		011	AC	All of the bits specified by the mask in the test condition are zero.
		100	E*	The mask and test location are equal.
		101	GE*	The mask is greater than or equal to the test location.
		110	LE*	The mask is less than or equal to the test location.
		111	NE*	The mask and the test location are not equal.
15	Direction	0	F	Search forward in the queue.
		1	B	Search backward in the queue.

NOTE: *The instruction treats the values contained in the mask and the test location as unsigned integers for these test conditions.*

The size of the field to search (bit 11) determines the size of the mask and the size of the offset. If you specify a narrow search, then bits 16–31 of the wide stack word contain the mask. AC3 specifies a relative word offset to the 16-bit test location. If you specify a wide search, then bits 0–31 of the wide stack word contain the mask. AC3 specifies the relative word offset to the 32-bit test location.

The instruction searches each data element in the queue from the element specified by AC1 to the head or tail of the queue (depending on the direction of the search). To perform the search on each element, the instruction adds the offset contained in AC3 to the address contained in AC1 to obtain an address of a location contained in some data element in the queue. Compares the mask field to the value contained in the calculated address.

If the search fails, AC1 contains the effective address of the last data element searched. Execution continues with the next sequential instruction. Interrupts are honored between the time the search fails and the time the next word executes.

If the search is interrupted, AC1 contains the effective address of the next data element to be examined. The next sequential word is skipped and execution continues with the second word. Interrupts are honored between the time the interrupt occurs and the time the second instruction executes.

If the search is successful, AC1 contains the address of the data element that met the specified condition. The next two sequential words are skipped and execution continues with the third word. Note that interrupts cannot occur between the time the search is successful and the time the third word executes.

For all returns, the contents of carry, WSP, AC0, AC2, and AC3 remain unchanged. *Overflow* is 0.

Narrow Search Queue Backward

NBSAC

1	1	0	0	0	1	1	1	0	0	0	1	1	0	0	1	RESERVED	0	0	1	1	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16 … 26	27	28	29	30	31

See instruction entry "Search Queue".

Narrow Search Queue Backward

NBSAS

1	1	0	0	0	1	1	1	0	0	0	1	1	0	0	1	RESERVED	0	0	1	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16 … 26	27	28	29	30	31

See instruction entry "Search Queue".

Narrow Search Queue Backward

NBSE

1	1	0	0	0	1	1	1	0	0	0	1	1	0	0	1	RESERVED	0	1	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16 … 26	27	28	29	30	31

See instruction entry "Search Queue".

Narrow Search Queue Backward

NBSGE

1	1	0	0	0	1	1	1	0	0	0	1	1	0	0	1	RESERVED	0	1	0	1	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16 … 26	27	28	29	30	31

See instruction entry "Search Queue".

Narrow Search Queue Backward

NBSLE

1	1	0	0	0	1	1	1	0	0	0	1	1	0	0	1	RESERVED	0	1	1	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16 … 26	27	28	29	30	31

See instruction entry "Search Queue".

Narrow Search Queue Backward

NBSNE

1	1	0	0	0	1	1	1	0	0	0	1	1	0	0	1	RESERVED	0	1	1	1	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16 … 26	27	28	29	30	31

See instruction entry "Search Queue".

Narrow Search Queue Backward

NBSSC

1	1	0	0	0	1	1	1	0	0	0	1	1	0	0	1	RESERVED	0	0	0	1	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16 … 26	27	28	29	30	31

See instruction entry "Search Queue".

Narrow Search Queue Backward
NBSSS

1	1	0	0	0	1	1	1	0	0	0	1	1	0	0	1	RESERVED	0	0	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16 ... 26	27	28	29	30	31

See instruction entry "Search Queue".

Narrow Divide

NDIV *acs,acd*

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

Divides an integer in an accumulator by an integer in another accumulator.

The instruction sign extends bits 16–31 of ACD to 32 bits. Divides this signed integer by the 16-bit signed integer contained in bits 16–31 of ACS. If the quotient is within the range -32,768 to +32,767 inclusive, sign extends the lower 16 bits of the result to 32 bits and places these 16 bits in ACD. If the quotient is outside of this range, or if ACS is zero, the instruction sets *overflow* to 1 and leaves ACD unchanged. Otherwise, *overflow* is 0. The contents of ACS and carry always remain unchanged.

Negate

NEG*[c][sh][#]* *acs,acd[,skip]*

1	ACS	ACD	0	0	1	SH	C	#	SKIP
0	1 2	3 4	5	6	7	8 9	10 11	12	13 ... 15

Forms the two's complement of the contents of bits 16-31 of an accumulator.

Initializes carry to the specified value. Places the two's complement of the unsigned, 16-bit number in bits 16-31 of ACS in the shifter. If the negate operation produces a carry of 1 out of the high-order bit, the instruction complements carry. Performs the specified shift operation and places the result in bits 16-31 of ACD if the no-load bit is 0. If the skip condition is true, the instruction skips the next sequential word. *Overflow* is 0.

If the load option is specified, bits 0-15 of ACD are undefined.

NOTE: *If ACS contains 0, the instruction complements carry.*

Narrow Search Queue Forward
NFSAC

1	1	0	0	0	1	1	1	0	0	0	1	1	0	0	1	RESERVED	0	0	1	1	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16 ... 26	27	28	29	30	31

See instruction entry "Search Queue".

Narrow Search Queue Forward
NFSAS

1	1	0	0	0	1	1	1	0	0	0	1	1	0	0	1	RESERVED	0	0	1	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16 ... 26	27	28	29	30	31

See instruction entry "Search Queue".

Narrow Search Queue Forward

NFSE

1	1	0	0	0	1	1	1	0	0	0	1	1	0	0	1	RESERVED	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16 ... 26	27	28	29	30	31

See instruction entry "Search Queue".

Narrow Search Queue Forward

NFSGE

1	1	0	0	0	1	1	1	0	0	0	1	1	0	0	1	RESERVED	0	1	0	1	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16 ... 26	27	28	29	30	31

See instruction entry "Search Queue".

Narrow Search Queue Forward

NFSLE

1	1	0	0	0	1	1	1	0	0	0	1	1	0	0	1	RESERVED	0	1	1	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16 ... 26	27	28	29	30	31

See instruction entry "Search Queue".

Narrow Search Queue Forward

NFSNE

1	1	0	0	0	1	1	1	0	0	0	1	1	0	0	1	RESERVED	0	1	1	1	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16 ... 26	27	28	29	30	31

See instruction entry "Search Queue".

Narrow Search Queue Forward

NFSSC

1	1	0	0	0	1	1	1	0	0	0	1	1	0	0	1	RESERVED	0	0	0	1	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16 ... 26	27	28	29	30	31

See instruction entry "Search Queue".

Narrow Search Queue Forward

NFSSS

1	1	0	0	0	1	1	1	0	0	0	1	1	0	0	1	RESERVED	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16 ... 26	27	28	29	30	31

See instruction entry "Search Queue".

Narrow Load Immediate

NLDAI *ac,immediate*

1	1	0	AC	1	1	0	0	0	1	0	1	0	0	1	IMMEDIATE FIELD
0	1	2	3 4	5	6	7	8	9	10	11	12	13	14	15	16 ... 31

Sign extends the 16-bit, two's complement literal value contained in the immediate field to 32 bits. Loads the result of the sign extension into the specified accumulator. Carry is unchanged and *overflow* is 0.

Narrow Multiply

NMUL *acs,acd*

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

Multiplies the signed integer contained in bits 16–31 of ACD by the signed integer contained in bits 16–31 of ACS. If the result is outside the range of -32,768 to +32,767 inclusive, sets *overflow* to 1; otherwise, *overflow* is 0. Sign extends the lower 16 bits of the result to 32 bits and places these 32 bits in ACD. The contents of ACS and carry remain unchanged.

Narrow Negate

NNEG *acs,acd*

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

Negates the 16 least significant bits of ACS by performing a two's complement subtract from zero. Sign extends these 16 bits to 32 bits and loads the result in ACD. Sets carry to the value of ALU carry.

NOTE: *Negating the largest negative 16-bit integer, (100000_8) sets overflow to 1.*

Narrow Skip on All Bits Set in Accumulator

NSALA *ac,immediate*

1	1	1	AC	1	1	0	0	0	0	0	1	0	0	1	IMMEDIATE FIELD
0	1	2	3 4	5	6	7	8	9	10	11	12	13	14	15	16 ... 31

Logically ANDs the value in the immediate field with the complement of the contents of an accumulator and skips depending on the result of the AND.

The instruction performs a logical AND on the contents of the immediate field and the complement of the least significant 16 bits contained in the specified accumulator. If the result of the AND is zero, then the next sequential word is skipped. If the result of the AND is nonzero, the next sequential word is executed. The contents of the specified accumulator remain unchanged. Carry is unchanged and *overflow* is 0.

Narrow Skip on All Bits Set in Memory Location

NSALM *ac,immediate*

1	1	1	AC	1	1	0	0	0	0	1	1	0	0	1	IMMEDIATE FIELD
0	1	2	3 4	5	6	7	8	9	10	11	12	13	14	15	16 ... 31

Performs a logical AND on the contents of the immediate field and the complement of the word addressed by the specified accumulator. If the result of the AND is zero, then

execution skips the next sequential word before continuing. If the result of the AND is nonzero, then execution continues with the next sequential word. The contents of the specified accumulator and memory location remain unchanged. Carry is unchanged and *overflow* is 0.

Narrow Skip on Any Bit Set in Accumulator

NSANA *ac,immediate*

1	1	1	ac	1	1	0	0	0	1	0	1	0	0	1	IMMEDIATE FIELD
0	1	2	3 4	5	6	7	8	9	10	11	12	13	14	15	16 ... 31

Logically ANDS the contents of an immediate field with the contents of an accumulator and skips, depending on the result.

The instruction performs a logical AND on the contents of the immediate field and the least significant 16 bits contained in the specified accumulator. If the result of the AND is nonzero, the next sequential word is skipped. If the result of the AND is zero, the next sequential word is executed. The contents of the specified accumulator remain unchanged. Carry is unchanged and *overflow* is 0.

Narrow Skip on Any Bit Set in Memory Location

NSANM *ac,immediate*

1	1	1	AC	1	1	0	0	0	1	1	1	0	0	1	IMMEDIATE FIELD
0	1	2	3 4	5	6	7	8	9	10	11	12	13	14	15	16 ... 31

The instruction performs a logical AND on the contents of the immediate field and the contents of the word addressed by the specified accumulator. If the result of the AND is nonzero, then the next sequential word is skipped. If the result of the AND is zero, the next sequential word is executed. The contents of the specified accumulator and memory location remain unchanged. Carry is unchanged and *overflow* is 0.

Narrow Subtract Immediate

NSBI *n,ac*

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

Subtracts a value in the range of 1 to 4 from the value contained in an accumulator.

The instruction subtracts the value $n+1$ from the 16-bit value contained in the specified accumulator, where n is an integer in the range of 0 to 3. Stores the result in bits 16–31 of the specified accumulator. Sign extends the specified accumulator to 32 bits. Sets carry to the value of ALU carry. Sets *overflow* to 1 if there is an ALU overflow.

> **NOTE:** *The assembler takes the coded value of n and subtracts 1 from it before placing it in the immediate field. Therefore, the programmer should code the exact value that he wishes to subtract.*

Narrow Subtract

NSUB *acs,acd*

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

Subtracts the 16-bit integer contained in bits 16–31 of ACS from the 16-bit integer contained in bits 16–31 of ACD. Stores the result in bits 16–31 of ACD. Sign extends ACD to 32 bits. Sets carry to the value of ALU carry, and *overflow* to 1 if there is an ALU overflow.

OR Referenced Bits

ORFB

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

AC1 contains a page frame number in bits 15-31. Bits 30-31 of AC1 are a don't care. The two least significant bits of AC1 are treated as if they are both 0. AC0 contains an origin 0 word count. This count indicates the number of words to be ORed. The referenced bits of 16 contiguous page frames are grouped for ORing. AC2 contains a word address. The referenced bits of the grouped 16 page frames beginning at the number specified in AC1 (treating bits 30-31 as 0) are inclusive ORed with a word. This word is part of a word string. The starting address of the word string is contained in AC2. The result of this ORing is stored back in the word string. The referenced bits accessed are then reset to "0" upon completion of the ORing operation, AC0 is decremented by 1, AC1 is incremented by 16, and AC2 is incremented by 1. If the MMU is not enabled, undefined results will occur. Specification of a non-existant page frame number results in an indeterminate data.

Purge the ATU

PATU

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

Purges the entire ATU of all entries. Carry is unchanged and *overflow* is 0.

NOTE: *This is a privileged instruction.*

Pop Block and Execute

PBX

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

Saves a 16-bit instruction, pops a wide return block off the stack, and executes the saved instruction. Carry and *overflow* are indeterminate.

Bits 16–31 of AC0 contain a 16-bit instruction.

The instruction temporarily saves the instruction contained in bits 16–31 of AC0. Executes a **WPOPB** instruction, except that execution does not continue with the value loaded into the PC. After the wide return block is popped, the instruction executes the instruction that was temporarily saved. The executed instruction determines the value of the processor flags. The next instruction to be executed is addressed by the popped value of the PC + 1.

Note that the value popped off the stack and loaded into the PC must reference a **BKPT** instruction. If it does not, undefined results occur. If it does, then the instruction effectively substitutes the 16-bit instruction in AC0 for the **BKPT** instruction referenced by the PC after the pop.

Pop Multiple Accumulators

POP *acs,acd*

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

Pops 1 to 4 words off the stack and places them in the indicated accumulators.

The set of accumulators from ACS through ACD, bits 16-31, is filled with words popped from the stack. Bits 16-31 of the accumulators are filled in descending order, starting with bits 16-31 of the accumulator specified by ACS and continuing down through bits 16-31 of the accumulator specified by ACD, wrapping around if necessary, with AC3 following AC0. If ACS is equal to ACD, only one word is popped and it is placed in ACS.

The stack pointer is decremented by the number of accumulators popped and the frame pointer is unchanged. A check for underflow is made only after the entire pop operation is done.

Bits 0-15 of the modified accumulator are undefined after completion of this instruction.

This instruction leaves carry unchanged; *overflow* is 0.

Pop Block

POPB

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

Returns control from a *System Call* routine or an I/O interrupt handler that does not use the stack change facility of the *Vector* instruction.

Five words are popped off the stack and placed in predetermined locations. The words popped and their destinations are as follows:

Word Popped	Destination
1	Bit 0 is loaded into carry Bits 1-15 are loaded into the PC
2	AC3
3	AC2
4	AC1
5	AC0

Sequential operation is continued with the word addressed by the updated value of the program counter. Carry remains unchanged and *overflow* is 0.

Bits 0-15 of the modified accumulator are undefined after completion of this instruction.

NOTE: *If the I/O handler uses the stack change facility of the* Vector on Interrupting Device Code *instruction, do not use the* Pop Block *instruction. Use the* Restore *instruction instead.*

Pop PC And Jump

POPJ

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

Pops the top word off the stack and places it in the program counter. Sequential operation continues with the word addressed by the updated value of the program counter.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

The stack pointer is decremented by one and the frame pointer is unchanged. A check for underflow occurs after the pop operation. Carry remains unchanged and *overflow* is 0.

Push Multiple Accumulators

PSH *acs,acd*

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

Pushes the contents of 1 to 4 accumulators onto the stack.

Bits 16-31 of the set of accumulators from ACS through ACD are pushed onto the stack. The contents of bits 16-31 of the accumulators are pushed in ascending order, starting with bits 16-31 of the AC specified by ACS and continuing up through bits 16-31 of the AC specified by ACD, wrapping around if necessary, with AC0 following AC3. The contents of the accumulators remain unchanged. If ACS equals ACD, only ACS is pushed. Carry remains unchanged and *overflow* is 0.

The stack pointer is incremented by the number of accumulators pushed and the frame pointer is unchanged. A check for overflow is made only after the entire push operation finishes.

Push Jump

PSHJ *[@]displacement[,index]*

1	0	0	0	0	1	INDEX		1	0	1	1	1	0	0	0	@	DISPLACEMENT
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17 ... 31

Pushes the address of the next sequential instruction onto the stack, computes the effective address, *E*, and places it in the program counter. Sequential operation continues with the word addressed by the updated value of the program counter.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

Carry remains unchanged and *overflow* is 0.

Push Return Address

PSHR

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

Pushes the address of this instruction + 2 onto the narrow stack. Carry remains unchanged and *overflow* is 0.

Reset Referenced Bit

RRFB

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

Resets the specified referenced bits.

AC1 contains *p*, a pageframe number, in bits 13–31.

AC0 contains *c*, an origin 0 pageframe count (a count of 0 implies 1 page).

The instruction loads a zero into the referenced bits of *c* pageframes, beginning with the pageframe specified by AC1. The last pageframe whose referenced bit will be changed is pageframe number $p + c$. If the ATU is not enabled, undefined results will occur. Specification of a non-existent pageframe results in indeterminate data.

At the end of the instruction, AC1 contains the sum of the original contents of AC1 + the contents of AC0 plus 1. AC0 contains -1 (all ones). Carry is unchanged and *overflow* is 0.

NOTE: *This is a privileged instruction.*

Restore

RSTR

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

Returns control from certain types of I/O interrupts.

Pops nine words off the stack and places them in predetermined locations. The words popped and their destinations are as follows:

Word Popped	Destination
1	Bit 0 is loaded into carry Bits 1–15 are loaded into the PC
2	AC3
3	AC2
4	AC1
5	AC0
6	Stack fault address
7	Stack limit
8	Frame pointer
9	Stack pointer

Sequential operation continues with the word addressed by the updated value of the program counter.

Bits 0-15 of the modified accumulator are undefined after completion of this instruction.

Carry remains unchanged and *overflow* is 0.

NOTES: *Use the* Restore *instruction to return control to the program only if the I/O interrupt handler uses the stack change facility of the* Vector on Interrupting Device Code *instruction.*

The Restore *instruction does not check for stack underflow.*

Return
RTN

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

Returns control from subroutines that issue a *Save* instruction at their entry points.

The *Save* instruction loads the current value of the stack pointer into the frame pointer. The *Return* instructions uses this value of the frame pointer to pop a standard return block off of the stack. The format of the return block is:

Word Popped	Destination
1	Bit 0 is loaded into carry Bits 1-15 are loaded into the PC
2	AC3
3	AC2
4	AC1
5	AC0

After popping the return block, the *Return* instruction loads the decremented value of the frame pointer into the stack pointer and the popped value of AC3 into the frame pointer.

Bits 0-15 of the modified accumulator are undefined after completion of this instruction.

Carry remains unchanged and *overflow* is 0.

Save
SAVE *i*

1	1	1	0	0	1	1	1	1	1	0	0	1	0	0	0	IMMEDIATE FIELD
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16 ... 31

Saves the information required by the *Return* instruction.

Saves the current value of the stack pointer in a temporary location. Adds 5 plus the unsigned, 16-bit integer contained in the immediate field to the current value of the stack pointer and loads the result into location 40. Compares this new value of the stack pointer to the stack limit to check for overflow. If no overflow condition exists, then the instruction places the current value of the frame pointer in bits 16-31 of AC3. Fetches the contents of the temporary location and loads them into the frame pointer. The instruction uses the value in the frame pointer to push a five-word return block. The formats and contents of the five-word return block is as follows:

Word Pushed	Contents
1	Bits 16-31 of AC0
2	Bits 16-31 of AC1
3	Bits 16-31 of AC2
4	Frame pointer before the *Save*
5	Bit 0 = carry Bits 1-15 = bits 16-31 of AC3

After pushing the return block on the narrow stack, the instruction places the value of the frame pointer (which now contains the old value of the stack pointer + 5) in bits 16-31 of AC3. Carry remains unchanged and *overflow* is 0.

If an overflow condition exists, the *Save* instruction transfers control to the stack fault routine. The program counter in the fault return block contains the address of the *Save* instruction.

The *Save* instruction allocates a portion of the stack for use by the procedure which executed the *Save*. The value of the *frame size*, contained in the immediate field, determines the number of words in this stack area. This portion of the stack will not normally be accessed by push and pop operations, but will be used by the procedure for temporary storage of variables, counters, etc. The frame pointer acts as the reference point for this storage area.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

Use the *Save* instruction with the *Jump to Subroutine* instruction. The *Jump to Subroutine* instruction places the return value of the program counter in bits 16-31 of AC3. *Save* then pushes the return value (contents of bits 16-31 of AC3) into bits 1–15 of the fifth word pushed.

Subtract Immediate

SBI *n,ac*

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

Subtracts an unsigned integer in the range 1 to 4 from the contents of an accumulator.

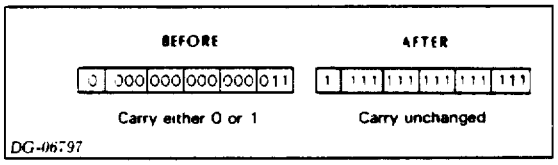
The instruction subtracts the value $N+1$ from the unsigned 16-bit number contained in bits 16-31 of the specified accumulator and the result is placed in bits 16-31 of ACD. Carry remains unchanged. *Overflow* is 0.

Bits 0-15 of the modified accumulator are undefined after completion of this instruction.

> **NOTE:** *The assembler takes the coded value of n and subtracts 1 from it before placing it in the immediate field. Therefore the programmer should code the exact value he wishes to subtract.*

Example

Assume that bits 16-31 of AC2 contains 000003_8. After the instruction **SBI 4,2** is executed, bits 16-31 of AC2 contains 177777_8 and carry remains unchanged.



Sign Extend

SEX *acs,acd*

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

Sign extends the 16-bit integer contained in ACS to 32 bits and loads the result into ACD. The contents of ACS remain unchanged, unless ACS and ACD are specified to be the same accumulator. Carry is unchanged and *overflow* is 0.

Skip If ACS Greater Than Or Equal to ACD

SGE *acs,acd*

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

Compares two signed integers in two accumulators and skips if the first is greater than or equal to the second.

The signed two's complement numbers in bits 16-31 of ACS and ACD are algebraically compared. If the number in bits 16-31 of ACS is greater than or equal to the number in bits 16-31 of ACD, the next sequential word is skipped. The contents of ACS, ACD, and carry remain unchanged. *Overflow* is 0.

NOTE: *The* Skip If ACS Greater Than ACD *and* Skip If ACS Greater Than Or Equal To ACD *instructions treat the contents of the specified accumulators as signed, two's complement integers. To compare unsigned integers, use the* Subtract and Add Complement *instruction.*

Skip If ACS Greater Than ACD

SGT *acs,acd*

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

Compares two signed integers in two accumulators and skips if the first is greater than the second.

The signed, two's complement numbers in bits 16-31 of ACS and ACD are algebraically compared. If the number in bits 16-31 of ACS is greater than the number in bits 16-31 of ACD, the next sequential word is skipped. The contents of ACS, ACD, and carry remain unchanged.

NOTE: *The* Skip If ACS Greater Than ACD *and* Skip If ACS Greater Than Or Equal To ACD *instructions treat the contents of the specified accumulators as signed, two's complement integers. To compare unsigned integers, use the* Subtract and Add Complement *instruction.*

Store Modified and Referenced Bits

SMRF

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

Stores new values into the modified and referenced bits of a pageframe.

AC1 contains a pageframe number in bits 13 -31.

The instruction fetches the contents of the two least significant bits of AC0. Stores these values in the modified and referenced bits of the pageframe specified by AC1. Carry is unchanged and *overflow* is 0.

If the ATU is not enabled, undefined results will occur. If a nonexistent pageframe is specified, the instruction loads the appropriate modified and referenced bits with indeterminate data.

NOTE: *This is a privileged instruction.*

Skip On Non-Zero Bit

SNB *acs,acd*

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

The two accumulators form a bit pointer. If the addressed bit is 1, the next sequential word is skipped.

Forms a 32-bit bit pointer from the contents of bits 16-31 of both ACS and ACD. Bit: 16-31 of ACS contains the high-order 16 bits and bits 16-31 of ACD contains the low-order 16 bits of the bit pointer. If ACS and ACD are specified as the same accumulator, the instruction treats the accumulator contents as the low-order 16 bits of the bit pointer and assumes the high-order 16 bits are 0.

If the addressed bit in memory is 1, the next sequential word is skipped. The contents of ACS, ACD, and carry remain unchanged. *Overflow* is 0. The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

NOTE: *The bit pointer formed by the two accumulators cannot make indirect memory references.*

Skip on OVR Reset

SNOVR

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

Tests the value of **OVR**. If the flag has the value 0, the next sequential word is skipped. If the flag has the value 1, the next sequential word is executed. Carry is unchanged and *overflow* is 0.

Store Processor Status Register From AC0

SPSR

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

Stores the contents of AC0 in the PSR.

Loads the contents of AC0 bits 0, 1, and 2 into **OVK**, **OVR**, and **IRES**, respectively. The contents of AC0 remain unchanged. Carry is unchanged and *overflow* is 0.

Store Accumulator

STA *ac,[@]displacement[,index]*

0	1	0	AC		@	INDEX		DISPLACEMENT
0	1	2	3	4	5	6	7	8 ... 15

Stores the contents of bits 16-31 of an accumulator into a memory location.

Places the contents of bits 16-31 of the specified accumulator in the word addressed by the effective address, *E*. The previous contents of the location addressed by *E* are lost.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

The contents of carry and the specified accumulator remain unchanged. *Overflow* is 0.

Store Accumulator in WFP

STAFP *ac*

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

Stores a copy of the contents of the specified accumulator into WFP (the wide frame pointer). Carry is unchanged and *overflow* is 0.

Store Accumulator in WSB

STASB *ac*

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

Stores a copy of the contents of the specified accumulator into WSB (the wide stack base) as well as locations $26–27_8$ of the current segment. Carry is unchanged and *overflow* is 0.

Store Accumulator in WSL

STASL *ac*

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

Stores a copy of the contents of the specified accumulator into WSP (the wide stack pointer) as well as locations $24–25_8$ of the current segment. Carry is unchanged and *overflow* is 0.

Store Accumulator in WSP

STASP *ac*

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

Stores a copy of the contents of the specified accumulator into WSP (the wide stack pointer). Carry is unchanged and *overflow* is 0.

Store Accumulator into Stack Pointer Contents

STATS *ac*

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

Uses the contents of WSP (the wide stack pointer) as the address of a double word. Stores a copy of the contents of the specified accumulator at the address contained in WSP. Carry is unchanged and *overflow* is 0.

Store Byte

STB *acs,acd*

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

Moves the rightmost byte of ACD to a byte in memory. ACS contains the byte pointer.

Places bits 24–31 of ACD in the byte addressed by the byte pointer contained in bits 16-31 of ACS.

The 32-bit effective address generated by this instruction is constrained to be within the first 64 Kbyte of the current segment.

The contents of ACS, ACD, and carry remain unchanged. *Overflow* is 0.

Store Integer

STI *fpac*

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

Under the control of accumulators AC1 and AC3, translates the contents of the specified FPAC to an integer of the specified type and stores it, right-justified, in memory, beginning at the specified location. The instruction leaves the floating point number unchanged in the FPAC, and destroys the previous contents of memory at the specified location(s).
Bits 16-31 of AC1 must contain the data-type indicator describing the integer.

Bits 16-31 of AC3 must contain a byte pointer which is the address of the high-order byte of the number in memory.

Upon successful completion, the instruction leaves accumulators AC0 and AC1 unchanged. AC2 contains the original contents of AC3 and AC3 contains a byte pointer which is the address of the next byte after the destination field. *Overflow* is 0.

The 32-bit effective address generated by this instruction is constrained to be within the first 64 Kbyte of the current segment.

> NOTES: *If the number in the specified FPAC has any fractional part, the result of the instruction is undefined. Use the* Integerize *instruction to clear any fractional part.*
>
> *If the destination field cannot contain the entire number being stored, high-order digits are discarded until the number will fit into the destination. The remaining low-order digits are stored and carry is set to 1.*
>
> *For data types 0, 1, 2, 3, 4, and 5, if the number being stored will not fill the destination field, the high-order bytes to the right of the sign are set to 0.*
>
> *For data type 6, if the number being stored will not fill the destination field, the sign bit is extended to the left to fill the field.*
>
> *For data type 7, if the number being stored will not fill the destination field, the low-order bytes are set to 0.*

Store Integer Extended

STIX

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

Converts the contents of the four FPAC's to integer form and uses the low-order 8 digits of each to form a 32-digit integer. The instruction stores this integer, right-justified, in

memory beginning at the specified location. The sign of the integer is the logical OR of the signs of all four FPAC's. The previous contents of the addressed memory locations are lost. Sets carry to 0. The contents of the FPAC's remain unchanged. The condition codes in the FPSR are unpredictable.

Bits 16-31 of AC1 must contain the data-type indicator describing the form of the in memory.

Bits 16-31 of AC3 must contain a byte pointer which is the address of the high-order byte of the destination field in memory.

Upon successful termination, the contents of AC0 are undefined; the contents of AC1 remain unchanged; AC2 contains the original contents of AC3; and AC3 contains a byte pointer which is the address of the next byte after the destination field. *Overflow* is 0.

The 32-bit effective address generated by this instruction is constrained to be within the first 64 Kbyte of the current segment.

NOTES: *If the destination field is not large enough to contain the number being stored, the instruction disregards high-order digits until the number will fit in the destination. The instruction stores low-order digits remaining and sets carry to 1.*

For data types 0, 1, 2, 3, 4, and 5, if the number being stored will not fill the destination field, the instruction sets the high-order bytes to 0.

For data type 6, if the number being stored will not fill the destination field, the instruction extends the sign bit to the left to fill the field.

Subtract

SUB*[c][sh][#]* *acs,acd[,skip]*

1	ACS	ACD	1	0	1	SH	C	#	SKIP
0	1 2	3 4	5	6	7	8 9	10 11	12	13 15

Performs unsigned integer subtraction and complements carry if appropriate.

Initializes carry to its specified value. The instruction subtracts the unsigned, 16-bit number in bits 16-31 of ACS from the unsigned, 16-bit number in bits 16-31 of ACD by taking the two's complement of the number in ACS and adding it to the number in ACD. The instruction places the result of the addition in the shifter. If the operation produces a carry of 1 out of the high-order bit, the instruction complements carry. The instruction performs the specified shift operation and places the result of the shift in bits 16-31 of ACD if the no-load bit is 0. If the skip condition is true, the instruction skips the next sequential word.

If the load option is specified, bits 0-15 of ACD are undefined.

Overflow is 0 for this instruction.

NOTE: *If the number in ACS is less than or equal to the number in ACD, the instruction complements carry.*

System Call

SYC *acs,acd*

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

Pushes a return block and transfers control to the *system call handler*.

If a user map is enabled, the instruction disables it and pushes a return block onto the stack. The program counter in the return block points to the instruction immediately following the *System Call* instruction. After pushing the return block, the instruction executes a *Jump Indirect* to location 2, which contains the address of the *system call handler.*

If this instruction disables a user map, then I/O interrupts cannot occur between the time the *System Call* instruction is executed and the time the first instruction of the system call handler is executed.

If the ATU is enabled, a privileged instruction protection fault occurs.

This instruction leaves carry unchanged; *overflow* is 0.

> NOTES: *If both accumulators are specified as AC0, the instruction does not push a return block onto the stack. The contents of AC0 remain unchanged.*
>
> *The assembler recognizes the mnemonic* SCL *as equivalent to* SYC 1,1.
>
> *The assembler recognizes the mnemonic* SVC *as equivalent to* SYC 0,0.

Skip On Zero Bit

SZB *acs,acd*

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

The two accumulators form a bit pointer. If the addressed bit is zero, the next sequential word is skipped.

Forms a 32-bit bit pointer from the contents of bits 16-31 of both ACS and ACD. Bits 16-31 of ACS contains the high-order 16 bits and bits 16-31 of ACD contains the low-order 16 bits of the bit pointer. If ACS and ACD are specified as the same accumulator, the instruction treats the accumulator contents as the low-order 16 bits of the bit pointer and assumes the high-order 16 bits are 0.

If the addressed bit in memory is 0, the next sequential word is skipped. The contents of ACS and ACD remain unchanged.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

This instruction leaves carry unchanged; *overflow* is 0.

> NOTE: *The bit pointer contained in ACS and ACD cannot make indirect memory references.*

Skip On Zero Bit And Set To One

SZBO *acs,acd*

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

The two accumulators form a bit pointer. The instruction sets the addressed bit to 1. If the addressed bit was 0 before being set to 1, the instruction skips the next sequential word. The contents of ACS, ACD, and carry remain unchanged. *Overflow* is 0.

Forms a 32-bit bit pointer from the contents of bits 16-31 of ACS and ACD. Bits 16-31 of ACS contains the high-order 16 bits and bits 16-31 of ACD contains the low-order 16 bits of the bit pointer. If ACS and ACD are specified as the same accumulator, the instruction treats the accumulator contents as the low-order 16 bits of the bit pointer and assumes the high-order 16 bits are 0.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

NOTES: *The bit pointer contained in ACS and ACD must not make indirect memory references*

This instruction facilitates the use of bit maps for such purposes as allocation of facilities (memory blocks, I/O devices, etc.) to several processes, or tasks, that may interrupt one another, or in a multiprocessor environment. The bit is tested and set to 1 in one memory cycle.

Skip on Valid Byte Pointer

VBP

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

Checks a byte pointer for valid reference, and skips or does not skip the next word depending on the outcome of the check. Carry is unchanged and *overflow* is 0.

AC1 contains a ring number in bits 1–3; all other bits contain zeroes.

AC0 contains a 32-bit byte pointer.

The instruction compares the ring field of AC1 to the ring field of AC0. If AC1's ring field is greater than AC0's ring field, the next sequential word is executed; otherwise, the next sequential word is skipped.

Skip on Valid Word Pointer

VWP

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

Checks a word pointer for valid reference, and skips or does not skip the next word depending on the outcome of the check.

AC1 contains a ring number in bits 1–3; all other bits contain zeroes.

AC0 contains a 31-bit word pointer (indirectable).

The instruction compares the ring field of AC1 to the ring field of AC0. If AC1's ring field is greater than AC0's ring field, the next sequential word is executed; otherwise, the next sequential word is skipped. Carry is unchanged and *overflow* is 0.

Wide Add Complement

WADC *acs,acd*

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

Forms the logical complement of the 32-bit integer contained in ACS and adds it to the 32-bit integer contained in ACD. Stores the result in ACD. Sets carry to the value of ALU carry. Sets *overflow* to 1 if there is an ALU overflow.

Wide Add

WADD *acs,acd*

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

Adds the 32-bit fixed point integer contained in ACS to the 32-bit fixed point integer contained in ACD. Stores the result in ACD. Sets carry to ALU carry. Sets *overflow* to 1 if there is an ALU overflow.

Wide Add With Wide Immediate

WADDI *ac,immediate*

1	0	0	AC	1	1	0	1	0	0	0	1	0	0	1	IMMEDIATE
0	1	2	3 4	5	6	7	8	9	10	11	12	13	14	15 16	47

Adds the 32-bit fixed point integer contained in the immediate field to the 32-bit fixed point integer contained in the specified accumulator. Stores the result in the specified accumulator. Sets *overflow* to 1 if there is an ALU overflow. Sets carry to the value of the ALU carry.

Wide Add Immediate

WADI *n,ac*

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

Adds the value $n+1$ to the 32-bit fixed point integer contained in the specified accumulator. Stores the result in the specified accumulator. Sets carry to the value of ALU carry. Sets *overflow* to 1 if there is an ALU overflow.

NOTE: *The assembler takes the coded value of* n *and subtracts one from it before placing it in the immediate field. Therefore, the programmer should code the exact value that he wishes to add.*

Wide AND with Complemented Source

WANC *acs,acd*

1	ACS	ACD	1	0	1	0	1	0	0	1	0	0	1
0	1 2	3 4	5	5	7	8	9	10	11	12	13	14	15

Forms the one's complement of the 32 bits contained in ACS and logically ANDs it with the 32 bits contained in ACD. Stores the result in ACD. Carry is unchanged and *overflow* is 0.

Wide AND

WAND *acs,acd*

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

Forms the logical AND between corresponding bits of ACS and ACD. Loads the 32-bit result into ACD. The contents of ACS remain unchanged. Carry is unchanged and *overflow* is 0.

Wide AND Immediate

WANDI *ac, immediate*

1	0	0	AC	1	1	0	1	0	0	1	1	0	0	1	IMMEDIATE
0	1	2	3 4	5	6	7	8	9	10	11	12	13	14	15 16	47

Forms the logical AND between corresponding bits of the specified accumulator and the value contained in the literal field. The instruction places the 32-bit result of the logical AND in the specified accumulator. Carry is unchanged and *overflow* is 0.

Wide Arithmetic Shift

WASH *acs,acd*

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

Shifts the contents of ACD left or right.

Bits 24–31 of ACS specify the number of bits to shift and the direction of shifting.

If ACS contains a positive number, the instruction shifts the contents of ACD left; zeroes fill the vacated bit positions. If ACS contains a negative number, the instruction shifts the contents of ACD right; the sign bit fills the vacated bit positions. If ACS contains zero, no shifting occurs. The instruction ignores bits 0–23 of ACS.

If the instruction is to shift the contents of ACD to the right, it truncates the contents one bit position for each shift.

In shifting negative numbers to the right, rounding towards zero is performed. For instance, -3 shifted one position to the right results in -1.

The value of ACS and carry remain unchanged. If, while performing a left shift, you shift out a bit whose value is the complement of ACD's sign bit, *overflow* is set to 1. Otherwise, *overflow* is 0.

Wide Block Move

WBLM

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

Moves words sequentially from one memory location to another, treating them as unsigned, 32-bit integers.

AC1 contains the two's complement of the number of words to be moved. If the contents of AC1 are positive, then data movement progresses from the lowest memory location to the highest (ascending). If the contents of AC1 are negative, then data movement progresses from the highest memory location to the lowest (descending).

Bits 1–31 of AC2 contain the address of the source location. Bits 1–31 of AC3 contain the address of the destination location. The address in bits 1–31 of AC2 or AC3 is an indirect address if bit 0 of that accumulator is 1. In that case, the instruction follows the indirection chain before placing the resultant effective address in the accumulator.

AC	Contents
0	Unused
1	Number of words to be moved
2	Source address
3	Destination address

For each word moved, the instruction decrements the count in AC1 by 1. If data movement is ascending, the instruction increments the source and destination addresses by 1 for each word moved. If data movement is descending, the instruction decrements the source and destination addresses by 1 for each word moved.

Upon completion of the instruction, AC1 contains zeroes, and AC2 and AC3 point to the word following (ascending) or preceding (descending) the last word in their respective fields. AC0 is unused. Carry is unchanged and *overflow* is 0.

> NOTES: *Since this instruction may require a long time to execute, it is interruptable. When this instruction is interrupted, the processor saves the address of the* WBLM *instruction. This instruction updates addresses and word count after storing each word, so any interrupt service routine returning control via the saved address will correctly restart the* WBLM *instruction.*
>
> *If data movement is descending and a ring crossing would occur, a protection trap occurs and this instruction does not execute. AC1 will contain the value 4.*

When updating the source and destination addresses, the *Wide Block Move* instruction forces bit 0 of the result to 0. This ensures that upon return from an interrupt, the *Wide Block Move* instruction will not try to resolve an indirect address in either AC2 or AC3.

Load PC

WBR *displacement*

1	DISP 0-3	0	DISP 4-7	1	1	1	0	0	0
0	1 – 4	5	6 – 9	10	11	12	13	14	15

Adds the 31-bit value contained in the PC to the value of the displacement and places the result in the PC. Carry is unchanged and *overflow* is 0.

> NOTE: *The processor always forces the value loaded into the PC to reference a location in the current segment of execution.*

Wide Search Queue Backward

WBSAC

1	1	0	0	0	1	1	1	0	0	0	1	1	0	0	1	RESERVED	1	0	1	1	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16 – 26	27	28	29	30	31

See instruction entry "Search Queue".

Wide Search Queue Backward

WBSAS

1	1	0	0	0	1	1	1	0	0	0	1	1	0	0	1	RESERVED	1	0	1	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16 – 26	27	28	29	30	31

See instruction entry "Search Queue".

Wide Search Queue Backward

WBSE

1	1	0	0	0	1	1	1	0	0	0	1	1	0	0	1	RESERVED	1	1	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16 – 26	27	28	29	30	31

See instruction entry "Search Queue".

Wide Search Queue Backward

WBSGE

1	1	0	0	0	1	1	1	0	0	0	1	1	0	0	1	RESERVED	1	1	0	1	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16 … 26	27	28	29	30	31

See instruction entry "Search Queue".

Wide Search Queue Backward

WBSLE

1	1	0	0	0	1	1	1	0	0	0	1	1	0	0	1	RESERVED	1	1	1	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16 … 26	27	28	29	30	31

See instruction entry "Search Queue".

Wide Search Queue Backward

WBSNE

1	1	0	0	0	1	1	1	0	0	0	1	1	0	0	1	RESERVED	1	1	1	1	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16 … 26	27	28	29	30	31

See instruction entry "Search Queue".

Wide Search Queue Backward

WBSSC

1	1	0	0	0	1	1	1	0	0	0	1	1	0	0	1	RESERVED	1	0	0	1	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16 … 26	27	28	29	30	31

See instruction entry "Search Queue".

Wide Search Queue Backward

WBSSS

1	1	0	0	0	1	1	1	0	0	0	1	1	0	0	1	RESERVED	1	0	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16 … 26	27	28	29	30	31

See instruction entry "Search Queue".

Wide Set Bit to One

WBTO *acs,acd*

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

Sets the specified bit to one. Carry is unchanged and *overflow* is 0.

ACS contains a 31-bit word address.

ACD contains a bit offset.

The instruction sets the bit specified by ACS and ACD to one. The contents of ACS and ACD remain unchanged.

If ACS and ACD are specified to be the same accumulator, then the processor assumes the word address is zero within the current segment. In this case, the specified accumulator contains a 32-bit bit pointer.

Wide Set Bit to Zero

WBTZ *acs,acd*

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

Sets the specified bit to zero. Carry is unchanged and *overflow* is 0.

ACS contains a 31-bit word address.

ACD contains a bit offset.

The instruction sets the bit specified by ACS and ACD to zero. The contents of ACS and ACD remain unchanged.

If ACS and ACD are specified to be the same accumulator, then the processor assumes the word address is zero within the current segment. In this case, the specified accumulator contains a 32-bit bit pointer.

Wide Compare to Limits

WCLM *acs,acd*

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

Compares a signed integer with two limit values and skips if the integer is between the limit values. The accumulators determine the location of the limit values. Carry is unchanged and *overflow* is 0.

Compares the signed, two's complement integer in ACS to two signed, two's complement integer limit values, *L* and *H*. If the number in ACS is greater than or equal to *L* and less than or equal to *H*, execution skips the next sequential word before continuing. If the number in ACS is less than *L* or greater than *H*, execution continues with the next sequential word.

If ACS and ACD are specified as different accumulators, bits 1–31 of ACD contain the address of the limit value *L*. The word following *L* contains the limit value *H*. Bit 0 of ACD is ignored.

If ACS and ACD are specified as the same accumulator, the integer to be compared must be in that accumulator and the limit values *L* and *H* must be in the two words following the instruction. The first word contains *L*, and the second contains *H*. The third word contains the next sequential word of the program.

Wide Character Compare

WCMP

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

Under control of the four accumulators, compares two strings of bytes and returns a code in AC1 reflecting the results of the comparison.

The instruction compares the strings one byte at a time. Each byte is treated as an unsigned 8-bit binary quantity in the range $0–255_{10}$. If two bytes are not equal, the string whose byte has the smaller numerical value is, by definition, the *lower valued* string. Both strings remain unchanged. The four accumulators contain parameters passed to the instruction. Two accumulators specify the starting address, the number of bytes, and the direction of processing (ascending or descending addresses) for each string.

AC0 specifies the length and direction of comparison for string 2. If the string is to be compared from its lowest memory location to the highest, AC0 contains the unsigned value of the number of bytes in string 2. If the string is to be compared from its highest memory location to the lowest, AC0 contains the two's complement of the number of bytes in string 2.

AC1 specifies the length and direction of comparison for string 1. If the string is to be compared from its lowest memory location to the highest, AC0 contains the unsigned value of the number of bytes in string 1. If the string is to be compared from its highest memory location to the lowest, AC1 contains the two's complement of the number of bytes in string 1.

AC2 contains a byte pointer to the first byte compared in string 2. When the string is compared in ascending order, AC2 points to the lowest byte. When the string is compared in descending order, AC2 points to the highest byte.

AC3 contains a byte pointer to the first byte compared in string 1. When the string is compared in ascending order, AC3 points to the lowest byte. When the string is compared in descending order, AC3 points to the highest byte.

Code	Comparison Result
- 1	String 1 < String 2
0	String 1 = String 2
+ 1	String 1 > String 2

The strings may overlap in any way. Overlap will not effect the results of the comparison.

Upon completion, AC0 contains the number of bytes left to compare in string 2. AC1 contains the return code as shown in the table above. AC2 contains a byte pointer either to the failing byte in string 2 (if an inequality was found), or to the byte following string 2 (if string 2 was exhausted). AC3 contains a byte pointer either to the failing byte in string 1 (if an inequality was found), or to the byte following string 1 (if string 1 was exhausted). Carry is unchanged and *overflow* is 0.
If AC0 and AC1 both contain zero (both string 1 and string 2 have length zero), the instruction returns 0 in AC1.

If the two strings are of unequal length, the instruction *fakes* space characters $<040_8>$ in place of bytes from the exhausted string, and continues the comparison.

> NOTE: *The original contents of AC2 and AC3 must be valid byte pointers to an area in the user's address space. If they are invalid a protection fault occurs, even if no bytes are to be compared. AC1 contains the code 4.*

Wide Character Move Until True
WCMT

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

Under control of the four accumulators, moves a string of bytes from one area of memory to another until either a table-specified delimiter character is moved or the source string is exhausted.

The instruction copies the string one byte at a time. Before it moves a byte, the instruction uses that byte's value to determine if it is a delimiter. It treats the byte as an unsigned 8-bit binary integer (in the range $0–255_{10}$) and uses it as a bit index into a 256-bit delimiter table. If the indexed bit in the delimiter table is zero, the byte pending is not a delimiter, and the instruction copies it from the source string to the destination string. If the indexed bit in the delimiter table is 1, the byte pending is a delimiter; the instruction does not copy it, and the instruction terminates.

The instruction processes both strings in the same direction, either from lowest memory locations to highest (*ascending order*), or from highest memory locations to lowest (*descending order*). Processing continues until there is a delimiter or the source string is exhausted. The four accumulators contain parameters passed to the instruction.

AC0 contains the address (word address), possibly indirect, of the start of the 256-bit (16-word) delimiter table.

AC1 specifies the length of the strings and the direction of processing. If the source string is to be moved to the destination field in ascending order, AC1 contains the unsigned value of the number of bytes in the source string. If the source string is to be moved to the destination field in descending order, AC1 contains the two's complement of the number of bytes in the source string.

AC2 contains a byte pointer to the first byte to be written in the destination field. When the process is performed in ascending order, AC2 points to the lowest byte in the destination field. When the process is performed in descending order, AC2 points to the highest byte in the destination field.

AC3 contains a byte pointer to the first byte to be processed in the source string. When the process is performed in ascending order, AC3 points to the lowest byte in the source string. When the process is performed in descending order, AC3 points to the highest byte in the source string.
The fields may overlap in any way. However, the instruction moves bytes one at a time, so certain types of overlap may produce unusual side effects.

Upon completion, AC0 contains the resolved address of the translation table and AC1 contains the number of bytes that were not moved. AC2 contains a byte pointer to the byte following the last byte written in the destination field. AC3 contains a byte pointer either to the delimiter or to the first byte following the source string. The value of carry is indeterminate and *overflow* is 0.

> **NOTE:** *The original contents of AC0, AC2, and AC3 must be valid byte pointers to an area in the user's address space. If they are invalid a protection fault occurs, even if no bytes are to be stored. AC1 contains the code 4.*

Wide Character Move
WCMV

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

Under control of the four 32-bit accumulators, moves a string of bytes from one area of memory to another and returns a value in carry reflecting the relative lengths of source and destination strings.

The instruction copies the source string to the destination field, one byte at a time. The four accumulators contain parameters passed to the instruction. Two accumulators specify the starting address, number of bytes to be copied, and the direction of processing (ascending or descending addresses) for each field.

AC0 specifies the length and direction of processing for the destination field. If the field is to be processed from its lowest memory location to the highest, AC0 contains the unsigned value of the number of bytes in the destination field. If the field is to be processed from its highest memory location to the lowest, AC0 contains the two's complement of the number of bytes in the destination field.

AC1 specifies the length and direction of processing for the source string. If the string is to be processed from its lowest memory location to the highest, AC1 contains the unsigned value of the number of bytes in the source string. If the field is to be processed from its highest memory location to the lowest, AC1 contains the two's complement of the number of bytes in the source string.

AC2 contains a byte pointer to the first byte to be written in the destination field. When the field is written in ascending order, AC2 points to the lowest byte. When the field is written in descending order, AC2 points to the highest byte.

AC3 contains a byte pointer to the first byte copied in the source string. When the field is copied in ascending order, AC3 points to the lowest byte. When the field is copied in descending order, AC3 points to the highest byte.

The fields may overlap in any way. However, the instruction moves bytes one at a time, so certain types of overlap may produce unusual side effects.

Upon completion, AC0 contains 0 and AC1 contains the number of bytes left to fetch from the source field. AC2 contains a byte pointer to the byte following the destination field; and AC3 contains a byte pointer to the byte following the last byte fetched from the source field. The value of carry is indeterminate and *overflow* is 0.
If the source field is shorter than the destination field, the instruction pads the destination field with space characters $<040_8>$. If the source field is longer than the destination field, the instruction terminates when the destination field is filled and returns the value 1 in carry; otherwise, the instruction returns the value 0 in carry.

NOTES: *If AC0 contains the number 0 at the beginning of this instruction, no bytes are fetched and none are stored. If AC1 is 0 at the beginning of this instruction, the destination field is filled with space characters; note that AC3 must still contain a valid byte pointer.*

The original values of AC2 and AC3 must be valid byte pointers to an area in the user's address space. If they are invalid a protection fault occurs, even if no bytes are to be moved. AC1 contains the code 4.

Wide Count Bits

WCOB *acs,acd*

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

Counts the number of bits in ACS whose value is 1. Adds the count of non-zero bits to the 32-bit, signed contents of ACD. The contents of ACS remain unchanged, unless ACS and ACD are the same accumulator. Carry is unchanged and *overflow* is 0.

Wide Complement

WCOM *acs,acd*

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

Forms the one's complement of the 32-bit fixed point integer contained in ACS and loads the result into ACD. The contents of ACS remain unchanged, unless ACS equals ACD. Carry is unchanged and *overflow* is 0.

Wide Character Translate

WCTR

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

Under control of the four accumulators, translates a string of bytes from one data representation to another and either moves it to another area of memory or compares it to a second translated string.

The instruction operates in two modes: translate and move, and translate and compare.

When operating in translate and move mode, the instruction translates each byte in string 1 and places it in a corresponding position in string 2. Translation is performed by using each byte as an 8-bit index into a 256-byte translation table. The byte addressed by the index then becomes the translated value.
When operating in translate and compare mode, the instruction translates each byte in string 1 and string 2 as described above, and compares the translated values. Each translated byte is treated as an unsigned 8-bit binary quantity in the range $0–255_{10}$. If two translated bytes are not equal, the string whose byte has the smaller numerical value is, by definition the *lower valued* string. Both strings remain unchanged.

AC0 specifies the address, either direct or indirect, of a word which contains a byte pointer to the first byte in the 256-byte translation table.

AC1 specifies the length of the two strings and the mode of processing. If string 1 is to be processed in translate and move mode, AC1 contains the two's complement of the number of bytes in the strings. If the strings are to be processed in translate and compare mode, AC1 contains the unsigned value of the number of bytes in the strings. Both strings are processed from lowest memory address to highest.

AC2 contains a 32-bit byte pointer to the first byte in string 2.

AC3 contains a 32-bit byte pointer to the first byte in string 1.

Upon completion of a translate and move operation, AC0 contains the address of the word which contains the byte pointer to the translation table and AC1 contains 0. AC2 contains a byte pointer to the byte following string 2 and AC3 contains a byte pointer to the byte following string 1. The value of carry is unchanged and *overflow* is 0.

Upon completion of a translate and compare operation, AC0 contains the address of the word which contains the byte pointer to the translation table. AC1 contains a return code as calculated in the table below. AC2 contains a byte pointer to either the failing byte in string 2 (if an inequality was found) or the byte following string 2 if the strings were identical. AC3 contains a byte pointer to either the failing byte in string 1 (if an inequality was found) or the byte following string 1 if the strings were identical. The value of carry is unchanged and *overflow* is 0.

Code	Result
-1	Translated value of string 1 < Translated value of string 2
0	Translated value of string 1 = Translated value of string 2
+1	Translated value of string 1 > Translated value of string 2

If the length of both string 1 and string 2 is zero, the compare option returns a 0 in AC1.

The fields may overlap in any way. However, processing is done one character at a time, so unusual side effects may be produced by certain types of overlap.

NOTE: *The original contents of AC0, AC2, and AC3 must be valid byte pointers to an area in the user's address space. If they are invalid a protection fault occurs, even if no bytes are to be moved or compared. AC1 contains the code 4.*

Wide Divide

WDIV *acs,acd*

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

Divides an integer contained in an accumulator by an integer contained in another accumulator.

The instruction sign extends the signed, 32-bit integer contained in ACD to 64 bits. Divides this integer by the signed, 32-bit integer contained in ACS. If the quotient is within the range -2,147,483,648 to +2,147,483,647 inclusive, loads the quotient in ACD. If the result is not within this range, or if ACS is zero, sets *overflow* to 1 and does not load the quotient into ACD; otherwise, *overflow* is 0. The contents of ACS and carry remain unchanged.

Wide Signed Divide

WDIVS

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

Divides an integer contained in AC0 and AC1 by an integer contained in AC2.

AC0 and AC1 contain a 64-bit, signed integer. AC0 contains the high order bits.

The instruction divides the 64-bit, signed integer contained in AC0 and AC1 by the 32-bit, signed integer contained in AC2. If the quotient is within the range -2,147,483,648 to +2,147,483,647 inclusive, then places the 32-bit quotient in AC1 and the remainder in AC0. If the quotient is not within this range, or AC2 is zero, AC0 and AC1 remain unchanged and *overflow* is 1; otherwise, *overflow* is 0. AC2 and carry will always remain unchanged.

NOTE: *Zero remainders are always positive. All other remainders have the same sign as the dividend.*

Pop MV/8000 Context Block

WDPOP

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

Restores the state of the machine to what it was at the time of the last page fault.

The instruction uses the information pointed to by page zero locations $32–33_8$ of Segment 0 to restore the state of the CPU to that of the time of the last page fault. Execution of the interrupted program resumes before, during, or after the instruction that caused the fault, depending on the instruction type and how far it had proceeded before the fault. Carry is unchanged and *overflow* is 0.

NOTE: *This is a privileged instruction.*

Wide Edit
WEDIT

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

Converts a decimal source number from either packed or unpacked form to a string of bytes under the control of an edit subprogram. This subprogram can perform many different operations on the number and its destination field including leading zero suppression, leading or trailing signs, floating fill characters, punctuation control, and insertion of text into the destination field. The instruction also performs operations on alphanumeric data if you specify data type 4.

Upon entry to the *Edit* instruction, the accumulators contain the following data:

- AC0 contains a 32-bit byte pointer to the first opcode of the *Edit* subprogram,
- AC1 contains a data-type indicator describing the number to be processed,
- AC2 contains a 32-bit byte pointer to the first byte of the destination field,
- AC3 contains a 32-bit byte pointer to the first byte of the source field.

The fields may overlap in any way. However, the instruction processes characters one at a time, so unusual side effects may be produced by certain types of overlap.

The instruction maintains two flags and three indicators or pointers. The flags are the Significance Trigger (*T*) and the Sign flag (*S*). The three indicators are the Source Indicator (SI), the Destination Indicator (DI), and the op-code Pointer (P).

At the start of execution, the *Edit* instruction sets *T* to 0. When the instruction manipulates the first non-zero digit, it sets *T* to 1 (unless an edit op-code specifies otherwise).

The instruction sets *S* to reflect the sign of the number currently being processed. If the number is positive, the instruction sets *S* to 0. If the number is negative, the instruction sets *S* to 1.

Each of the three indicators is 16 bits wide and contains a byte pointer to the *current* byte in each respective area. At the start of execution, the *Edit* instruction sets SI to the value contained in AC3 (the starting address of the source string). It also sets DI to the value contained in AC2 (the starting address of the destination string), and P to the value contained in AC0 (a pointer to the first *Edit* opcode).

During execution, the subprogram can test and modify *S* and *T*, as well as modify SI, DI and P.

When execution begins, the instruction checks the sign of the source number for validity. If the sign is invalid, the instruction ends. If the sign is valid, execution continues with the *Edit* sub-program.

The sub-program is made up of 8-bit op-codes followed by one or more 8-bit operands. The byte pointer contained in P acts as the *program counter* for the subprogram. The subprogram proceeds sequentially until a branching operation occurs — much the same way programs are processed. Unless instructed to do otherwise, the *Edit* instruction updates P after each operation to point to the next sequential op-code. The instruction continues to process 8-bit opcodes until directed to stop by the DEND op-code. Note that all 8-bit opcodes must be contained in the current segment.

Upon successful termination, carry contains *T*; AC0 contains P, which points to the next opcode to be processed; AC1 is undefined; AC2 contains DI, which points to the next destination byte; and AC3 contains SI, which points to the next source byte. The value of carry is indeterminate and *overflow* is 0.

> NOTES: *If SI references bytes not contained in the source number, then the instruction supplies zeroes for future manipulations. The instruction will use these zeroes for all subsequent operations, even if SI later references bytes contained by the source number.*
>
> *Opcodes that move numeric data may perform special actions. Opcodes that move non-numeric data copy characters exactly into the destination string.*
>
> *The* Edit *instruction places information on the wide stack. Therefore, the stack must be set up and have at least 16 words available for use.*
>
> *If an interrupt occurs during the* Edit *instruction, the instruction places restart information on the stack and in the accumulators and sets bit 2 of the PSR to 1.*
>
> *If bit 2 of the PSR contains a 1, then the* Edit *instruction assumes it is restarting from an interrupt. Make sure you do not set this bit under any other circumstances.*

Many of the *Edit* opcodes use the symbol *j*. This symbol represents a number; when *j* is greater than or equal to zero, it specifies the number of characters the instruction should process. When *j* is less than zero, it represents a pointer into the wide stack. The pointer references a stack word that denotes the number of characters the instruction should process. The number on the stack is at address:

WSP + 2 + 2**j*.

An *Edit* operation that processes numeric data (e.g., DMVN) skips a leading or trailing sign code it encounters; similarly, such an operation converts a high-order or low-order sign to its correct numeric equivalent.

Add To DI

DADI *p0*

0	0	0	1	0	0	1	0	p0
0	1	2	3	4	5	6	7	8 … 15

Adds the 8-bit two's complement integer specified by *p0* to the Destination Indicator (DI).

Add To P Depending On S

DAPS *p0*

0	0	0	0	1	1	1	1	p0
0	1	2	3	4	5	6	7	8 … 15

If *S* is 0, the instruction adds the 8-bit two's complement integer specified by *p0* to the op-code Pointer (P). Before the add is performed, P is pointing to the byte containing the DAPS op-code.

Add To P Depending On T

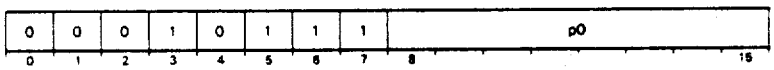
DAPT *p0*

0	0	0	0	1	0	1	1	p0
0	1	2	3	4	5	6	7	8 … 15

If *T* is one, the instruction adds the 8-bit two's complement integer specified by *p0* to the op-code Pointer (P). Before the add is performed, P is pointing to the byte containing the DAPT op-code.

Add To P

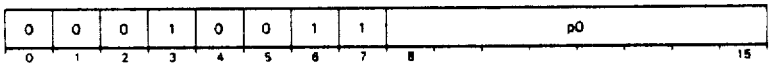
DAPU *p0*



Adds the 8-bit two's complement integer specified by *p0* to the op-code Pointer (P). Before the add is performed, P is pointing to the byte containing the **DAPU** op-code.

Add To SI

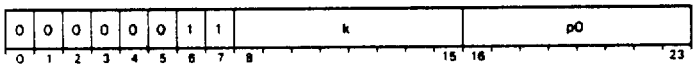
DASI *p0*



Adds the 8-bit two's complement integer specified by *p0* to the Source Indicator (SI).

Decrement And Jump If Non-Zero

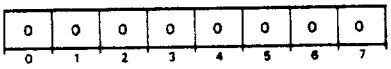
DDTK *k,p0*



Decrements a word in the stack by one. If the decremented value of the word is non-zero, the instruction adds the 8-bit two's complement integer specified by *p0* to the op-code Pointer (P). Before the add is performed, P is pointing to the byte containing the **DDTK** op-code. If the 8-bit two's complement integer specified by *k* is negative, the word decrement is at the address (WSP + 2 + (2*)*k*). If *k* is positive, the word decremented is at the address (WFP + 2 + (2*)*k*).

End Edit

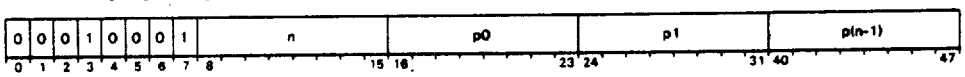
DEND



Terminates the EDIT sub-program.

Insert Characters Immediate

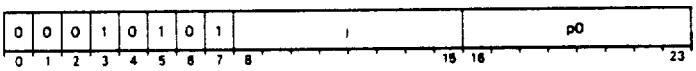
DICI *n,p0,p1,...,p(n-1)*



Inserts *n* characters from the op-code stream into the destination field beginning at the position specified by DI. Increases P by (*n*+2), and increases DI by *n*.

Insert Character J Times

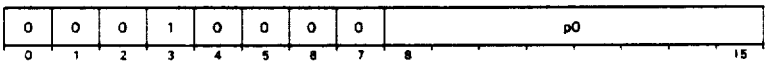
DIMC *j,p0*



Inserts the character specified by *p0* into the destination field a number of times equal to *j* beginning at the position specified by DI. Increases DI by *j*.

Insert Character Once

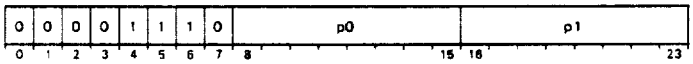
DINC *p0*



Inserts the character specified by *p0* in the destination field at the position specified by DI. Increments DI by 1.

Insert Sign

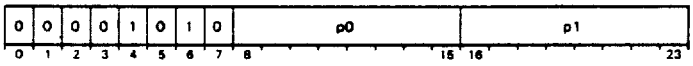
DINS *p0,p1*



If the Sign flag (S) is 0, the instruction inserts the character specified by *p0* in the destination field at the position specified by DI. If *S* is 1, the instruction inserts the character specified by *p1* in the destination field at the position specified by DI. Increments DI by 1.

Insert Character Suppress

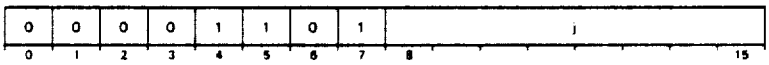
DINT *p0,p1*



If the significance Trigger *(T)* is 0, the instruction inserts the character specified by *p0* in the destination field at the position specified by DI. If *T* is 1, the instruction inserts the character specified by *p1* in the destination field at the position specified by DI. Increments DI by 1.

Move Alphabetics

DMVA *j*

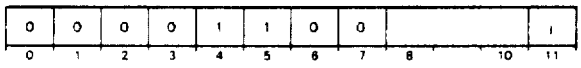


Moves *j* characters from the source field beginning at the position specified by SI to the destination field beginning at the position specified by DI. Increases both SI and DI by *j*. Sets *T* to 1.

Initiates a commercial fault if the attribute specifier word indicates that the source field is data type 5 (packed). Initiates a commercial fault if any of the characters moved is not an alphabetic (A-Z, a-z, or space).

Move Characters

DMVC *j*



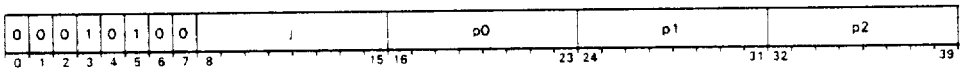
Increments SI if the source data type is 3 and $j>0$. The instruction then moves *j*

characters from the source field beginning at the position specified by SI to the destination field beginning at the position specified by DI. Increases both SI and DI by *j*. Sets *T* to 1.

Initiates a commercial fault if the attribute specifier word indicates that the source is data type 5 (packed). Performs no validation of the characters.

Move Float

DMVF *j,p0,p1,p2*



If the source data type is 3, $j>0$, and SI points to the sign of the source number, the instruction increments SI. Then for *j* characters, the instruction either places a digit substitute in the destination field beginning at the position specified by DI, or it moves a digit from the source field beginning at the position specified by SI to the destination field beginning at the position specified by DI. When *T* changes from 0 to 1, the instruction places both the digit substitute and the digit in the destination field, and compares *j* to the number of digits left to move. Increments SI by the smaller of the two values.

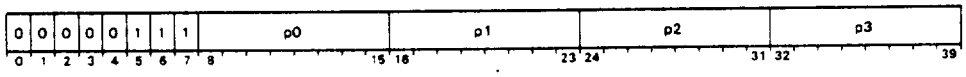
Move Numerics

j\8") Increments SI if the source data type is 3 and $j>0$. Moves *j* characters from the source field beginning at the position specified by SI to the destination field beginning at the position specified by DI. Increases DI by *j*. Compares *j* to the number of source characters left to move, and increments SI by the smaller of the two values. Sets *T* to 1.

Initiates a commercial fault if any of the characters moved is not valid for the specified data type.

Move Digit With Overpunch

DMVO *p0,p1,p2,p3*



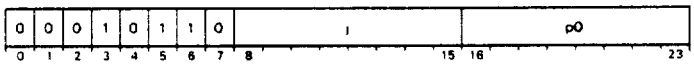
Increments SI if the source data type is 3 and SI points to the sign of the source number. The instruction then either places a digit substitute in the destination field at the position specified by DI, or it moves a digit plus overpunch the source field at the position specified by SI to the destination field at the position specified by DI. Increases DI by 1. Compares the number of digits left to move with 1 and increments SI by the smaller of the two values.

If the digit is a zero or space and *S* is 0, then the instruction places *p0* in the destination field. If the digit is a zero or space and *S* is 1, then the instruction places *p1* in the destination field. If the digit is a non-zero and *S* is 0, the instruction adds *p2* to the digit and places the result in the destination field. If the digit is a non-zero and *S* is 1, the instruction adds *p3* to the digit and places the result in the destination field. If the digit is a non-zero the instruction sets *T* to 1. The instruction assumes *p2* and *p3* are ASCII characters.

The instruction initiates a commercial fault if the character is not valid for the specified data type.

Move Numeric With Zero Suppression

DMVS *j,p0*

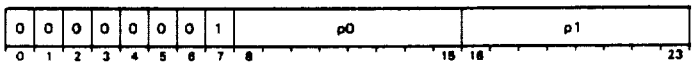


Increments SI if the source data type is 3 and $j>0$, and SI points to the sign of the source number. The instruction then moves *j* characters from the source field beginning at the position specified by SI to the destination field beginning at the position specified by DI. Moves the digit from the source to the destination if *T* is 1. Replaces all zeros and spaces with *p0* as long as *T* is 0. Sets *T* to 1 when the first non-zero digit is encountered. Increases DI by *j*. Compares *j* to the number of source characters left to move, and increments SI by the smaller of the two values.

Initiates a commercial fault if any of the characters moved is not a numeric (0-9 or space).

End Float

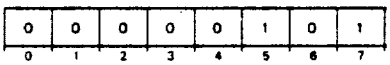
DNDF *p0,p1*



If *T* is 1, the instruction places nothing in the destination field and leaves DI unchanged. If *T* is 0 and *S* is 0, the instruction places *p0* in the destination field at the position specified by DI. If *T* is 0 and *S* is 1, the instruction places *p1* in the destination field at the position specified by DI. Increases DI by 1, and sets *T* to 1.

Set S To One

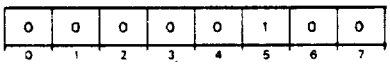
DSSO



Sets the Sign flag *(S)* to 1.

Set S To Zero

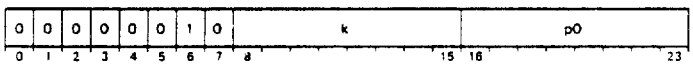
DSSZ



Sets the Sign flag *(S)* to 0.

Store In Stack

DSTK *k,p0*



Stores the byte specified by *p0* in bits 24-31 of a word in the wide stack. Sets bits 0-23 of the word that receives *p0* to 0. If the 8-bit two's complement integer specified by *k* is negative, the instruction addresses the word receiving *p0* by (WSP + 2 + (2*)*k*). If *k* is positive then the instruction stores *p0* at the address (WFP + 2 + (2*)*k*).

Set T To One

DSTO

0	0	0	0	1	0	0	1
0	1	2	3	4	5	6	7

Sets the significance Trigger *(T)* to 1.

Set T To Zero

DSTZ

0	0	0	0	0	1	1	0
0	1	2	3	4	5	6	7

Sets the significance Trigger *(T)* to 0.

Wide Fix from Floating Point Accumulator

WFFAD *ac,fpac*

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

Converts the integer portion of the floating point number contained in the specified FPAC to a 32-bit, signed, two's complement integer. Places the result in an accumulator.

If the integer portion of the number contained in FPAC is less than -2,147,483,649 or greater than +2,147,483,648, the instruction sets *MOF* in the FPSR to 1. Takes the absolute value of the integer portion of the number contained in the FPAC. Takes the 31 least significant bits of the absolute value and appends a 0 onto the leftmost bit to give a 32-bit number. If the sign of the number is negative, formes the two's complement of the 32-bit result. Places the 32-bit integer in the specified accumulator.

If the integer portion is within the range of -2,147,483,648 to +2,147,483,647 inclusive, the instruction places the 32-bit, two's complement of the integer portion of the number contained in the FPAC in the specified accumulator.

The instruction leaves the FPAC and the *Z* and *N* flags of the FPSR unchanged.

Wide Float from Fixed Point Accumulator

WFLAD *ac,fpac*

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

Converts the contents of a 32-bit accumulator to floating point format and places the result in a specified FPAC.
Converts the 32-bit, signed, two's complement number contained in the specified accumulator to a double precision floating point number. Places the result in the specified FPAC. Updates the *Z* and *N* flags in the floating point status register to reflect the new contents of the FPAC.

The range of numbers that can be converted is -2,147,483,648 to +2,147,483,647 inclusive.

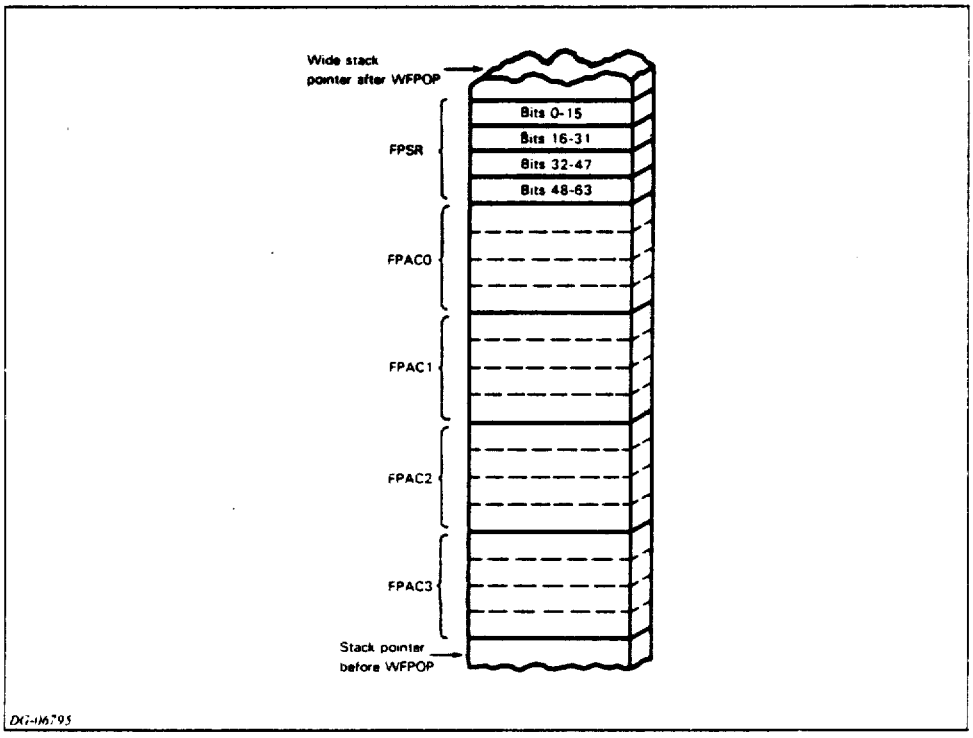
Wide Floating Point Pop

WFPOP

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

Pops the state of the floating point unit off the wide stack.

Pops a 20-word block off the wide stack and loads the contents into the FPSR and the four FPACs. The format of the 20-word block is shown below.



This instruction loads the FPSR as follows:

- Places bits 0-15 of the operand in bits 0-15 of the FPSR. Sets bits 16-32 of the FPSR to 0.
- If *ANY* is 0, bits 33-63 of the FPSR are undefined.
- If *ANY* is 1, the instruction places the value of the current segment in bits 33-35 of the FPSR, zeroes in bits 36-48, and bits 17-31 of the operand in bits 49-63 of the FPSR.

> **NOTES:** *This instruction moves unnormalized data without change.*
>
> *This instruction does not set the* ANY *flag from memory. If any of bits 1-4 are loaded as 1,* ANY *is set to 1; otherwise,* ANY *is 0.*
>
> *Bits 12-15 of the FPSR are not set from memory. These bits are the floating point identification code an are read protected. In the MV/8000 they are set to 0111.*
>
> *This instruction does not initiate a floating point trap under any conditions of the FPSR.*

See Chapter 8 and Appendix G for more information about floating point manipulation.

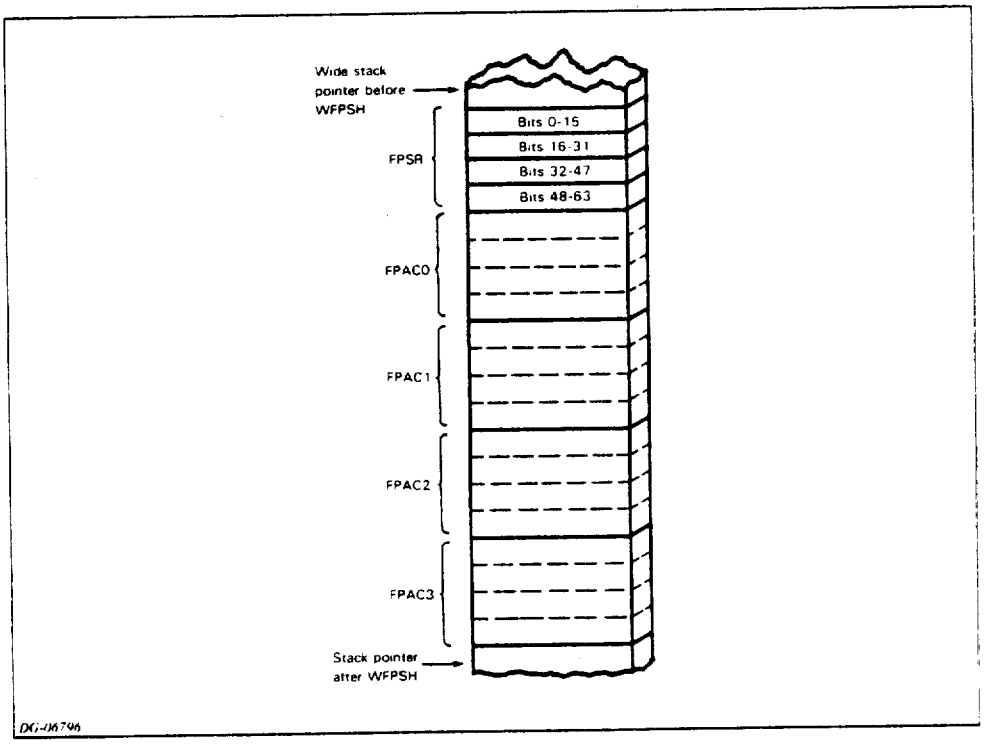
Wide Floating Point Push

WFPSH

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

Pushes the state of the floating point unit onto the wide stack.

Pushes a 20-word block onto the wide stack. The block contains the contents of the FPSR and the contents of the four FPACs, as shown in the figure below:



The instruction pushes the FPSR onto the stack as follows:

- Stores bits 0-15 of the FPSR in the first memory word.
- Sets bits 16-31 of the first memory double word and bit 0 of the second memory double word to 0.
- If *ANY* is 0, the contents of bits 1-31 of the second memory double word are undefined.
- If *ANY* is 1, the instruction stores bits 33-63 of the FPSR into bits 1-31 of the second memory double word.

The rest of the block is pushed onto the stack after the FPSR has been pushed.

NOTES: *This instruction moves unnormalized data without change.*

This instruction does not initiate a floating point trap under any conditions of the FPSR.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Wide Search Queue Forward

WFSAC

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	RESERVED	26	27	28	29	30	31
1	1	0	0	0	1	1	1	0	0	0	1	1	0	0	1				1	0	1	1	0

See instruction entry "Search Queue".

Wide Search Queue Forward

WFSAS

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	RESERVED	26	27	28	29	30	31
1	1	0	0	0	1	1	1	0	0	0	1	1	0	0	1				1	0	1	0	0

See instruction entry "Search Queue".

Wide Search Queue Forward

WFSE

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	RESERVED	26	27	28	29	30	31
1	1	0	0	0	1	1	1	0	0	0	1	1	0	0	1				1	1	0	0	0

See instruction entry "Search Queue".

Wide Search Queue Forward

WFSGE

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	RESERVED	26	27	28	29	30	31
1	1	0	0	0	1	1	1	0	0	0	1	1	0	0	1				1	1	0	1	0

See instruction entry "Search Queue".

Wide Search Queue Forward

WFSLE

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	RESERVED	26	27	28	29	30	31
1	1	0	0	0	1	1	1	0	0	0	1	1	0	0	1				1	1	1	0	0

See instruction entry "Search Queue".

Wide Search Queue Forward

WFSNE

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	RESERVED	26	27	28	29	30	31
1	1	0	0	0	1	1	1	0	0	0	1	1	0	0	1				1	1	1	1	0

See instruction entry "Search Queue".

Wide Search Queue Forward

WFSSC

1	1	0	0	0	1	1	1	0	0	0	1	1	0	0	1	RESERVED	1	0	0	1	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16 … 26	27	28	29	30	31

See instruction entry "Search Queue".

Wide Search Queue Forward

WFSSS

1	1	0	0	0	1	1	1	0	0	0	1	1	0	0	1	RESERVED	1	0	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16 … 26	27	28	29	30	31

See instruction entry "Search Queue".

Wide Halve

WHLV *ac*

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

Divides the 32-bit contents of the specified accumulator by 2 and rounds the result toward 0.

The signed, 32-bit two's complement number contained in the specified accumulators divided by 2 and rounded toward 0. The result is placed in the specified accumulator.

This instruction leaves carry unchanged; *overflow* is 0.

Wide Increment

WINC *acs,acd*

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

Increments an integer contained in an accumulator.

The instruction increments the 32-bit contents of ACS by 1 and loads the result into ACD. Sets carry to the value of ALU carry. Sets *overflow* to 1 if there is an ALU overflow. The contents of ACS remain unchanged, unless ACS equals ACD.

Wide Inclusive OR

WIOR *acs,acd*

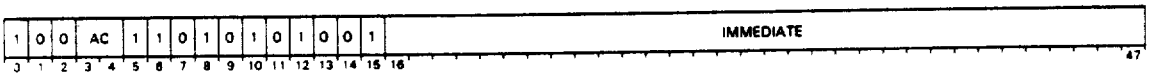
1	ACS	ACD	1	0	0	0	1	1	0	1	0	0	1
0	1 2	3 4	5	6	7	8	9	10	11	12	13	14	15

Performs an inclusive OR between two accumulators.

Forms the logical inclusive OR between corresponding bits of ACS and ACD. Loads the 32-bit result into ACD. The contents of ACS remain unchanged. Carry is unchanged and *overflow* is 0.

Wide Inclusive OR Immediate

WIORI *ac, immediate*

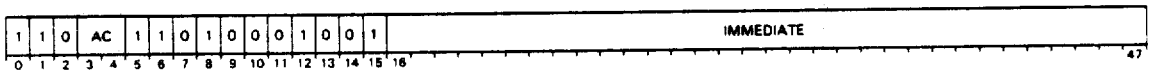


Performs an inclusive OR.

The instruction forms the logical inclusive OR between corresponding bits of the specified accumulator and the value contained in the literal field. The instruction places the result of the inclusive OR in the specified accumulator. Carry is unchanged and *overflow* is 0.

Wide Load with Wide Immediate

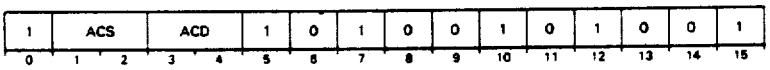
WLDAI *ac,immediate*



Loads the 32-bit value contained in the immediate field into the specified accumulator. Carry is unchanged and *overflow* is 0.

Wide Load Byte

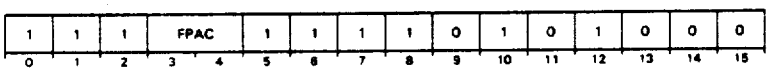
WLDB *acs,acd*



Uses the 32-bit byte address contained in ACS to load a byte into ACD. Sets bits 0–23 of ACD to zero. Bits 24–31 of ACD contain a copy of the contents of the addressed byte. The contents of ACS remain unchanged, unless ACS and ACD are the same accumulator. Carry is unchanged and *overflow* is 0.

Wide Load Integer

WLDI *fpac*



Translates a decimal integer from memory to floating point format and places the result in a floating point accumulator.

AC1 must contain the data-type indicator describing the integer.

AC3 must contain a 32-bit byte pointer pointing to the high-order byte of the integer in memory.

Uses AC1 and AC3 to convert a decimal integer to floating point form. Normalizes the result and places it in the specified FPAC. Updates the *Z* and *N* flags in the FPSR to describe the new contents of the specified FPAC. Leaves the decimal number unchanged in memory.

By convention, the first byte of a number stored according to data type 7 contains the sign and exponent of the floating point number. The instruction copies each byte (following the lead byte) directly to the mantissa of the specified FPAC. It then sets to zero each low-order byte in the FPAC that does not receive data from memory.

Upon successful completion, AC0 and AC1 remain unchanged. AC2 contains the original contents of AC3. AC3 points to the first byte following the integer field. Carry is unchanged and *overflow* is 0.

Wide Load Integer Extended

WLDIX

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

Distributes a decimal integer of data type 0, 1, 2, 3, 4, or 5 into the four FPACs.

AC1 must contain the data-type indicator describing the integer.

AC3 must contain a 32-bit byte pointer which is the address of the high-order byte of the integer.

The instruction uses the contents of AC3 to reference the integer. Extends the integer with high-order zeros until it is 32 digits long. Divides the integer into 4 units of 8 digits each and converts each unit to a floating point number. Places the number obtained from the 8 high-order digits into FAC0. Places the number obtained from the next 8 digits into FAC1. Places the number obtained from the next 8 digits into FAC2. Places the number obtained from the low-order 8 bits into FAC3. Sets the sign of each FPAC by checking the number just loaded into the FPAC. If the FPAC contains a nonzero number, then sets the sign of the FPAC to be the sign of the integer. If the FPAC contains an 8-digit zero, sets the FPAC to true zero. The *Z* and *N* flags in the floating point status register are unpredictable.

Upon successful termination, the contents of AC0 and AC1 remain unchanged. AC2 contains the original contents of AC3. AC3 points to the first byte following the integer field. Carry is unchanged and *overflow* is 0.

Wide Locate Lead Bit

WLOB *acs,acd*

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

Counts the number of high-order zeroes in ACS. Adds the count of high-order zeroes to the 32-bit, signed contents of ACD. Stores the result of the add in ACD. The contents of ACS remain unchanged, unless ACS and ACD are the same accumulator. Carry is unchanged and *overflow* is 0.

Wide Locate and Reset Lead Bit

WLRB *acs,acd*

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

Counts the number of high-order zeroes in ACS.

The instruction counts the high order zeroes in ACS. Adds the count of high-order zeroes to the 32-bit, signed contents of ACD. Stores the result in ACD. Sets the leading bit of ACS to 0. Carry is unchanged and *overflow* is 0.

If ACS equals ACD, then sets the leading bit to 0 and adds nothing to the contents of the specified accumulator.

Wide Logical Shift

WLSH *acs,acd*

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

Shifts the 32-bit contents of ACD either left or right.
Bits 24–31 of ACS specify the number of bits to shift ACD. If this number is positive, then the instruction shifts the contents of ACD the appropriate number of bits to the left. If this number is negative, then the instruction shifts the contents of ACD the appropriate number of bits to the right. If ACS contains zero, then no shifting occurs. The instruction ignores bits 0–23 of ACS.

Bits shifted out during this instruction are lost. Zeroes fill the vacated bit positions. The contents of ACS remain unchanged, unless ACD equals ACS. Carry is unchanged and *overflow* is 0.

Wide Logical Shift Immediate

WLSI *n,ac*

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

Shifts the contents of the specified accumulator to the left $n+1$ positions, where n is in the range 0 to 3. Carry is unchanged and *overflow* is 0.

> NOTE: *The assembler takes the coded value of* n *and subtracts one from it before placing it in the immediate field. Therefore, the programmer should code the exact value that he wishes to shift.*

Wide Load Sign

WLSN

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

Evaluates a decimal number as zero or nonzero, and the sign as positive or negative.

AC1 must contain the data type indicator describing the number.

AC3 must contain a byte pointer which is the address of the high-order byte of the number.

The instruction evaluates a decimal number in memory and returns in AC1 a code that classifies the number as zero or nonzero and identifies its sign. The meaning of the returned code is as follows:

Value of Number	Code
Positive non-zero	+1
Negative non-zero	-1
Positive zero	0
Negative zero	-2

Upon successful termination, the contents of AC0 remain unchanged; AC1 contains the value code; AC2 contains the original contents of AC3; and the contents of AC3 are unpredictable. The contents of the addressed memory locations remain unchanged. Carry is unchanged and *overflow* is 0.

Wide Move

WMOV *acs,acd*

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

Moves a copy of the 32-bit contents of ACS into ACD. The contents of ACS remain unchanged. Carry is unchanged and *overflow* is 0.

Wide Modify Stack Pointer

WMSP *ac*

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

Changes the value of the stack pointer and tests for potential overflow.

Shifts the contents of the specified accumulator left one bit. Adds the shifted value to the contents of the WSP and temporarily saves the result. Checks for fixed point overflow. If overflow occurs, the processor does not alter WSP and treats the overflow as a stack fault. AC1 contains the code 1.

If no overflow occurs, the instruction checks the value of the result. If the result is positive, the processor checks it against the stack limit for stack overflow; if negative, against the stack limit for stack underflow. If underflow or overflow does not occur, the instruction loads WSP with the saved value.

If either overflow or underflow occurs, the instruction does not alter WSP and a stack fault occurs. AC1 contains the code 1. The PC in the return block points to this instruction.

This instruction does not change carry; *overflow* is 0.

Wide Multiply

WMUL *acs,acd*

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

Multiplies two integers contained in accumulators.

The instruction multiplies the 32-bit, signed integer contained in ACD by the 32-bit, signed integer contained in ACS. Places the 32 least significant bits of the result in ACD. The contents of ACS and carry remain unchanged. *Overflow* is 0.

If the result is outside the range of -2,147,483,648 to +2,147,483,647 inclusive, sets *overflow* to 1; otherwise, *overflow* is 0. ACD will contain the 32 least significant bits of the result.

Wide Signed Multiply

WMULS

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

Multiplies two integers contained in accumulators.

The instruction multiplies the 32-bit, signed integer contained in AC1 by the 32-bit, signed integer contained in AC2. Adds the 32-bit signed integer contained in AC0 to the 64-bit result. Loads the 64-bit result into AC0 and AC1. AC0 contains the 32 high-order bits. AC2 and carry remain unchanged. *Overflow* is 0.

Wide Add with Narrow Immediate

WNADI *ac,immediate*

1	1	1	AC	1	1	0	1	1	1	1	1	0	0	1	IMMEDIATE
0	1	2	3 4	5	6	7	8	9	10	11	12	13	14	15	16 ... 31

Adds an immediate value to an integer contained in an accumulator.

The instruction sign extends the two's complement literal value contained in the immediate field to 32 bits. Adds the sign extended value to the 32-bit integer contained in the specified accumulator. Loads the result into the specified accumulator. Sets carry to the value of ALU carry. Sets *overflow* to 1 if there is an ALU overflow.

Wide Negate

WNEG *acs,acd*

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

Negates the contents of an accumulator.

The instruction forms the two's complement of the 32-bit contents of ACS. Loads the result into ACD. Sets carry to the value of ALU carry. Sets *overflow* to 1 if there is an ALU overflow. The contents of ACS remain unchanged, unless ACS equals ACD.

Wide Pop Accumulators

WPOP *acs,acd*

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

Pops up to 4 double words off the top of the wide stack and places them in the specified 32-bit accumulators.
Pops the top double word off the wide stack and places it in ACS. Pops the next double word off the wide stack and places it in ACS-1, and so on, until all specified accumulators have been loaded. If necessary, the accumulators wrap around, with AC3 following AC0, until all specified accumulators have been loaded. If ACS equals ACD, then the instruction pops only one double word off of the wide stack and places it in the specified accumulator.

The instruction decrements the contents of WSP by twice the number of double words popped. Carry is unchanged and *overflow* is 0.

Wide Pop Block

WPOPB

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

Pops six double words off the wide stack and places them in the appropriate locations.

The popped words and their destinations are as follows:

Double Word Popped	Destination
1	Bit 0 to carry; bits 1-31 to PC
2	AC3
3	AC2
4	AC1
5	AC0
6	Bit 0 to OVK; bit 1 to OVR; bit 2 to IRES; bits 17-31 are multiplied by 2 and incremented by 12. This number is subtracted from WSP. WSP is loaded with the result.

If the instruction specifies an inward ring crossing, then a protection fault occurs and the current wide stack remains unchanged. Note that the return block pushed as a result of the protection fault will contain undefined information. After the fault return block is pushed, AC0 contains the contents of the PC (which point to the instruction that caused the fault) and AC1 contains the code 8.

If the instruction specifies an intra-ring address, it pops the six-double-word block, then checks for stack underflow. If underflow has occurred, a stack underflow fault occurs. Note that the return block pushed as a result of the stack underflow will contain undefined information. After the fault return block is pushed, AC0 contains the contents of the PC (which point to the instruction that caused the fault) and AC1 contains the code 3. If there is no underflow, execution continues with the location addressed by the program counter.

If the instruction specifies an outward ring crossing, it pops the six-double-word return block and checks for stack underflow. If underflow has occurred, a stack underflow fault occurs. Note that the return block pushed as a result of the stack underflow will contain undefined information. After the fault return block is pushed, AC0 contains the contents of the PC (which point to the instruction that caused the fault) and AC1 contains the code 3. If there is no underflow, the instruction stores WSP and WFP in the appropriate page zero locations of the current segment. It then performs the outward ring crossing and loads the wide stack registers with the contents of the appropriate page zero locations of the new ring. Loads WSP with the value:

(*current contents of WSP*) - (2 x (*argument count*))

Checks for stack underflow. If underflow has occurred, a stack underflow fault occurs. Note that the return block pushed as a result of the stack underflow will contain undefined information. After the fault return block is pushed, AC0 contains the contents of the PC (which point to the instruction that caused the fault) and AC1 contains the code 3. If there is no underflow, execution continues with the location addressed by the program counter.

Pop PC and Jump

WPOPJ

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

Pops the top 31-bit value off the wide stack, loads it into the PC, then checks for stack overflow. Carry is unchanged and *overflow* is 0.

Push Accumulators

WPSH *acs,acd*

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

Pushes the contents of the specified 32-bit accumulators onto the top of the wide stack.

Pushes the contents of ACS onto the top of the wide stack, then pushes the contents of next sequential accumulators up to and including ACD. If necessary, the accumulators wrap around, with AC0 following AC3, until the contents of all specified accumulators have been pushed. If ACS equals ACD, then the instruction pushes the contents of only one accumulator onto the wide stack.

Note that the instruction increments the contents of WSP by two times the number of accumulators pushed (32-bit accumulators). Carry is unchanged and *overflow* is 0.

Wide Restore

WRSTR

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

Returns control from an interrupt.
When this instruction is used, the wide stack should contain the following information, in the given order:

Contents	Size of Word	Notes
WFP	(32 bits)	
WSP	(32 bits)	
WSL	(32 bits)	
WSB	(32 bits)	
SFA	(Lower 16 bits)	Stack fault address
OVK, OVR	(Bits 0 and 1)	
AC0	(32 bits)	
AC1	(32 bits)	
AC2	(32 bits)	
AC3	(32 bits)	
Carry, PC	(32 bits)	This is the top of the wide stack.

The instruction checks to see if the ring crossing specified is inward. If the crossing is inward, a protection fault occurs (code=8 in AC1).

If the crossing is not inward, the instruction pops the return block on top of the wide stack and places the block contents in the appropriate registers. Next, the instruction pops the stack registers and the stack fault address, temporarily saves them, and checks for stack underflow. If no underflow occurs, further actions depend upon the type of ring call.

If the restore is to be to the same ring, the instruction places the temporarily saved stack management information in the four stack registers. Stores the stack fault address in location 14_8 of the current segment. Checks for stack underflow. If underflow has occurred, a stack underflow fault occurs (code=3 in AC1). If underflow has not occurred, execution continues with the location specified by the PC.

If the ring crossing is outward, the instruction stores the stack management information held internally into the appropriate page zero locations of the current segment. Performs the outward ring crossing. Loads the stack registers with the contents of the appropriate page zero locations of the new segment. Checks for stack underflow. If underflow has occurred, a stack underflow fault occurs (code=3 in AC1). If underflow has not occurred, execution continues with the location specified by the PC.

Wide Return

WRTN

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

Returns control from subroutines that issue a **WSAVS** or a **WSAVR** instruction at their entry point. Places the contents of WFP in WSP and executes a **WPOPB** instruction. Places the popped value of AC3 in WFP.

Wide Skip on All Bits Set in Accumulator

WSALA *ac,immediate*

1	0	1	AC	1	1	0	1	0	0	1	1	0	0	1	IMMEDIATE
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15 16 ... 47

Performs a logical AND between an immediate value and the contents of an accumulator. Skips depending on the result of the AND.

The instruction performs a logical AND on the contents of the immediate field and the complement of the contents of the specified accumulator. If the result of the AND is zero, then execution skips the next sequential word before continuing. If the result of the AND is nonzero, then execution continues with the next sequential word. The contents of the specified accumulator remain unchanged. Carry is unchanged and *overflow* is 0.

Wide Skip on All Bits Set in Double-word Memory Location

WSALM *ac,immediate*

1	0	1	AC	1	1	0	1	0	1	1	1	0	0	1	IMMEDIATE
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15 16 ... 47

Performs a logical AND between an immediate value and the complement of a memory word. Skips depending on the result of the AND.

The instruction performs a logical AND on the contents of the immediate field and the complement of the double word addressed by the specified accumulator. If the result of the AND is zero, then execution skips the next sequential word before continuing. If the result of the AND is nonzero, then execution continues with the next sequential word. The contents of the specified accumulator and memory location remain unchanged. Carry is unchanged and *overflow* is 0.

Wide Skip on Any Bit Set in Accumulator

WSANA *ac,immediate*

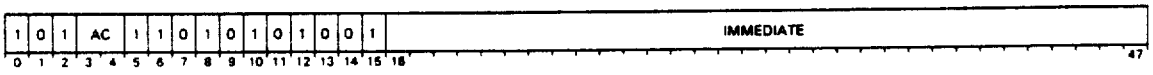
1	0	1	AC	1	1	0	1	0	0	0	1	0	0	1	IMMEDIATE
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15 16 ... 47

Performs a logical AND between an immediate value and the contents of an accumulator. Skips depending on the result of the AND.

The instruction performs a logical AND on the contents of the immediate field and the contents of the specified accumulator. If the result of the AND is nonzero, then execution skips the next sequential word before continuing. If the result of the AND is zero, then execution continues with the next sequential word. The contents of the specified accumulator remain unchanged. Carry is unchanged and *overflow* is 0.

Wide Skip on Any Bit Set in Double-word Memory Location

WSANM *ac,immediate*

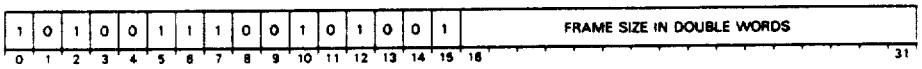


Performs a logical AND between an immediate value and the contents of a memory word. Skips depending on the result of the AND.

The instruction performs a logical AND on the contents of the immediate field and the contents of the double word addressed by the specified accumulator. If the result of the AND is nonzero, then execution skips the next sequential word before continuing. If the result of the AND is zero, then execution continues with the next sequential word. The contents of the specified accumulator and memory location remain unchanged. Carry is unchanged and *overflow* is 0.

Wide Save/Reset Overflow Mask

WSAVR

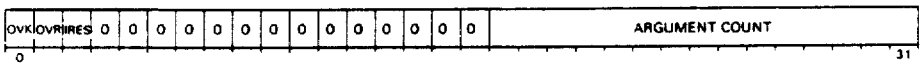


Pushes a return block onto the wide stack and resets OVK.

The instruction checks for stack overflow. If an overflow would occur, then control transfers to the wide stack fault routine. If no overflow would occur, then the instruction pushes five double words of a wide six-double word return block onto the wide stack. The words pushed have the following contents:

Double Word Pushed	Contents
1	AC0
2	AC1
3	AC2
4	AC3
5	carry and PC

Note that the five words described above do not make up the entire return block. Either the LCALL or the XCALL instruction pushes the first double word of the return block onto the wide stack. This word has the following format:



After pushing the return block, the instruction places the value of the stack pointer in WFP and AC3. Multiplies the 16-bit, unsigned integer contained in the second instruction word by 2. Adds the result to WSP. Sets OVK to 0, disabling integer overflow.

Wide Save/Set Overflow Mask

WSAVS

1	0	1	0	0	1	1	1	0	0	1	1	1	0	0	1	FRAME SIZE IN DOUBLE WORDS
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16 ... 31

Pushes a return block onto the wide stack, resets WSP and WFP, and sets **OVK** to 1.

The instruction checks for stack overflow. If an overflow would occur, then control transfers to the wide stack fault routine. If no overflow would occur, then the instruction pushes five double words of a wide six-double word return block onto the stack. The words pushed have the following contents:

Double Word Pushed	Contents
1	AC0
2	AC1
3	AC2
4	AC3
5	carry and PC

Note that the five double words described above do not make up the entire return block. Either the **LCALL** or the **XCALL** instruction pushes the first double word of the return block onto the wide stack. This word has the following format:

OVK	OVR	RES	0	0	1	1	1	0	0	1	1	1	0	0	1	FRAME SIZE IN DOUBLE WORDS
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16 ... 31

After pushing the return block, the instruction places the value of WSP in WFP and AC3. Multiplies the 16-bit, unsigned integer contained in the second instruction word by 2. Adds the result to WSP. Sets **OVK** to 1, enabling integer overflow.

Wide Subtract Immediate

WSBI *n,ac*

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

Subtracts an integer in the range 1 to 4 from an integer contained in an accumulator.

The instruction subtracts the value $n+1$ from the value contained in the specified accumulator. Stores the result in the specified accumulator. Sets carry to the value of ALU carry. Sets *overflow* to 1 if there is an ALU overflow.

> NOTE: *The assembler takes the coded value of* n *and subtracts 1 from it before placing it in the immediate field. Therefore, the programmer should code the exact value that he wishes to subtract.*

Wide Skip If Equal To

WSEQ *acs,acd*

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

Compares one integer to another and skips if the two integers are equal. Carry is unchanged and *overflow* is 0.

The instruction compares the 32-bit integer contained in ACS to the 32-bit integer in ACD. If the integer contained in ACS is equal to the integer contained in ACD, the next 16-bit word is skipped; otherwise, the next word is executed.

If ACS and ACD are the same accumulator, then the instruction compares the integer contained in the accumulator to zero. The skip will occur if the integer equals zero.

Wide Signed Skip If Greater Than Or Equal To

WSGE *acs,acd*

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

Compares one integer to another and skips if the first is greater than or equal to the second. Carry is unchanged and *overflow* is 0.

The instruction compares the signed, 32-bit integer contained in ACS to the signed, 32-bit integer in ACD. If the integer contained in ACS is greater than or equal to the integer contained in ACD, then the next word is skipped; otherwise, the next instruction is executed.

If ACS and ACD are the same accumulator, then the instruction compares the integer contained in the accumulator to zero. The skip will occur if the integer is greater than or equal to zero.

Wide Signed Skip If Greater Than

WSGT *acs,acd*

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

Compares one integer to another and skips if the first is greater than the second. Carry is unchanged and *overflow* is 0.

The instruction compares the signed, 32-bit integer contained in ACS to the signed 32-bit integer in ACD. If the integer contained in ACS is greater than the integer contained in ACD, the next word is skipped; otherwise, the next word is executed.

If ACS and ACD are the same accumulator, then the instruction compares the integer contained in the accumulator to zero. The skip will occur if the integer is greater than zero.

Wide Skip on Bit Set to One

WSKBO *bit number*

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

Tests a specified bit in AC0 and skips if the bit is one.

The instruction uses the bits specified in bits 1–3 and 10–11 to specify a bit position in the range 0–31. This number specifies one bit in AC0; the value 0 specifies the highest-order bit, and the value 31 specifies the lowest-order bit. If the specified bit has the value 1, then the next sequential word is skipped. If the bit has the value 0, then the next sequential word is executed. The contents of AC0 remain unchanged. Carry is unchanged and *overflow* is 0.

Wide Skip on Bit Set to Zero

WSKBZ *bit number*

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

Tests a specified bit in AC0 and skips if the bit is 0.

The instruction uses the bits specified in bits 1–3 and 10–11 to specify an a bit position in the range 0–31. This number specifies one bit in AC0; the value 0 specifies the highest-order bit, and the value 31 specifies the lowest-order bit. If the specified bit has the value 0, then the next sequential word is skipped. If the bit has the value 1, then the next sequential word is executed. The contents of AC0 remain unchanged. Carry is unchanged and *overflow* is 0.

Wide Signed Skip If Less Than Or Equal To

WSLE *acs,acd*

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

Compares one integer to another and skips if the first is less than or equal to the second. Carry is unchanged and *overflow* is 0.

The instruction compares the signed, 32-bit integer contained in ACS to the signed, 32-bit integer in ACD. If the integer contained in ACS is less than or equal to the integer contained in ACD, the next word is skipped; otherwise, the next sequential word is executed.

If ACS and ACD are the same accumulator, then the instruction compares the integer contained in the accumulator to zero. The skip will occur if the integer is less than or equal to zero.

Wide Signed Skip If Less Than

WSLT *acs,acd*

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

Compares one integer to another and skips if the first is less than the second. Carry is unchanged and *overflow* is 0.

The instruction compares the signed, 32-bit integer contained in ACS to the signed, 32-bit integer in ACD. If the integer contained in ACS is less than the integer contained in ACD, the next word is skipped; otherwise, the next sequential word is executed.

If ACS and ACD are the same accumulator, then the instruction compares the integer contained in the accumulator to zero. The skip will occur if the integer is less than zero.

Wide Skip on Nonzero Bit

WSNB *acs,acd*

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

Tests the value of an addressed bit and skips if the bit is one. Carry is unchanged and *overflow* is 0.

The instruction forms a bit pointer from the contents of ACS and ACD. ACS contains the high-order bits of the bit pointer; ACD contains the low-order bits. ACS and ACD can be specified to be the same accumulator; in this case, the specified accumulator supplies the low-order bits of the bit pointer. The high-order bits are treated as if they were zero in the current segment.

The instruction checks the value of the bit referenced by the bit pointer. If the bit has the value 1, the next sequential word is skipped. If the bit has the value 0, the next sequential instruction is executed.

Wide Skip If Not Equal To

WSNE *acs,acd*

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

Compares one integer to another and skips if the two are not equal. Carry is unchanged and *overflow* is 0.

The instruction compares the 32-bit integer contained in ACS to the 32-bit integer in ACD. If the integer contained in ACS is not equal to the integer contained in ACD, then execution skips the next word; otherwise, execution proceeds with the next sequential word.
If ACS and ACD are the same accumulator, then the instruction compares the integer contained in the accumulator to zero. The skip will occur if the integer does not equal zero.

Wide Special Save/Set Overflow Mask

WSSVR

1	0	0	0	0	1	1	1	0	0	1	0	1	0	0	1	FRAME SIZE IN DOUBLE WORDS
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16 ... 31

Pushes a wide return block onto the wide stack and sets **OVK** to 0.

The instruction checks for stack overflow. If executing the instruction would cause an overflow, the instruction transfers control to the wide stack fault handler. The PC in the fault return block will contain the address of the **WSSVR** instruction.

Pushes a wide return block onto the wide stack. After pushing the sixth double word, places the value of WSP in WFP and AC3. Increments WSP by twice the frame size. The frame size is a 16-bit, unsigned integer contained in the second word of this instruction. Sets **OVK** to 0, which disables integer overflow. Sets **OVR** to 0.

The structure of the wide return block pushed is as follows:

Word in Block	Contents
1-2	OVK, OVR, IRES, 29 zeroes
3-4	AC0
5-6	AC1
7-8	AC2
9-10	AC3
11-12	Previous WFP
13-14	Carry, return PC value
15-18	Stack frame

NOTE: *This instruction saves the information required by the* **WRTN** *instruction.*

This instruction is typically executed after an **XJSR** or **LJSR** instruction. Note that neither of these jump instructions can perform a cross ring call. However, they may be used with **WSSVS** to perform an intra-ring transfer to a subroutine that requires no parameters, and that uses **WRTN** to return control back to the calling sequence.

Wide Special Save/Set Overflow Mask

WSSVS

1	0	0	0	0	1	1	1	0	0	1	1	1	0	0	1	FRAME SIZE IN DOUBLE WORDS
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16 ... 31

Pushes a wide return block onto the wide stack and sets **OVK** to 1.
The instruction checks for stack overflow. If executing the instruction would cause an overflow, the instruction transfers control to the wide stack fault handler. The PC in the fault return block will contain the address of the **WSSVS** instruction.

If no overflow would occur, the instruction pushes a wide return block onto the wide stack. After pushing the sixth double word, places the value of WSP in WFP and AC3. Increments WSP by twice the frame size (a 16-bit, unsigned integer contained in the second word of this instruction). Sets **OVK** to 1, which enables integer overflow. Sets **OVR** to 0.

The structure of the wide return block pushed is as follows:

Word in Block	Contents
1-2	OVK, OVR, IRES , 29 zeroes
3-4	AC0
5-6	AC1
7-8	AC2
9-10	AC3
11-12	Previous WFP
13-14	Carry, return PC value
15-18	Stack frame

NOTE: *This instruction saves the information required by the* **WRTN** *instruction.*

This instruction is typically executed after an **XJSR** or **LJSR** instruction. Note that neither of these jump instructions can perform a cross ring call. However, they may be used with **WSSVR** to perform an intra-ring transfer to a subroutine that requires no parameters, and that uses **WRTN** to return control back to the calling sequence.

Wide Store Byte

WSTB *acs,acd*

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

Stores a copy of the rightmost byte of ACD into memory at the address specified by ACS.

ACS contains a 32-bit byte address of some location of memory.

The instruction stores a copy of ACD's bits 24–31 at the locations specified by ACS. The contents of ACS and ACD remain unchanged. Carry is unchanged and *overflow* is 0.

Wide Store Integer
WSTI *fpac*

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

Converts a floating point number to an integer and stores it into memory.

AC1 contains the data-type indicator that describes the integer.

AC3 contains a 32-bit byte pointer to a byte in memory. The instruction will store the high order byte of the number in this location, with the low order bytes following in subsequent locations.

Under the control of accumulators AC1 and AC3, the instruction translates the contents of the specified FPAC to an integer of the specified type and stores it, right-justified, in memory beginning at the specified location. The instruction leaves the floating point number unchanged in the FPAC, and destroys the previous contents of memory at the specified location(s).

Upon successful completion, the instruction leaves accumulators AC0 and AC1 unchanged. AC2 contains the original contents of AC3. AC3 contains a byte pointer to the first byte following the destination field. The value of carry is indeterminate and *overflow* is 0.

> NOTES: *If the number in the specified FPAC has any fractional part, the result of the instruction is undefined. Use the* Integerize *instruction to clear any fractional part.*
>
> *If the number to be stored is too large to fit in the destination field, this instruction discards high-order digits until the number fits. This instruction stores the remaining low-order digits and sets carry to 1.*
>
> *If the number to be stored does not completely fill the destination field, the data type of the number determines the instruction's actions. If the number is data type 0, 1, 2, 3, 4, or 5, the instruction sets the high-order bytes to 0. If the number is data type 6, the instruction sign extends it to fill the gap. If the number is data type 7, the instruction sets the low-order bytes to 0.*

Wide Store Integer Extended
WSTIX

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

Converts a floating point number to an integer and stores it in memory.

AC1 must contain the data-type indicator describing the integer.

AC3 must contain a 32-bit byte pointer pointing to the high-order byte of the destination field in memory.

Using the information in AC1, the instruction converts the contents of each of the FPACs to integer form. Forms a 32-bit integer from the low-order 8 digits of each FPAC. Right justifies the integer and stores it in memory beginning at the location specified by AC3. The sign of the integer is the logical OR of the signs of all four FPAC's. The previous contents of the addressed memory locations are lost. Sets carry to 0. The contents of the FPACs remain unchanged. The condition codes in the FPSR are unpredictable.

Upon successful termination, the contents of AC0 and AC1 remain unchanged; AC2 contains the original contents of AC3; and AC3 contains a byte pointer pointing to the

first byte following the destination field. The contents of carry are indeterminate and *overflow* is 0.

> NOTES: *If the integer is too large to fit in the destination field, the instruction discards high-order digits until the integer fits. The instruction stores remaining low-order digits and sets carry to 1.*
>
> *If the integer does not completely fill the destination field, the data type of the integer determines the instruction's actions. If the data type is 0, 1, 2, 3, 4, or 5, the instruction sets the high-order bytes to 0. Data types 6 and 7 are illegal and will cause a commercial fault.*

Wide Subtract

WSUB *acs,acd*

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

Subtracts the 32-bit integer contained in ACS from the 32-bit integer contained in ACD. Stores the result in ACD. Sets carry to the value of ALU carry. Sets *overflow* to 1 if there is an ALU overflow. The contents of ACS remain unchanged. Carry is unchanged and *overflow* is 0.

Wide Skip on Zero Bit

WSZB *acs,acd*

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

Tests a bit and skips if the bit is zero. Carry is unchanged and *overflow* is 0.

The instruction forms a bit pointer from the contents of ACS and ACD. ACS contains the high-order bits of the bit pointer; ACD contains the low-order bits. ACS and ACD can be specified to be the same accumulator; in this case, the specified accumulator supplies the low-order bits of the bit pointer. The high-order bits are treated as if they were zero in the current ring.

The instruction checks the value of the bit referenced by the bit pointer. If the bit has the value 0, the next sequential word is skipped. If the bit has the value 1, the next sequential word is executed.

Wide Skip on Zero Bit and Set Bit To One

WSZBO *acs,acd*

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

Tests a bit. Sets the tested bit to 1 and skips if the tested value was zero. Carry is unchanged and *overflow* is 0.

The instruction forms a bit pointer from the contents of ACS and ACD. ACS contains the high-order bits of the bit pointer; ACD contains the low-order bits. ACS and ACD can be specified to be the same accumulator; in this case, the specified accumulator supplies the low-order bits of the bit pointer. The high-order bits are treated as if they were zero.

The instruction checks the value of the bit referenced by the bit pointer. If the bit has the value 0, then the instruction sets the bit to one and skips the next sequential word. If the bit has the value 1, then no skip occurs.

Wide Unsigned Skip If Greater Than Or Equal To

WUSGE *acs,acd*

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

Compares one integer to another and skips if the first is greater than or equal to the second. Carry is unchanged and *overflow* is 0.

The instruction compares the unsigned, 32-bit integer contained in ACS to the unsigned 32-bit integer in ACD. If the integer contained in ACS is greater than or equal to the integer contained in ACD, the next sequential word is skipped; otherwise, the next sequential word is executed.

If ACS and ACD are the same accumulator, then the instruction compares the integer contained in the accumulator to zero. The skip will occur if the integer is greater than or equal to zero.

Wide Unsigned Skip If Greater Than

WUSGT *acs,acd*

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

Compares one integer to another and skips if the first is greater than the second. Carry is unchanged and *overflow* is 0.

The instruction compares the unsigned, 32-bit integer contained in ACS to the unsigned 32-bit integer in ACD. If the integer contained in ACS is greater than the integer contained in ACD, the next sequential word is skipped; otherwise, the next sequential word is executed.

If ACS and ACD are the same accumulator, then the instruction compares the integer contained in the accumulator to zero. The skip will occur if the integer is greater than zero.

Wide Exchange

WXCH *acs,acd*

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

Exchanges the 32-bit contents of ACS and ACD. Carry is unchanged and *overflow* is 0.

Wide Extended Operation

WXOP *acs,acd,operation #*

1	0	1	0	0	1	1	1	0	0	0	0	1	0	0	1	ACS	ACD	0	0	0	0	0	0	OP #	0
0	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 30	31

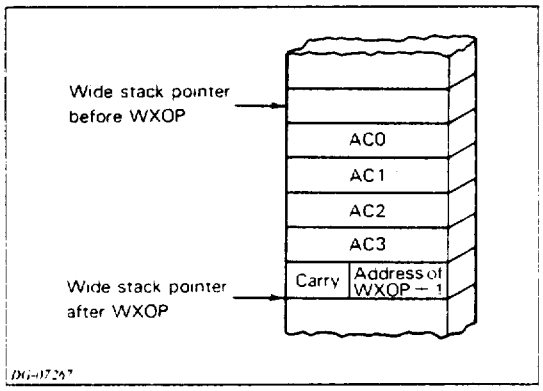
Pushes a return block onto the wide stack and transfers control to an extended operation procedure. Carry is unchanged and *overflow* is 0.

The instruction pushes a return block onto the wide stack. Places the address in the wide stack of ACS into AC2; places the address in the wide stack of ACD into AC3. Memory

locations $12–13_8$ must contain the WXOP origin address, the starting address of a 40_8 word table of addresses. These addresses are the starting location of the various WXOP operations.

The instruction adds the operation number in the WXOP instruction to the WXOP origin address to produce the address of a double word in the WXOP table. Fetches that word and treats it as the intermediate address in the effective address calculation. After the indirection chain, if any, has been followed, the instruction places the effective address in the program counter. The contents of AC0, AC1, and the WXOP origin address remain unchanged. All addresses must be in the current segment.

The format of the return block pushed by the instruction is as follows:



This return block is designed so that the WXOP procedure can return control to the calling program via the **WPOP** instruction.

Wide Alternate Extended Operation

WXOP1 *acs,acd,operation #*

1	0	1	0	0	1	1	1	0	0	0	1	1	0	0	1	ACS	ACD	0	0	0	0	0	1	0	OP #	0
0	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 30	31

Pushes a return block and transfers control to an extended operation procedure. Carry is unchanged and *overflow* is 0.

The instruction operates in exactly the same way as **WXOP** except that it adds 40_8 to the entry number before it adds the entry number to the **WXOP** origin address. In addition, it can specify only 16 entry locations.

Wide Exclusive OR

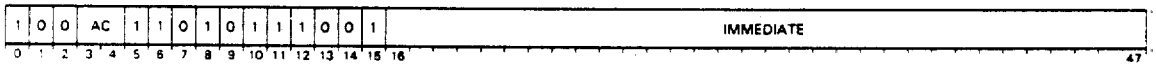
WXOR *acs,acd*

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

Forms the logical exclusive OR between corresponding bits of ACS and ACD. Loads the 32-bit result into ACD. The contents of ACS remain unchanged, unless ACS equals ACD. Carry is unchanged and *overflow* is 0.

Wide Exclusive OR Immediate

WXORI *ac,immediate*

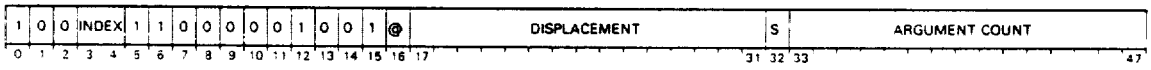


Forms a logical OR between two values.

The instruction forms the logical exclusive OR between corresponding bits of the specified accumulator and the value contained in the literal field. The instruction places the result of the exclusive OR in the specified accumulator. Carry is unchanged and *overflow* is 0.

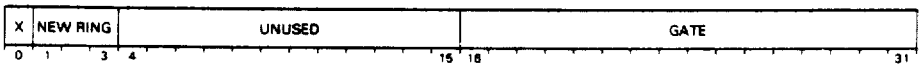
Call Subroutine (Extended Displacement)

XCALL *opcode,argument count,displacement*



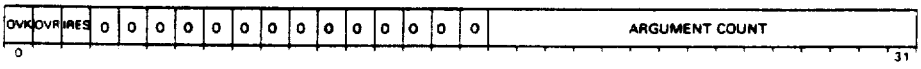
Evaluates the address of a subroutine call.
If the target address specifies an outward ring crossing, a protection fault (code=7 in AC1) occurs. Note that the contents of the PC in the return block are undefined.

If the target address specifies an inward ring call, then the instruction assumes the target address has the following format:

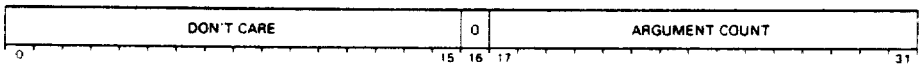


The instruction checks the gate field of the above format for a legal gate. If the specified gate is illegal, a protection fault (code = 6 in AC1) occurs and call is made. Note that the contents of the PC in the return block are undefined.

If the specified gate is legal, or if the target address specifies an intra-ring crossing, then the instruction loads the contents of the PC, + 3, into AC3. The contents of AC3 will always reference the current segment. If bit 0 of the argument count is 0, then the instruction creates a word with the following format:



The instruction pushes this word onto the wide stack. If a stack overflow occurs after this push, a stack fault occurs and no call is made. Note that the value of the PC in the return block is undefined. If bit 0 of the argument count is 1, then the instruction assumes the top word of the wide stack has the following format:



The instruction modifies this word to include the correct settings of **OVK** and **OVR** in bits 0 and 1.

Regardless of the setting of the argument count's bit 0, the instruction next unconditionally sets **OVR** to 0 and loads the PC with the target address. Execution continues with the word referenced by the PC.

Exchange Accumulators

XCH *acs,acd*

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

Exchanges the contents of two accumulators.

Places the original contents of bits 16-31 of ACS into bits 16-31 of ACD and the original contents of bits 16-31 of ACD in bits 16-31 of ACS. Carry remains unchanged and *overflow* is 0.

Bit 0-15 of the modified accumulator are undefined after completion of this instruction.

Execute

XCT *ac*

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

Executes the instruction contained in bits 16-31 of the specified accumulator as if it were in main memory in the location occupied by the *Execute* instruction. If the instruction in bits 16-31 of the specified accumulator is an *Execute* instruction that specifies the same accumulator, the processor is placed in a one-instruction loop.

This instruction leaves carry unchanged; *overflow* is 0.

Because of the possibility of bits 16-31 of the specified accumulator containing an *Execute* instruction, this instruction is interruptible. An I/O interrupt can occur immediately prior to each time the instruction in accumulator is executed. If an I/O interrupt does occur, the program counter in the return block pushed on the system stack points to the *Execute* instruction in main memory. This capability to execute an *Execute* instruction gives you a *wait for I/O interrupt* instruction.

> NOTES: *If bits 16-31 of the specified accumulator contains the first word of a two-word instruction, the word following the* XCT *instruction is used as the second word. Normal sequential operation then continues from the second word after the* XCT *instruction.*
>
> *Do not use the* XCT *instruction to execute an instruction that requires all four accumulators, such as* CMV, CMT, CMP, CTR, *or* BAM.

The results of XCT are undefined if bits 16-31 of the specified accumulator contains an instruction that modifies that same accumulator. For example:")

```
        LDA     0,TOT
        XCT     0                   ;UNDEFINED
        JMP     ON
TOT:    ADD     1,0
```

Add Double (Memory to FPAC) (Extended Displacement

XFAMD *fpac,[@]displacement[,index]*

1	INDEX	FPAC	0	0	0	0	0	0	1	1	0	0	1	@	DISPLACEMENT
0	1 2	3 4	5	6	7	8	9	10	11	12	13	14	15	16	17 ... 31

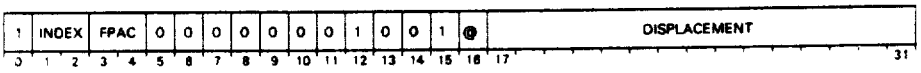
Adds the 64-bit floating point number in the source location to the 64-bit floating point number in FPAC and places the normalized result in FPAC.

Computes the effective address, *E*. Uses *E* to address a double precision (four word) operand. Adds this 64-bit floating point number to the floating point number in the specified FPAC. Places the normalized result in the specified FPAC. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Add Single (Memory to FPAC) (Extended Displacement)

XFAMS *fpac,[@]displacement[,index]*



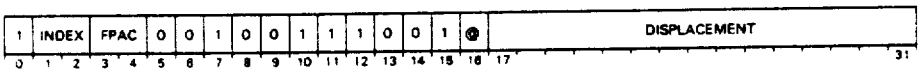
Adds the 32-bit floating point number in the source location to the 32-bit floating point number in FPAC and places the normalized result in FPAC.

Computes the effective address, *E*. Uses *E* to address a single precision (double word) operand. Adds this 32-bit floating point number to the floating point number in bits 0-31 of the specified FPAC. Places the normalized result in the specified FPAC. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Divide Double (FPAC by Memory) (Extended Displacement)

XFDMD *fpac,[@]displacement[,index]*



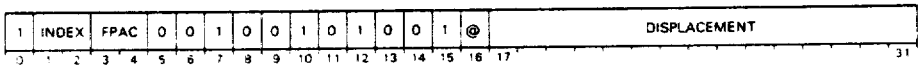
Divides the 64-bit floating point number in FPAC by the 64-bit floating point number in the source location and places the normalized result in FPAC.

Computes the effective address, *E*. Uses *E* to address a double precision (four word) operand. Divides the floating point number in the specified FPAC by this 64-bit floating point number. Places the normalized result in the specified FPAC. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Divide Single (FPAC by Memory) (Extended Displacement)

XFDMS *fpac,[@]displacement[,index]*



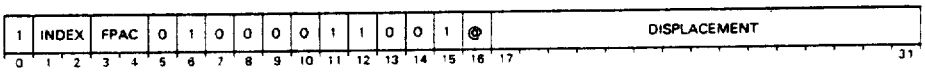
Divides the 32-bit floating point number in bits 0-31 of FPAC by the 32-bit floating point number in the source location and places the normalized result in FPAC.

Computes the effective address, *E*. Uses *E* to address a single precision (double word) operand. Divides the floating point number in bits 0-31 of the specified FPAC by this 32-bit floating point number. Places the normalized result in the specified FPAC. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Extended Load Floating Point Double

XFLDD *fpac,[@]displacement[,index]*



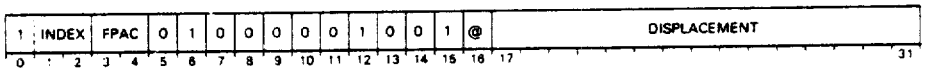
Moves four words out of memory and into a specified FPAC.

Computes the effective address, *E*. Fetches the double precision floating point number at the address specified by *E* and places it in FPAC. Updates the *Z* and *N* flags in the FPSR to reflect the new contents of FPAC.

> NOTE: *This instruction will move unnormalized data without change, but the Z and N flags will be undefined.*

Extended Load Floating Point Single

XFLDS *fpac,[@]displacement[,index]*



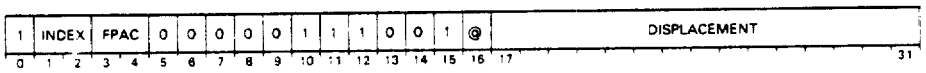
Moves two words out of memory into a specified FPAC.

Computes the effective address, *E*. Fetches the single precision floating point number at the address specified by *E*. Places the number in the high-order bits of FPAC. Sets the low-order 32 bits of FPAC to 0. Updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

> NOTE: *This instruction will move unnormalized or illegal data without change, but the Z and N flags will be undefined.*

Multiply Double (FPAC by Memory) (Extended Displacement)

XFMMD *fpac,[@]displacement[,index]*



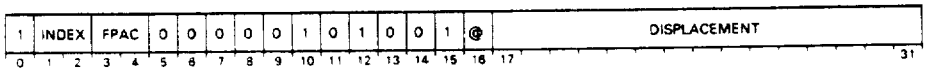
Multiplies the 64-bit floating point number in the source location by the 64-bit floating point number in FPAC and places the normalized result in FPAC.

Computes the effective address, *E*. Uses *E* to address a double precision (four word) operand. Multiplies this 64-bit floating point number by the floating point number in the specified FPAC. Places the normalized result in the specified FPAC. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Multiply Single (FPAC by Memory) (Extended Displacement)

XFMMS *fpac,[@]displacement[,index]*



Multiplies the 32-bit floating point number in the source location by the 32-bit floating point number in bits 0-31 of FPAC and places the normalized result in FPAC.

Computes the effective address, *E*. Uses *E* to address a single precision (double word) operand. Multiplies this 32-bit floating point number by the floating point number in bits 0-31 of the specified FPAC. Places the normalized result in bits 0-31 of the specified FPAC. Sets bits 32-63 of FPAC to 0. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Subtract Double (Memory from FPAC) (Extended Displacement)

XFSMD *fpac,{@}displacement[,index]*

1	INDEX	FPAC	0	0	1	0	0	0	1	1	0	0	1	@	DISPLACEMENT
0 1	2	3 4	5	6	7	8	9	10	11	12	13	14	15	16	17 31

Subtracts the 64-bit floating point number in the source location from the 64-bit floating point number in FPAC and places the normalized result in FPAC.

Computes the effective address, *E*. Uses *E* to address a double precision (four word) operand. Subtracts this 64-bit floating point number from the floating point number in the specified FPAC. Places the normalized result in the specified FPAC. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Subtract Single (Memory from FPAC) (Extended Displacement)

XFSMS *fpac,{@}displacement[,index]*

1	INDEX	FPAC	0	0	1	0	0	0	0	1	0	0	1	@	DISPLACEMENT
0 1	2	3 4	5	6	7	8	9	10	11	12	13	14	15	16	17 31

Subtracts the 32-bit floating point number in the source location from the 32-bit floating point number in bits 0-31 of FPAC and places the normalized result in FPAC.

Computes the effective address, *E*. Uses *E* to address a single precision (double word) operand. Subtracts this 32-bit floating point number from the floating point number in bits 0-31 of the specified FPAC. Places the normalized result in the specified FPAC. Sets bits 32-63 of FPAC to 0. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Store Floating Point Double (Extended Displacement)

XFSTD *fpac,{@}displacement[,index]*

1	INDEX	FPAC	0	1	0	0	0	1	1	1	0	0	1	@	DISPLACEMENT
0 1	2	3 4	5	6	7	8	9	10	11	12	13	14	15	16	17 31

Stores the contents of a specified FPAC into a memory location.

Computes the effective address, *E*. Places the floating point number contained in FPAC in memory beginning at the location addressed by *E*. Destroys the previous contents of the addressed memory location. The contents of FPAC and the condition codes in the FPSR remain unchanged.

NOTE: *This instruction moves unnormalized or illegal data without change.*

Store Floating Point Single (Extended Displacement)

XFSTS *fpac,[@]displacement[,index]*

1	INDEX	FPAC	0	1	0	0	0	1	0	1	0	0	1	@	DISPLACEMENT

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 31

Stores the contents of a specified FPAC into a memory location.

Computes the effective address, *E*. Places the 32 high-order bits of FPAC in memory beginning at the location addressed by *E*. Destroys the previous contents of the addressed memory location. The contents of FPAC and the condition codes in the FPSR remain unchanged.

NOTE: *This instruction moves unnormalized or illegal data without change.*

Jump (Extended Displacement)

XJMP *index,displacement*

1	1	0	INDEX	1	1	0	0	0	0	0	1	0	0	1	@	DISPLACEMENT

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 31

Calculates the effective address, *E*. Loads *E* into the PC. Carry is unchanged and *overflow* is 0.

NOTE: *The calculation of* E *is forced to remain within the current segment of execution.*

Jump to Subroutine (Extended Displacement)

XJSR *index,displacement*

1	1	0	INDEX	1	1	0	0	0	0	1	1	0	0	1	@	DISPLACEMENT

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 31

Calculates the effective address, *E*. Loads the current value of the PC, plus two, into AC3. Loads *E* into the PC. Carry is unchanged and *overflow* is 0.

NOTE: *The calculation of* E *is forced to remain within the current segment of execution.*

Load Effective Address (Extended Displacement)

XLEF *ac,index,displacement*

1	INDEX	AC	1	0	0	0	0	0	0	1	0	0	1	@	DISPLACEMENT

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 31

Loads an effective address into an accumulator.

The instruction calculates the effective address, *E*. Checks *E* for ring crossing errors. If no errors occur, loads *E* into the specified accumulator. If errors occur, issues a protection fault. Carry is unchanged and *overflow* is 0.

Load Effective Byte Address (Extended Displacement)

XLEFB *ac,index,displacement*

1	INDEX	AC	1	0	0	0	0	1	1	1	0	0	1	DISPLACEMENT

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 31

Loads an effective byte address into an accumulator. Carry is unchanged and *overflow* is 0.

The instruction calculates the effective byte address. Checks the byte address for ring crossing errors. If no errors occur, loads the byte address into the specified accumulator. If errors occur, issues a protection fault.

NOTE: *Index bits of 00 force the first address in the effective address calculation to be in the current segment of execution.*

Narrow Add Accumulator to Memory Word (Extended Displacement)

XNADD *ac,index,displacement*

1	INDEX	AC	0	0	0	0	0	0	1	1	0	0	0	@	DISPLACEMENT
0	1 2	3 4	5	6	7	8	9	10	11	12	13	14	15	16	17 31

Adds an integer in a memory location to an integer in an accumulator.

The instruction calculates the effective address, *E*. Adds the 16-bit integer contained in the location specified by *E* to the integer contained in bits 16–31 of the specified accumulator. Sign extends the 16-bit result to 32 bits and loads it into the specified accumulator. Sets carry to the value of ALU carry, and *overflow* to 1 if there is an ALU overflow. The contents of the referenced memory location remain unchanged.

Narrow Divide Memory Word (Extended Displacement)

XNDIV *ac,index,displacement*

1	INDEX	AC	0	0	0	1	1	0	1	1	0	0	0	@	DISPLACEMENT
0	1 2	3 4	5	6	7	8	9	10	11	12	13	14	15	16	17 31

Divides an integer contained in an accumulator by an integer in memory.

The instruction calculates the effective address, *E*. Sign extends the integer contained in bits 16–31 of the specified accumulator to 32 bits and divides it by the 16-bit integer contained in the location specified by *E*. If the quotient is within the range -32,768 to +32,767 inclusive, sign extends the result to 32 bits and loads it into the specified accumulator. If the quotient is outside of this range, or if the divisor is zero, the instruction sets *overflow* to 1 and leaves the specified accumulator unchanged. Otherwise, *overflow* is 0. The contents of the referenced memory location and carry remain unchanged.

Narrow Decrement and Skip if Zero (Extended Displacement)

XNDSZ *index,displacement*

1	0	1	INDEX	1	1	0	0	0	0	0	1	0	0	1	@	DISPLACEMENT
0	1	2	3 4	5	6	7	8	9	10	11	12	13	14	15	16	17 31

Calculates the effective address, *E*. Decrements the 16-bit contents of the location addressed by *E*. If the decremented result is equal to zero, then the instruction skips the next sequential word. Carry is unchanged and *overflow* is 0.

NOTE: *This instruction is indivisible.*

Narrow Increment and Skip if Zero (Extended Displacement)

XNISZ *index,displacement*

1	0	0	INDEX	1	1	0	0	0	1	1	1	0	0	1	@	DISPLACEMENT
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16 17 ... 31

Calculates the effective address, *E*. Increments the 16-bit contents of the location specified by *E*. If the incremented result is equal to zero, then the instruction skips the next sequential word. Carry is unchanged and *overflow* is 0.

NOTE: *This instruction is indivisible.*

Narrow Load Accumulator (Extended Displacement)

XNLDA *ac,index,displacement*

1	INDEX	AC	0	1	1	0	0	0	0	1	0	0	1	@	DISPLACEMENT
0	1 2	3 4	5	6	7	8	9	10	11	12	13	14	15	16	17 ... 31

Loads a value into an accumulator.
The instruction calculates the effective address, *E*. Uses *E* as the address of a 16-bit value. Loads this 16-bit value into the specified accumulator, then sign extends the value to 32 bits. Carry is unchanged and *overflow* is 0.

Narrow Multiply Memory Word (Extended Displacement)

XNMUL *ac,index,displacement*

1	INDEX	AC	0	0	0	1	0	0	1	1	0	0	0	@	DISPLACEMENT
0	1 2	3 4	5	6	7	8	9	10	11	12	13	14	15	16	17 ... 31

Multiplies an integer in an accumulator by an integer in memory.

The instruction calculates the effective address, *E*. Multiplies the 16-bit, signed integer contained in the location referenced by *E* by the signed integer contained in bits 16–31 of the specified accumulator. If the result is outside the range of -32,768 to +32,767 inclusive, sets *overflow* to 1; otherwise, *overflow* is 0. Sign extends the result to 32 bits and places the result in the specified accumulator. The contents of the referenced memory location and carry remain unchanged.

Narrow Store Accumulator (Extended Displacement)

XNSTA *ac,index,displacement*

1	INDEX	AC	0	1	1	0	0	0	1	1	0	0	1	@	DISPLACEMENT
0	1 2	3 4	5	6	7	8	9	10	11	12	13	14	15	16	17 ... 31

Stores the contents of an accumulator into memory.

The instruction calculates the effective address, *E*. Stores a copy of the 16-bit contents of the specified accumulator in the location specified by *E*. Carry is unchanged and *overflow* is 0.

Narrow Subtract Memory Word (Extended Displacement)

XNSUB *ac,index,displacement*

1	INDEX	AC	0	0	0	0	1	0	1	1	0	0	0	@	DISPLACEMENT
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17 ... 31

Subtracts an integer in memory from an integer in an accumulator.

The instruction calculates the effective address, *E*. Subtracts the 16-bit integer contained in the location referenced by *E* from the integer contained in bits 16–31 of the specified accumulator. Sign extends the result to 32 bits and stores it in the specified accumulator. Sets carry to the value of ALU carry, and *overflow* to 1 if there is an ALU overflow. The contents of the specified memory location remain unchanged.

Extended Operation

XOP0 *acs,acd,operation #*

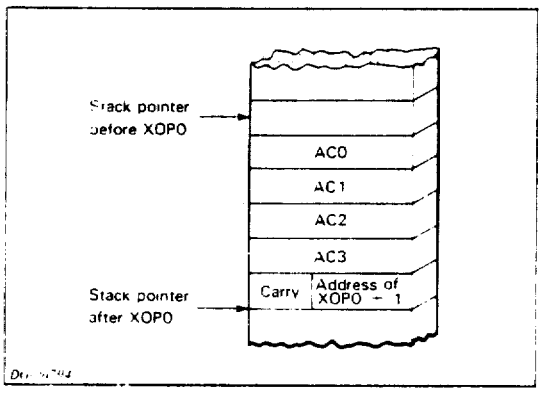
1	ACS	ACD	1	1	0	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	op#
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24 ... 31

Pushes a return block onto the narrow stack and transfers control to an extended operation procedure.

The instruction pushes a return block onto the narrow stack. Places the address in the narrow stack of ACS into AC2; places the address in the narrow stack of ACD into AC3. Memory location 44_8 must contain the **XOP0** origin address, the starting address of a 40_8 word table of addresses. These addresses are the starting location of the various **XOP0** operations.

The instruction adds the operation number in the **XOP0** instruction to the **XOP0** origin address to produce the address of a double word in the **XOP0** table. Fetches that word and treats it as the intermediate address in the effective address calculation. After the indirection chain, if any, has been followed, the instruction places the effective address in the program counter. The contents of carry, AC0, AC1, and the **XOP0** origin address remain unchanged. *Overflow* is 0.

The format of the return block pushed by the instruction is as follows:



This return block is designed so that the **XOP0** procedure can return control to the calling program via the *Pop Block* instruction.

Exclusive OR

XOR *acs,acd*

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

Forms the logical exclusive OR of the contents of bits 16-31 of ACS and the contents of bits 16-31 of ACD and places the result in bits 16-31 of ACD. Sets a bit position in the result to 1 if the corresponding bit positions in the two operands are unlike; otherwise, the instruction sets result bit to 0. The contents of ACS and carry remain unchanged. *Overflow* is 0.

Bits 0-15 of the modified accumulator are undefined after completion of this instruction.

Exclusive OR Immediate

XORI *i,ac*

1	0	1	AC		1	1	1	1	1	1	1	1	0	0	0	IMMEDIATE
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16 ... 31

Forms the logical exclusive OR of the contents of the immediate field and the contents of bits 16-31 of the specified accumulator and places the result in bits 16-31 of the specified accumulator. Carry remains unchanged and *overflow* is 0.

Bits 0-15 of the modified accumulator are undefined after completion of this instruction.

Push Address (Extended Displacement)

XPEF *index, displacement*

1	0	0	INDEX		1	1	0	0	0	1	0	1	0	0	1	@	DISPLACEMENT
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17 ... 31

Calculates the effective address, *E*. Pushes *E* onto the wide stack, then checks for stack overflow. Carry is unchanged and *overflow* is 0.

Push Byte Address (Extended Displacement)

XPEFB *index, displacement*

1	0	1	INDEX		1	1	0	0	0	1	0	1	0	0	1	@	DISPLACEMENT
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17 ... 31

Calculates a 32-bit byte address. Pushes this byte address onto the wide stack, then checks for stack overflow. Carry is unchanged and *overflow* is 0.

Push Jump (Extended Displacement)

XPSHJ *index,displacement*

1	0	0	INDEX		1	1	0	0	0	0	1	1	0	0	1	@	DISPLACEMENT
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17 ... 31

Calculates the effective address, *E*. Pushes the current 31-bit current value of the PC plus two onto the wide stack. Loads the PC with *E*. Checks for stack overflow. Carry is unchanged and *overflow* is 0.

NOTE: *The address pushed onto the wide stack will always reference the current segment.*

Vector on Interrupting Device (Extended Displacement)

XVCT

1	1	0	0	0	1	1	1	0	0	0	0	1	0	0	1	@	DISPLACEMENT
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17 ... 31

When a device requests an interrupt, transfers control to the appropriate interrupt sequence. Carry is unchanged and *overflow* is 0.

The instruction interprets the displacement field as an absolute address in the current segment. See the chapter on interrupt processing for a complete description of this instruction.

NOTE: *This is a privileged instruction.*

Wide Add Accumulator to Memory Word (Extended Displacement)

XWADD *ac,index,displacement*

1	INDEX	AC	0	0	1	0	0	0	1	1	0	0	0	@	DISPLACEMENT
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17 ... 31

Adds an integer contained in memory to an integer contained in an accumulator.

The instruction calculates the effective address, *E*. Adds the 32-bit integer contained in the location specified by *E* to the 32-bit integer contained in the specified accumulator. Loads the result into the specified accumulator. Sets carry to the value of ALU carry, and *overflow* to 1 if there is an ALU overflow. The contents of the referenced memory location remain unchanged.

Wide Divide Memory Word (Extended Displacement)

XWDIV *ac,index,displacement*

1	INDEX	AC	0	0	1	1	1	0	1	1	0	0	0	@	DISPLACEMENT
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17 ... 31

Divides an integer in an accumulator by an integer in memory.

The instruction calculates the effective address, *E*. Sign extends the 32-bit integer contained in the specified accumulator and divides it by the 32-bit integer contained in the location specified by *E*. If the quotient is within the range of -2,147,483,648 to +2,147,483,647 inclusive, the instruction loads it into the specified accumulator. If the quotient is outside this range, the instruction does not load it into the specified accumulator. The contents of the referenced memory loaction and carry remain unchanged.

If the divisor in memory is zero, or if the dividend is the largest negative number and the divisor is -1, the instruction sets *overflow* to 1 and leaves the specified accumulator unchanged. Otherwise, *overflow* is 0.

Wide Decrement and Skip if Zero (Extended Displacement)

XWDSZ *index,displacement*

1	0	1	INDEX	1	1	0	0	0	1	1	1	0	0	1	@	DISPLACEMENT
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17 ... 31

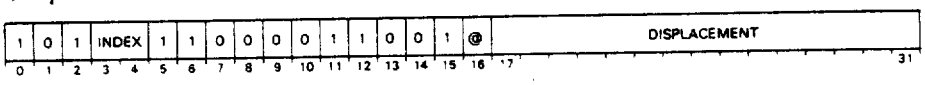
Calculates the effective address, *E*. Decrements the 32-bit contents of the location

addressed by *E* by one. If the decremented result is equal to zero, then the instruction skips the next sequential word. Carry is unchanged and *overflow* is 0.

NOTE: *This instruction executes in one indivisible memory cycle if the word to be decremented is located on a double word boundary.*

Wide Increment and Skip if Zero (Extended Displacement)

XWISZ *index,displacement*

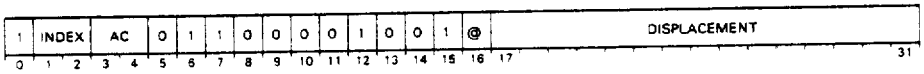


Calculates the effective address, *E*. Increments the 32-bit contents of the location addressed by *E* by one. If the incremented result is equal to zero, then the instruction skips the next sequential word. Carry is unchanged and *overflow* is 0.

NOTE: *This instruction executes in one indivisible memory cycle if the word to be incremented is located on a double word boundary.*

Wide Load Accumulator (Extended Displacement)

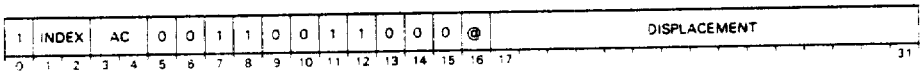
XWLDA *ac,index,displacement*



Calculates the effective address, *E*. Loads a copy of the 32-bit word addressed by *E* into the specified accumulator. Carry is unchanged and *overflow* is 0.

Wide Multiply Memory Word (Extended Displacement)

XWMUL *ac,index,displacement*



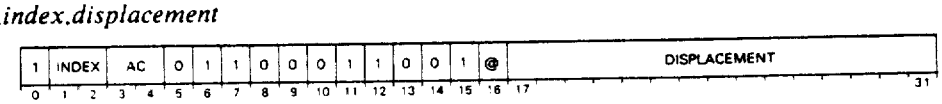
Multiplies an integer in an accumulator by an integer in memory.

The instruction calculates the effective address, *E*. Multiplies the 32-bit, signed integer contained in the location referenced by *E* by the 32-bit, signed integer contained in the specified accumulator. Loads the 32 least significant bits of the result into the specified accumulator.

If the result is within the range of -2,147-483,648 to +2,147,483,647 inclusive, the instruction sets *overflow* to 0; otherwise, *overflow* is 1. The contents of the referenced memory location and carry remain unchanged.

Wide Store Accumulator (Extended Displacement)

XWSTA *ac,index,displacement*



Calculates the effective address, *E*. Stores a copy of the 32-bit contents of the specified accumulator in the memory location specified by *E*. Carry is unchanged and *overflow* is 0.

Wide Subtract Memory Word (Extended Displacement)

XWSUB *ac,index,displacement*

1	INDEX	AC	0	0	1	0	1	0	1	1	0	0	0	@	DISPLACEMENT
0	1 2	3 4	5	6	7	8	9	10	11	12	13	14	15	16	17 31

Subtracts an integer contained in memory from an integer contained in an accumulator.

The instruction calculates the effective address, *E*. Subtracts the 32-bit integer contained in the location referenced by *E* from the 32-bit integer contained in the specified accumulator. Loads the result into the specified accumulator. Sets carry to the value of ALU carry, and *overflow* to 1 if thee is an ALU overflow. The contents of the specified memory location remain unchanged.

Zero Extend

ZEX *acs,acd*

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

Zero extends the 16-bit integer contained in ACS to 32 bits and loads the result into ACD. The contents of ACS remain unchanged, unless ACS equals ACD. Carry is unchanged and *overflow* is 0.

APPENDIX C

ALU TESTS

```
Mnem   | Value | Description
------+-------+------------
CARRY  |  10   | CARRY = 1
LINK   |  11   | LINK = 1
RND    |  12   | FPSR8 = 1
TE     |  13   | FPSR5 = 1
CRY    |  14   | CRY0 = 1 if FLAG0 = 1; CRY16 = 1 if FLAG0 = 0
OVFL   |  15   | OVR0 = 1 if FLAG0 = 1; OVR16 = 1 if FLAG0 = 0
ALUNZ  |  16   | ALU<0-31> <> 0
SEQG   |  17   | SIGN xor OVFL.  For SUB, tests  S < R.
       |       |  For RSB, tests  R < S.
D0     |  18   | D<0> = 1
D28    |  19   | D<28> = 1
D29    |  1A   | D<29> = 1
D30    |  1B   | D<30> = 1
D31    |  1C   | D<31> = 1
COM1   |  1D   | Commercial test - validates data types.
SEX    |  1E   | IF D2ND = PMD previous cycle,
       |       |  Test (FAU16 or FAU0) equal to 0 if the most
       |       |  previous memory start was a (word or double word
       |       | IF D2ND = NSHL &  SHFT = SEX, Test TLCH16=0
       |       | ELSE always true.
TL31   |  1F   | TLCH31 = 1
FNOP   |  20   | Result of floating compare if CMPX coded previous:
       |       |  single precision (FLAG5 = 0): tests BEXP-AEXP > 7
       |       |  double precision (FLAG5 = 1): ABS(BEXP-AEXP) > 15
       |       | else tests for normalized mantissa on ALU<0-31>
D24    |  21   | D<24> = 1
       |  22   | Unassigned
```

IOSKP	23	Comercial test - used to detect I/O skips.
COM2	24	Commercial Tests - Validates data types.
SIGN	25	ALU0 = 1 if FLAG0 = 1; ALU16 = 1 if FLAG0 = 0
MR30	26	MREG<30> = 1
COMP	27	ACSR<2,3> = ACDR<2,3> (After any inc or dec)
	28	Unassigned
	29	"

NOTE:

Conditions reflect state at the conclusion of the previous micro-instruction unless stated differently. Exception: Since ZLCH and TOVR LATCH hold data the second half for split cycle modes, then SIGN, CRY, OVFL and ALUNZ are effectively tested at the conclusion of the first half cycle during these modes.

ATU, INSTRUCTION PROCESSOR, and IOC TESTS

Mnem	Value	Description
VWP	30	LA<1-3> < ESR
CBLK	31	LA<29,30,31> <> 7
MMPU	32	MMPU on
ATU	33	ATU on
PURGE	34	ATU busy purging
PGVLD	35	Page valid, resident, not protected (previous start)
PGEXT	36	Page invalid, non-resident, or execute not allowed.
PTVLD	37	Page Table invalid.
PROT	38	A protection fault has occurred
SBVLD	39	SBR valid, and page table depth ok.
GTCRE	3A	bits 1-3 of LA greater than CRE
EQCRE	3B	bits 1-3 of LA equal to CRE
LTCRE	3C	bits 1-3 of LA less than CRE
	3D	Reserved
	3E	Reserved
RING0	3F	CRE <> 0

U-SEQUENCER TESTS

Mnem	Value	Description
TRUE	0	Always true
INTR	1	Interrupt pending
CPU31	2	CPU31 = 0
IOB	3	IO not busy. Test the 3rd cycle after the command IOS
		Continue testing (and reading if data in) until true.
FLAG0	4	FLAG0 = 1
FLAG1	5	FLAG1 = 1
FLAG2	6	FLAG2 = 1
FLAG3	7	FLAG3 = 1
FLAG4	8	FLAG4 = 1
FLAG5	9	FLAG5 = 1
FLAG6	A	FLAG6 = 1
FLAG7	B	FLAG7 = 1
TFLG6	C	Test FLAG6 = 1, then Toggle FLAG 6
TFLG7	D	Test FLAG7 = 1, then Toggle FLAG 7
LCTR	E	Test CNTR<2-5> <> 0; then CNTR<2-5> := CNTR<2-5> - 1;
		If LCNT is coded, this test is on the value loaded
		and no decrement occurs.
USMT	F	microstack empty

APPENDIX D

Absolute Address Conditional

```
 0        2        3        4        9 10      19
+---------+-------------+---------+---------+
| MODE(3) | POLARITY(1) | TEST(6) | AA(10)  |
+---------+-------------+---------+---------+
```

NOTE: The Absolute Address Field (AA) is extended to a full 12 bit address by concatenating the most significant 2 bits of the current PC (termed the Page bits) to the most significant side (left) of the AA:

```
 0        1 2       11
+---------+---------+
| PAGE(2) | AA(10)  |
+---------+---------+
```

Instructions:

Mode	Mnem.	Explanation	True Action	False Action
000	CJMP	Conditional Jump	PC <-- AA(10)	PC <-- PC+1
001	CJSR	Cond. Jump subroutine	PC <-- AA(10) Push PC+1	PC <-- PC+1
010	CRTN	Conditional Return	PC <-- TOS Pop	PC <-- AA(10)
011	TWB	Two Way branch	PC <-- AA(10) Pop	PC <-- TOS Pop
100	CPD	Load stack from CPD and Jump if True.** -primarily used for stack restore after context save.	PC <-- AA(10) Push CPD*	PC <-- PC+1 Push CPD

LCNT, CPUP and LOOP

```
 0   2        3        4        9 10       11 12      19
+-----+-------------+---------+----------+----------+
| 110 | POLARITY(1) | TEST(6) |  Mod2(2) | OTHER(6)|
+-----+-------------+---------+----------+----------+
```

Instructions:

Mod2	Mnem.	Explanation	True Action	False Action
00	LCNT	Load the 4 bit loop Counter with the ones complement of the 4 least significant bits of the the OTHER field	PC <-- PC+1	PC <-- TOS pop stack
01	CPOP	Conditional Pop; OTHER field may be used for loading the SPAR (Scratch Pad Addr. Reg.)	PC <-- PC+1	PC <-- TOS pop stack
10	LOOP	Conditionally Jump to TOS or continue and pop. OTHER field used as in CPOP.	PC <-- PC+1 pop stack	PC <-- TOS

Flag Controls

Instructions:

POP	SET	Mnem.	Explanation	True Action	False Action
0	00	F01	Control Flag Set 0-1: FLAG 0 and FLAG 1 using CNTL field	PC <-- PC+1	PC <-- TOS pop stack
0	01	F23	Control Flag Set 2-3: FLAG 2 and FLAG 3 using CNTL field	PC <-- PC+1	PC <-- TOS pop stack
0	10	F45	Control Flag Set 4-5: FLAG 4 and FLAG 5 using CNTL field	PC <-- PC+1	PC <-- TOS pop stack
0	11	F67	Control Flag Set 6-7: FLAG 6 and FLAG 7 using CNTL field	PC <-- PC+1	PC <-- TOS pop stack
1	00	FP01	Control Flag Set 0-1: FLAG 0 and FLAG 1 using CNTL field	PC <-- PC+1 pop stack	PC <-- TOS pop stack
1	01	FP23	Control Flag Set 2-3: FLAG 2 and FLAG 3 using CNTL field	PC <-- PC+1 pop stack	PC <-- TOS pop stack

1	10	FP45	Control Flag set 4-5: FLAG 4 and FLAG 5 using CNIL field	PC <-- PC+1 pop stack	PC <-- TOS pop stack
1	11	FP67	Control Flag Set 6-7: FLAG 6 and FLAG 7 using CNIL field	PC <-- PC+1 pop stack	PC <-- TOS pop stack

SPLIT

Mod3	FS	Mnem.	Explanation	True Action	False Action
00	0	SPLT01	Conditional branch to TOS pop without pop; also, control flag set 0-1 with CNIL field (see 1.4)	PC <-- PC+1	PC <-- TOS
00	1	SPLT23	Conditional branch to TOS pop without pop; also, control flag set 2-3 with CNIL field (see 1.4)	PC <-- PC+1	PC <-- TOS

Context Restore Instructions:

Mod3	Mnem.	Explanation	True Action	False Action
01	RCF	Restore 6 bit loop counter and all 6 flags from CPU bits 20-31 for counter; bits 8-15,24,25 for flags	PC <-- PC+1	PC <-- TOS pop stack
10	RESI	If FALSE; Restore after context (sets RESI FF)	PC <-- PC+1	PC <-- TOS pop stack

Unconditional Instructions:

Unconditional with 12 bit AA:

```
 0   2 3              5 6   7 8       19
+-----+--------------+----+---------+
| 111 | Ext. Mode(5) | xx | AA(12)  |
+-----+--------------+----+---------+
```

Instructions:

Ext. Mode	MNEM	Description	Actions
000	LEAP	12 bit Jump	PC <-- AA(12)
001	LSR	Lead to subroutine	PC <-- AA(12); Push PC+1
010	PUSH	Push 12 bit AA	PC <-- PC+1; Push AA(12)
011	LPOP	Lead and Pop	PC <-- AA(12); Pop (STUS is loaded)
100	LPSH	Lead and Push	PC <-- AA(12); Push AA(12)
101	CTXL	Context Lead	PC <-- AA(12); Push [illegible]

Unconditional Dispatches:

```
  0   2 3              5 6  7 8     16 17  18      19
 +-----+--------------+----+--------+---+----------+
 | 111 | Ext. Mode(3) | XX | AA(9)  | X | DSRC(2)  |
 +-----+--------------+----+--------+---+----------+
```

NOTE: The Absolute Address Field (AA) is extended to a full 12 bit address by concatenating 3 zeros to the least significant (right hand) side of the AA:

```
Formed AA:  0        8 9  11
            +--------+-----+
            | AA(9)  | 000 |
            +--------+-----+
```

Instructions:

Ext. Mode	MNEM	Description	Actions
110	DSPA	Unconditional Dispatch	PC <-- Dispatch*
111	DSPR	Dispatch to subroutine	PC <-- Dispatch*; Push PC+1

APPENDIX E

Description of special modes:

MPY mode: Code ALUS = DZ, ALUOP = RSB. The following values will be forced:

Previous half cycle PMR<30,31>	MLTC	forced this cycle ALUS	ALUOP
00	0	DZ	ADD
00	1	DA	ADD
01	0	DA	ADD
01	1	DQ	ADD
10	0	DQ	RSB
10	1	DA	RSB
11	0	DA	RSB
11	1	DZ	RSB

In both unsigned and signed multiply, the first operand (D bus) is sign-extended two bits to 34 bits (Except when CRYINS = 0 and DaOS <> MACC, in which case it is zero-extended) The second operand (A, Z, or Q) is zero-extended to 34 bits for unsigned multiply and sign-extended for signed multiply. (The extension of the second operand is made either from the 32 bit AREG, 0, or from the 33 bit WdIT,QREG). Thus, the result has 34 bits, with DS<X,Y> the two MSb's.

MLTC and PMR are clocked both half cycles in MPY mode.

DIV mode: The forced value of ALUOP is dependent upon TCRYY (loaded with the divide carry) as shown below. For all other modes, TCRYY gets 0. ALUOP should be coded as SUB.

TCRYY	ALUOP
0	ADD
1	SUB

The first operand, the dividend, is in BREG. It is extended one bit by LINK. The second operand, the divisor in AREG, is zero-extended one bit. Thus, the result has 33 bits with DS<Y> the MSb.

PRESC mode: To get prescaled mantissa into QREG, code BREG - AREG -> ALU and CMPX during the previous u-instruction to set up prescale hardware, and code + PASS - DZ JR QREG + this cycle. PRESC hardware forces ZA if FACD >= FACS, else ZB, and right shifts the # of nibbles equal to AbS(BX - AX)mod 8.

NORM mode: Code ADNM (MATH:FPOP random, for proper adjustment of the exponent.
Code PASS second half for proper shifting of the mantissa. Shift is RSHN, PASS, LSH1,LSH2, LSH3, LSH4, for MUF, 0, 1, 2, 3, 4 leading nibbles of zero.

All SPLIT CYCLE modes: TOVR latch, ZLCH, and TLCH latch and hold data during the 2nd cycle.

APPENDIX F

SHFT FIELD

The SHFT field has two basic functions: 1) Control of the inputs for bit shifts into QREG or BREG, and 2) Control of the nibble shift hardware which operates upon TLCH and outputs to the D buss. Which function the SHFT field specifies is also governed by the D2ND field.

3.6.1 NSHR type shifts

If the D2ND field contains NSHR, then the bit inputs are all one, and the nibble hardware is controled by the SHFT field as shown in the following chart:

SHFT Field Mnem	Value	TLCH loaded 1st half: A	B	C	D	E	F	G	H
		D Bus source during second half:							
RSH1	0	0	A	B	C	D	E	F	G
RSH2	1	0	0	A	B	C	D	E	F
RSH3	2	0	0	0	A	B	C	D	E
RSH4	3	0	0	0	0	A	B	C	D
RSH5	4	0	0	0	0	0	A	B	C
RSH6	5	0	0	0	0	0	0	A	B
RSH7	6	0	0	0	0	0	0	0	A
BEN6	7	0	0	0	0	0	0	G	H
BEN5	8	0	0	0	0	0	0	F	G
BEN4	9	0	0	0	0	0	0	E	F
BEN3	A	0	0	0	0	0	0	D	E
BEN2	B	0	0	0	0	0	0	C	D
BEN1	C	0	0	0	0	0	0	B	C
RSHN	D	1	A	B	C	D	E	F	G
EXP	E	(S & EXP)	A	B	C	D	E	F	
@ACS	F	RIGHT SHIFT (ACSR<2,3> + 1)							

Notes: EXP - D0 := Fsign, D1 := EXP1', D<2-7> := EX<2-7>, and D<8-31> gets TLCH right shifted two nibbles.

@ACS - D<0-31> gets TLCH shifted right the number of times indicated by ACSR<2,3> + 1 with zeroes shifted in.

RSHN only works for split cycles. In Fxxx modes, it will act like RSH1.

NSHL type shifts

If the D2ND field contains NSHL, then the bit inputs are all one, and the nibble hardware is controlled by the SHFT field as shown in the table below:

SHFT Field Mnem	Value	TLCH loaded 1st half: A	B	C	D	E	F	G	H
		D Bus source during second half:							
LSH1	0	B	C	D	E	F	G	H	0
LSH2	1	C	D	E	F	G	H	0	0
LSH3	2	D	E	F	G	H	0	0	0
LSH4	3	E	F	G	H	0	0	0	0
SWPB	4	C	D	A	B	G	H	E	F
SWPW	5	E	F	G	H	A	B	C	D
ZEXT	6	0	0	0	0	E	F	G	H
SEX	7	S	S	S	S	E	F	G	H
BSEX	8	S	S	S	S	S	S	G	H
ROTL	9	C	D	E	F	G	H	A	B
reserved	A								
reserved	B								
FPSR	C	(FPSR 0 - 15)				0	0	0	0
DECC	D	0	0	0	0	0	0	0	H'
@FLG4	E	@ACS; LEFT,RIGHT FOR FLAG4 = 0,1							
@ACS	F	LEFT SHIFT (ACSR<2,3> + 1)							

Notes: @FLG4 - Does a right or left shift depending upon the value of FLAG4, and shifts the number of nibbles indicated by ACSR<2,3> + 1.

FPSR - The upper 16 bits of the Floating Point Status Register is placed on D<0-15>, and D<16-31> gets zeroes.

DECC produces a decimal correct on the least significant nibble.

BIT SHIFTS

If the D2ND field contains any value except NSHR or NSHL, then the SHFT field controls the bit shift inputs as per the following chart. If D2ND contains NSHR or NSHL then the bit inputs are as specified by the "ONE" mnemonic in the following chart.

	FLAGO = 1 Wide operation				FLAGO = 0 Narrow operation			
ALUD field:	RSHB or RSQB	RSQB	LSHB or LSQB	LSQB	RSHB or RSQB	RSQB	LSHB or LSQB	LSQB
SHFT mnem Val	R0IO	Q0IO	R31IO	Q31IO	R16IO	Q16IO	R31IO	Q31IO
ZZ 0	0	0	0	0	0	0	0	0
SSH 1	0	ALU31	0	ALU0	0	ALU31	0	ALU16
DRT 2	Q31	ALU31	Q0	ALU0	NVACRY	ALU31	NVACRY	ALU16
DSH 3	ALU0	ALU31	Q0	0	ALU16	ALU31	Q16	0
LNK 4	LINK	ALU31	LINK	ALU0	LINK	ALU31	LINK	ALU16
CAR 5	CARRY	ALU31	CARRY	ALU0	CARRY	ALU31	CARRY	ALU16
SRT 6	ALU31	ALU31	ALU0	Q0	ALU31	ALU31	ALU16	Q16
CRY 7	CRY0	ALU31	Q0	CRY0	CRY16	ALU31	Q16	CRY16
ONE 8	1	1	1	1	1	1	1	1
9,A	Reserved							
CRYY B	1	ALU31	Q0	CRYY	1	ALU31	Q16	CRYY
C-F	Reserved							

Note: ALC's code DRT in the SHFT field. NVACRY = ALCCRY XOR TCRY16

APPENDIX G

CRYINS FIELD

ALU carry input select field. The use of this field is governed by the CSM RAND fields. There are 4 types of usage.

Type0

All modes except SFIXP, FFIXP, MPY, and DIV are type 0.

Mnem	Value	Description
Z	0	CRYIN = 0
H	1	CRYIN = 1

Type1

Modes SFIXP and FFIXP. This field combines with bit 0 of the rand field. Rand type FIXP is used with type1 CRYINS.

CRYINS Mnem	CRYINS Value	CEXT Mnem	CEXT (RAND<0>) Value	Description
Z	0	N	0	CRYIN = 0
H	1	N	0	CRYIN = 1
Z,C	0	CARRY	1	CRYIN = CARRY
H,B	1	CARRY	1	CRYIN = CARRY'

Type2

MPY mode.

CRYIN = MLTC

Mnem	Value	Description
U	0	Unsigned multiplication
S	1	Signed multiplication

Type3

DIV mode.

CRYIN = TCRYY

Mnem	Value	Description
	0	Unassigned
	1	Unassigned

APPENDIX H

Type Math

ROUND - Floating point Round control.

Mnem	Value	Description
NOP	0	
FRND	1	Add the ROUND BIT to the ALU result by forcing CRY24, and force 0's into bits 24-31 of NSHL, NSHR, or PASS

Note: FRND overides TRNC. Do not code FRND with PASS in a full cycle mode. Do not code FRND with NSHR: BEN<1-6>, or NSHL: ZEXT, SEX, BSEX, FPSR, or DECC.

```
ᵖOP - Floating Point exponent, multiplication, and truncation.
---

  Mnem   | Value | Description
  ------+-------+------------
  NOP    |   0   |
         |   1   | Unassigned
  LDQB   |   2   | QBIT := TCRY0
  TRNC   |   3   | Truncate. Force 0's into bits 28-31 of PASS if
         |       |   FPSR<6> = 0. (Only code with PASS & Split cycle)
  SHMR   |   4   | Shift MREG right 2 bits each half-cycle. MLTC and PMR
         |       |   are clocked both half cycles.
  MSHQ   |   5   | LDQB and SHMR
  MRAX   |   6   | LDMR and ADDX
  LDMR   |   7   | Load MREG each half for split cycle, once for full
         |       | cycle.  Do not read MREG concurrently.
  LDEX   |   8   | Load exp.   BX<0-1> := EXP<0-1> = TLCH1'
         |       |             BX<2-7> := EXP<2-7> = TLCH<2-7>
  ADDX   |   9   | Add exp.    BX<0-7> := EXP<0-7> = BX<0-7> + AX<0-7>
  SUBX   |   A   | Sub  "      BX<0-7> := EXP<0-7> = BX<0-7> - AX<0-7>
  CMPX   |   B   | Compare "   EXP<0-7> = BX<0-7> - AX<0-7>
  MOVX   |   C   | Move exp.   BX<0-7> := EXP<0-7> = AX<0-7>
  ADCN   |   D   | Add const. BX<0-7> := AX<0-7> + NAC<12-19>,
         |       |      EXP<0-7> = AX<0-7>
  ADNM   |   E   | Add norm.   BX<0-7> := AX<0-7> + NORM OFFSET<0-7>
         |       |      EXP<0-7> = AX<0-7>
  LDCN   |   F   | BX<0-7> := NAC<12-19>, EXP<0-7> := AX<0-7>

Notes:   EXP<0-7> defaults to AX if no exponent u-order is coded.

         NORM OFFSET is 1, 0,-1,-2,-3,-4, for MOF, 0,1,2,3, or  4
         leading nibbles of zeroes of ALU<0-31>. (MUF can not occur
         during logical operations).

ype Math cont.


ISC - Scratchpad, Sign, Normalization and SPAR operations.
---

  Mnem   | Value | Description
  ------+-------+------------
  NOP    |   0   |
  RDAA   |   1   | SPAD addressed by NAC<12-19>.
  RACS   |   2   | SPAD addressed by 111000, OP1<2-3>   Used to address

  RACD   |   3   | SPAD addressed of 000000,OP1<2-3>.FPACs.to address
         |       | low order half of double precision FPAC's.
  WRSP   |   4   | SPAD addressed by SPAR<0-7>; SPAD<0-31> := CPD<0-31>
  WRAA   |   5   | SPAD addressed by NAC<12-19>; SPAD<0-31> := CPD<0-31>
  WANC   |   6   | WRAA*NOCR
  WACD   |   7   | SPAD addressed by 111000, OP2<2-3> ;
         |       | SPAD<0-31> := CPD<0-31>  Used to write low-order
         |       | half of double precision FPAC's.
  RSBR   |   8   | SPAD addressed by 11101, LAR<1-3>, no load LAR
  LDUP   |   9   | ATU OP1<2-3> := CPD<30-31>, OP2<2-3> := CPD<26-27>
         | A - F | Unassigned
  LDSN   |   10  | Load sign   Sign(BREG) := FSIGN = TLCH0
  XORS   |   11  | Sign(BREG) := FSIGN = Sign(BREG) XOR Sign(AREG))
  SMOV   |   12  | Move sign   Sign(BREG) := FSIGN = Sign(AREG)
  SCOM   |   13  | Move complement of sign  Sign(BREG) := FSIGN =
         |       |  Sign(AREG)'
```

Type Math cont.

NOCR	14	Force a zero to the carry input of the normalization prom.
	15	Unassigned
SMNC	16	SMOV and NOCR
SCNC	17	SCOM and NOCR
LDFS	18	FPSR1 := (FPSR1 OR OVF); FPSR2 := (FPSR2 OR UNF);LDZN
XSFS	19	LDFS and XORS
SMFS	1A	LDFS and SMOV
SCFS	1B	LDFS and SCOM
LDSS	1C	LDSN and LDZN
LDZN	1D	FPSR7 := FSIGN; FPSR6 := 1 if ALU = 0, else := 0
	1E	Unassigned
	1F	Unassigned

APPENDIX I

Type Fixp

MISC1 - ALU loading control

Mnem	Value	Description
NOP	0	Unassigned
DEC1	1	Force loading of least significant nibble of BREG if ALUD = NLD. (By forcing bit 1 of ALUD field to 1)
LALC	2	ALC conditional load.
ABSV	3	Conditional loading of BREG and QREG a) FLAG0 = 1 loading enabled iff ALU0 = 0 b) FLAG0 = 0 loading enabled iff ALU15 = 0
NARC	4	Override FLAG0 (wide op) and force narrow ALU Status. Effects CRY, SIGN, OVFL, SEGG, ALUNZ tests and LOVC, LOCY, & LCRY randoms. Does not effect loading BREG or bit shifts.
	5-F	Unassigned

MISC2 - Carry, Overflow and Status randoms. (5 bits).

Mnem	Value	Description
NOP	0	
RDAA	1	SPAD addressed by NAC<12-19>
RACS	2	SPAD addressed by 111000, OP1<2-5> Used to address low order half of double precision FPAC's.
RACD	3	SPAD addressed by 111000, OP2<2-5> Used to address low order half of double precision FPAC's.
WRSP	4	SPAD addressed by SPAR<0-7>; SPAD<0-31> := CPD<0-31>
WRAA	5	SPAD addressed by NAC<12-19>; SPAD<0-31> := CPD<0-31.
	6	WRAA
WACD	7	SPAD addressed by 111000, OP2<2-5> ; SPAD<0-31> := CPD<0-31> Used to write low-order half of double precision FPAC's.
RSBR	8	SPAD addressed by 11101, LAR<1-5>, no load LAR
LDOP	9	OP1<2-5> := CPD<30-31>, OP2<2-5> := CPD<26-27>
	A - F	Unassigned
LPSR	10	PSR<0-2> := D<0-2>, Do not have uncompleted memory operation, because the next cylce may trap.

COVR	11	PSR1 := 0, Clear OVR
SOVR	12	PSR1 := 1, Set OVR (see note)
LOVC	13	If FLAG0 = 0, PSR1 := OVR15 + PSR1; CARRY := CRY16
		If FLAG0 = 1, PSR1 := OVR0 + PSR1; CARRY := CRY0
COVR	14	PSR0 := 0 Clear OVR
SOVR	15	PSR0 := 1 Set OVR
LDCY	16	CARRY := D0, D16 for FLAG0 = 1,0
LCRY	17	CARRY := CRY
	18	Unassigned
LFPS	19	FPSR<1-4> := DS<1-4>
SMOF	1A	Set mantissa overflow FPSR4 := 1
SDVZ	1B	Set divide by zero FPSR5 := 1
DCRY	1C	CARRY := Decimal carry, code NSHL DECC concurrent
ALC	1D	CARRY := ALC shift; Enable ALC skip logic.
CLRC	1E	CARRY := 0 Clear CARRY
SETC	1F	CARRY := 1 Set CARRY

NOTE: If LSPR, SOVR, LOVC, or SOVR cause both OVR and OVR to be 1, a fixed point overflow trap will occur to uloc 1A. At the end the cycle at uloc 1A, OVR will be set to 0.

APPENDIX J

Type Gen

REGS - General and ACS, ACD operations

Mnem	Value	Description
NOP	0	Unassigned
	1	Unassigned
	2	Unassigned
LDMR	3	MREG<0-31> := D<0-31>; Do not read MREG
	4	Unassigned
NCPDR	5	No-load CPDR
FCY	6	Force CARRY into CPDR if CPDR is loaded.
		If FLAG1 = 0 CPDR<16> = CARRY
		If FLAG1 = 1 CPDR<0> = CARRY
RPSR	7	CPDR<0-2> = PSR<0-2> if CPDR is loaded.
SGR7	8	ACSR := GR7
LACS	9	ACSR<0-3> := D<28-31>, SRC = D<28-31>
LACD	A	ACDR<0-3> := D<24-27>, DES = D<24-27>
LREG	B	LACS & LACD
INCS	C	Inc ACSR. ACSR<0-3> := ACSR<0-3> + 1 mod 4
DECS	D	Dec ACSR. ACSR<0-3> := ACSR<0-3> - 1 mod 4
INCD	E	Inc ACDR. ACDR<0-3> := ACDR<0-3> + 1 mod 4
DECD	F	Dec ACDR. ACDR<0-3> := ACDR<0-1> - 1 mod 4

SPAR - SPAR operations

Mnem	Value	Description
NOP	0	
APCRE	1	LA<1-3> = CRE Append CRE
SCPD	2	SPAR<0-7> := CPD<24-31>
SPAA	3	SPAR<0-7> := NAC<12-19>

SPAU - scratchpad operations

Mnem	Value	Description
NOP	0	
RUAA	1	SPAU addressed by IAC<12-17>
RALS	2	SPAU addressed by 1110000, SP1<2-5> used to address low-order half of double precision F-AC's.
RACD	3	SPAU addressed by 1110000, SP2<2-5> used to address low order half of double precision F-AC's.
WRSP	4	SPAU addressed by SPAP<0-7>; SPAU<0-31> := CPD<0-31>
WRAA	5	SPAU addressed by IAC<12-17>, SPAU<0-31> := CPD<0-31>
	6	Unassigned
WALS	7	SPAU addressed by 1110000, SP1<2-5>, SPAU<0-31> := CPD<0-31> used to write low-order half of double precision F-AC's.
RSBR	8	SPAU addressed by 1110000, SP1<2-5>, no load bus
LDCP	9	SP1<2-5> := CPD<30-31>, SP2<2-5> := CPD<28-29>
	A	Unassigned
	B	Unassigned
	C	Unassigned
	D	Unassigned
	E	Unassigned
	F	Unassigned

APPENDIX K

Type ATU

ATUO - ATU operations

Mnem	Value	Description
NOP	0	
AON	1	ATU on. Do not code IPOP concurrently
AOFF	2	ATU off. Do not code IPOP concurrently
LDCR	3	CRE<1-3> & ESR<1-3> := LA<1-3>. Don't start memory. Code RSbR next cycle to read ID & LEF bits to ATU
PRGA	4	Purge the ATU translation buffer.
LATUS	5	Restore ATU context from CPD bus. See note next pg.
RESR	6	ESR<1-3> := CRE<1-3>, Reset the ESR after DFR loops.
RMAX	7	Ring maximiztion check on LA (mem start not nec.)
LPTA	8	Low order page table addresses memory Previous read (MD<18-31>),LAR<13-21> -> PHY<8-31>
OPAD	9	Object page table addresses memory Previous read (MD<18-31>),LAR<22-31> -> PHY<8-31> If test condition true, load ATU Translation buffer Protection<0-2>,Translation<8-21> <- MD<2-4,18-31>; Set valid; Tag<2-14> <- LAR<2-14>; Buffer addressed by LA<1,15-21>. Use only during LAI trap.
SIO	A	Send IO Command or Data
LMAP	B	Protection<0-2>,Translation<8-21> <- CPD<2-4,18-31> Valid <- CPD0. Buffer addressed by LA<1,15-21>
WRRF	C	CPD<26-27> -> Mod,Ref bits addressed last cycle.
	D	Unassinged
DMAP	E	DUA MAP; Loads the MMPU data from the CPD bus.
CDR	F	CCP data; CDR<0-15> := CPD<16-31>.
MMPON	10	If user mode enabled, turn on ATU.
	11	Unassigned
	12	Unassigned
	13	Unassigned

```
HLT   |  14   | Force the IP to halt state to allow single step or
      |       |  continue.  CCP halt flag must be set.
PRS   |  15   | Parse IDR; Used after XCT or to clear an EFA after trap.
PLD   |  16   | ICP<1-31> := LA<1-31>, LENGTH := CPD<30,31>
ICXT  |  17   | XCT and inhibit LEF mode for XCTed opcode.
PST   |  18   | IP STRT: LA -> PC; CPA -> IP Physical Page, APCRE
PCN   |  19   | IP CANCEL
CAT   |  1A   | ICP -> LA; CPA -> IP Physical Page register
DISI  |  1B   | Disable interrupts one instruction
      |       |   Do not code an IPOP concurrently with DISI
DOFF  |  1C   | ION off.  (Do not code IPOP concurrently)
ION   |  1D   | ION on, DISI. (Do not code IPOP)
XCT   |  1E   | CPD<16-31> is inserted into the instruction stream.
      |       |   Code IPRS next cycle and IPOP the next.  The IP
      |       |   decode prom must contain 0 instruction length.
SKIP  |  1F   | Skip the next instruction. Takes 1 cycle to do.
      |       |   Do not code SKIP after an IPST.  Do not code a
      |       |   memory operation after a skip.
```

Type ATU cont.

ATU1 - More ATU operations

Mnem	Value	Description
NOP	0	
APCRE	1	LA<1-3> = CRE Append CRE to LA
DFR	2	Increment Defer counter, ESR := LA<1-3>
SPAA	3	SPAR<0-7> := NAC<12-19>

ATU2 - General

Mnem	Value	Description
NOP	0	
BYLCH	1	Force BYLCH regardless of memory start type
RDAA	2	SPAD addressed by NAC<12-19>
LDOP	3	OP1 := CPD<30-31>, OP2 := CPD<26-27>
WRSP	4	SPAD addressed by SPAR<0-7>; SPAD<0-31> := CPD<0-31>
WRAA	5	SPAD addressed by NAC<12-19>; SPAD<0-31> := CPD<0-31>
	6	Unassigned
	7	Unassigned

Note: ATU state for ATUD:LATUS and CPD Source:ATUD

Bits		Bits	
0	Was IPST*	13	Was single reference
1-3	0	14-15	0
4	Was PC Ref	16	EFA*
5	Was IC Ref	17-19	ESR<1-3>*
6	Enable Split*	20-22	0
7	0	23	Usr md intr
8	Write p	24,25	0
9-10	Mode<0-1>	26	Mod*
11,12	0	27	Ref*
		28-31	RF<0-3>*

* ATUD only

APPENDIX L

CPDS FIELD

Mnem	Value	Description
NOP	0	CPD<0-31> = 0, No load CPDR
ZERO	1	CPDR<0-31> := CPD<0-31> = 0
FPU1	2	Reserved for future use by Floating Point hardware.
FPU2	3	Reserved for future use by Floating Point hardware.
MSB	4	ALU Status Buffer
		CPDR<0-31> := CPD<0-31>. CPD<0-23> = 0,
		CPD<24-27> = ACDR<0-3>, CPD<28-31> = ACSR<0-3>
DSPCH	5	No load CPDR. Used for dispatches.
		CPD<0-3,> = 0's , PRES<0-3>, DS<24-31>
IPS	6	CPDR<0-31> := CPD<0-31>; CPD<0-25,28,29> = 0;
		CPD<26-27> = Length'; CPD31 = IUN, CPD30 = IURA_VALID
DSR	7	CPDR<0-31> := CPD<0-31> = DSR<0-31>
DISP	8	CPDR<0-31> := CPD<0-31> = DISP<0-31>
		only valid during IPOP.
SPAD	9	CPDR<0-31> := CPD<0-31> = Scratch pad<0-31>.
PC	A	CPDR<0-31> := CPD<0-31> = 0,CRE<0-2>,PC<4-31>
IOC	B	I/O Channel Data Register
		CPDR<0-31> := CPD<0-31> = 0, IOC<0-15>
LAR	C	ATU Logical Address Register
		CPDR<0-31> := CPD<0-31> = LAR<0-31>
PHY	D	CPDR<0-31> := CPD<0-31>. CPD0 = Validity, 1 = Hit
		2-4 = PROT<1-3>, 5-17 = TAG<2-14>, 18-31 = PHY<8-21>
ATUF	E	CPDR<0-31> := CPD<0-31> := ATU fault
		codes & SPAR. Clears faults
ATUO	F	CPDR<0-31> := CPD<0-31> := R&M bits of page addressed
		by LA & ATU state. See note under ATU randoms.
CIR	10	CPDR<0-31> := CPD<0-31> = CCP Command
		bits 0-17 = 0, 18-30 = CIR<18-30> 31 = con request
WIDE	11	CPDR<0-31> := CPD<0-31> = CCP Data
		bits 0-15 = 0, 16-31 = WIDE<16-31>
	12	Unassigned
RACS	13	CPDR<0-31> := CPD<0-31> = 0's, ACSR<0-3>
UCTR	14	Microsequencer Loop Counter. CPDR<0-31> := CPD<0-31>
		bits 0-7,16-19 undefined, 8-13 = FLAGS<0-5>'
		14 = 1, 15 = TOS<0>, 20-23 = TOS<4-7>,
		24-25 = FLAGS<6-7>', 26-27 = 0, 28-31 = CNTR<2-5>'
UTOS	15	Microsequencer Top of Stack. CPDR<0-31> := CPD<0-31>
		BITS 0-7,16-19 undefined, 8-13 = FLAGS<0-5>'
		14 = 1, 15 = TOS<0>', 20-31 = TOS<4-15>'
	16-1E	Unassigned

Note: The last u-instruction of each routine must code DISP in CPDS to load CPDR with the displacement for the next instruction.

What is claimed is:

1. A data processing system comprising central processor means including:

instruction processor means for decoding macro-instructions to produce a starting address of one or more micro-instructions;

micro-sequencing means responsive to said starting address for providing a sequence of one or more micro-instructions which include a plurality of microcontrol signals,

arithmetic logic means responsive to selected ones of said microcontrol signals for performing arithmetic or logical operations;

address translation means responsive to selected ones of said microcontrol signals for converting logical addresses into physical addresses;

memory means for storing information for use in said data processing system, said memory means including:

main memory means for storing said information;

temporary storage means for storing a selected portion of said information and having at least one set of input/output ports which includes one input-/output port for handling address information and another input/output port for handling data information;

controller means interconnected between said main memory means and said temporary storage means for controlling the transfer of information between said main memory means and said temporary storage means; and

first means for interconnecting said at least one set of input/output ports with said instruction processor means, said arithmetic logic means and said address translation means for transferring information therebetween.

2. A data processing system in accordance with claim 1 wherein said first interconnecting means includes:

a first address bus for transferring address information to said one input/output port; and

a first data bus for transferring non-address information.

3. A data processing system in accordance with claims 1 or 2 wherein said temporary storage means includes another set of input/output ports which includes one input/output port for handling address information and another input/output port for handling data information, and further wherein said system comprises:

an input/output channel means for communicating with one or more input/output devices external to said data processing system; and

second means for interconnecting said another set of input/output ports with said input/output channel means for transferring information therebetween.

4. A data processing system in accordance with claim 3 wherein said second interconnecting means comprises:

a second address bus for transferring address information to said one input/output port of said another set thereof; and

a second data bus for transferring non-address information to said another input/output port of said another set thereof.

5. A data processing system in accordance with claim 4 and further including:

a further data bus interconnecting said instruction processor means, said arithmetic logic means, said microsequencing means, said address translation unit and said input/output channel means for transferring non-address information among said interconnected means;

logical address bus means interconnecting said instruction processor means, said arithmetic logic means, said microsequencing means, said address translation unit and said input/output channel means for transferring logical address information among said interconnected means; and

physical address bus means interconnecting said instruction processor means, said arithmetic logic means, said microsequency means, said address translation unit and said input/output channel means for transferring physical address information among said interconnected means.

6. A data processing system in accordance with claim 5 and further including timing control means connected to said temporary storage means for controlling the transfer of information at said one and said another sets of input/output ports of said temporary storage means so that said first interconnecting means provides for transfer of information at said one set of ports during a first portion of an operating time cycle of said data processing system and for transfer of information at said another set of ports during a second portion of said operating time cycle.

* * * * *