# A SYSTEM FOR TESTING CUSTOM DESIGNED VLSI

by

J. Dinur

Memorandum No. UCB/ERL M85/57

15 July 1985

# A SYSTEM FOR TESTING CUSTOM DESIGNED VLSI

by

J. Dinur

Memorandum No. UCB/ERL M85/57

15 July 1985

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

# A System For Testing Custom Designed VLSI

*Julian Dinur*

Department of Electrical Engineering and Computer Science
Electrical Engineering Division
University of California
Berkeley
California 94720

*ABSTRACT*

This manual contains a general description of a system for testing custom designed VLSI chips. The manual also explains how to use the system and the main steps of testing a VLSI chip.
The system was built at U. C. Berkeley, E.E. Department .

May 24, 1985

This research was funded by the Defense Advanced Research Projects Agency, contract number: N00039-85-C-0107.

# Table of Contents

# A System For Testing Custom Designed VLSI

*Julian Dinur*

Department of Electrical Engineering and Computer Science
Electrical Engineering Division
University of California
Berkeley
California 94720

## 1. INTRODUCTION

An inexpensive and very efficient system for functionally testing custom designed VLSI chips has been designed and built. The system, currently used at U.C. Berkeley in the EECS Department, is very efficient for testing digital signal processor (DSP) and RAM chips.

The tester is connected to a host computer which down-loads a testing program and an input data file. During the program execution, the system stores the results in an output data file. After the execution of the program, the system up-loads the output file to the host computer for examination.

The system has two operation modes : a test mode and a debug mode.

In the test mode, the stored output results are the regular data outputs from the tested chip. In the debug mode, the stored output results are the contents of the output bus from the tested chip, after each input clock.

## 2. OPERATION MODES

### 2.1. The test mode

In this operation mode, the regular output results from the tested chip are stored in the tester memory. The results are then up-loaded to a file in the host computer for examination. The resultant file is compared with a pre-prepared file which contains the expected results.

In the case the two compared files are identical, the assumption is that the chip works properly. (To verify this assumption, the process is repeated for many different input files).

In the case the two compared files are different, further investigation is needed to find out what is wrong in the chip. This can be done by using the debug mode.

### 2.2. The debug mode

In this operation mode, the stored results in the tester memory are the contents of the output bus from the tested chip, after each input clock cycle. The results are up-loaded to a file in the host computer for examination. As in the test mode, the resultant file is compared with a pre-prepared file which contains the expected results. In this case an incorrect result can indicate which unit in the tested chip does not work properly, or which command is not executed as expected.

A farther investigation of the chip's design then needs to be performed, in order to pinpoint flaws and allow for any necessary design corrections.

## 3. HARDWARE

### 3.1. General description

The VLSI tester consists of 2 boards : a general purpose processor (GPP) board and a special purpose board. All the boards are built on Multibus compatible cards.

The GPP board is built around a 16-bit microprocessor and its main roles are:

a. Store the test program loaded from the host computer.

b. Run the test program in conjunction with the special purpose board and store the results.

c. Up-load the results to the host computer.

The general purpose board contains the interface between the GPP and the device under test (DUT). Two special purpose boards have been built : a board for the digital signal processor (DSP) type chips and a board for the RAM type chips.

The main roles of the board for testing DSP chips are :

a. Store the input data file sent by the GPP board and enable the DUT to read the input data at its own rate.

b. Store the output results from the DUT and enable the GPP to read this file at its own rate.

c. Enable data write/reads directly to/from the DUT.

The main roles of the board for testing RAM chips are :

a. Store the RAM address sent from the GPP.

b. Store the input data sent from the GPP and write it into the RAM.

c. Store the output data from the RAM and to enable the GPP to read it.

In order to enable the testing of chips with different pinout, a special adaptor has to be prepared and placed on the special purpose board.

### 3.2. The GPP board

The GPP is based on the Intel 80186 microprocessor and contains the following functional blocks (please see fig. 1 - fig.3 ) :

a. EPROM (2 k * 16 bits) - stores the firmware which enables the GPP to communicate with a host computer (VAX, for example) through the serial or parallel ports.

b. DRAM (64 k * 16 bits) - stores the test program, the input data file and the output testing results.

c. Two serial I/O ports - enable the serial communication between the GPP and a host computer through the RS-232 protocol.

d. Parallel interface - supports the communication with a host computer through the Multibus based protocol.

e. A/D converter - enables the GPP to read samples from the analog input at a rate up to 25 kwords (12 bits) per second.

f. D/A converter - enables the GPP to send data to the analog output at a rate up to 200 kwords (12 bits) per second.

The GPP board is based on the SPUDS board (for details on the SPUDS board, please see the report "SPUDS" by William Baringer, Memorandum No. UCB/ERL M84/4, January, 1984).

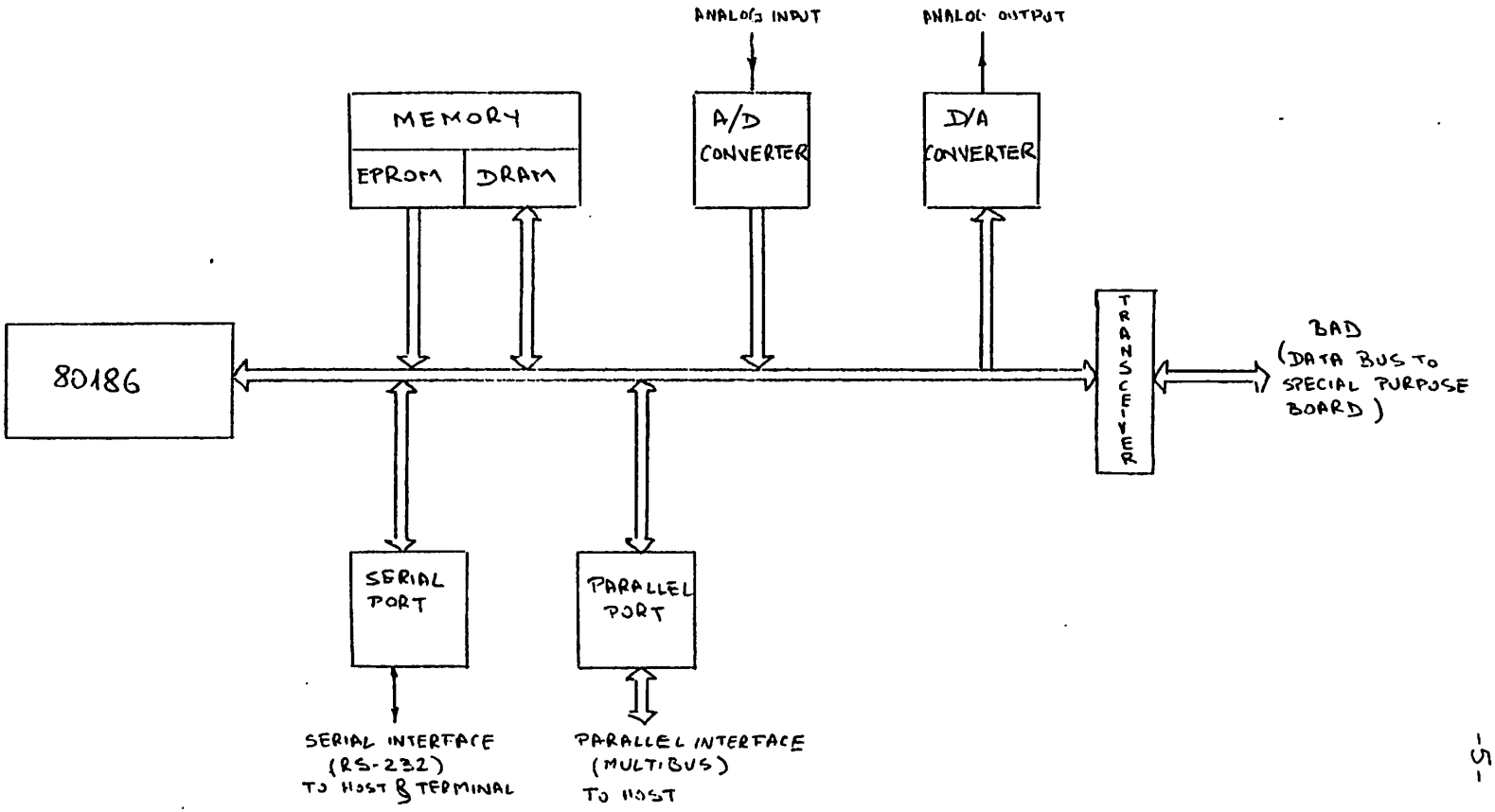



FIGURE 1 : GPP BOARD - GENERAL BLOCK DIAGRAM

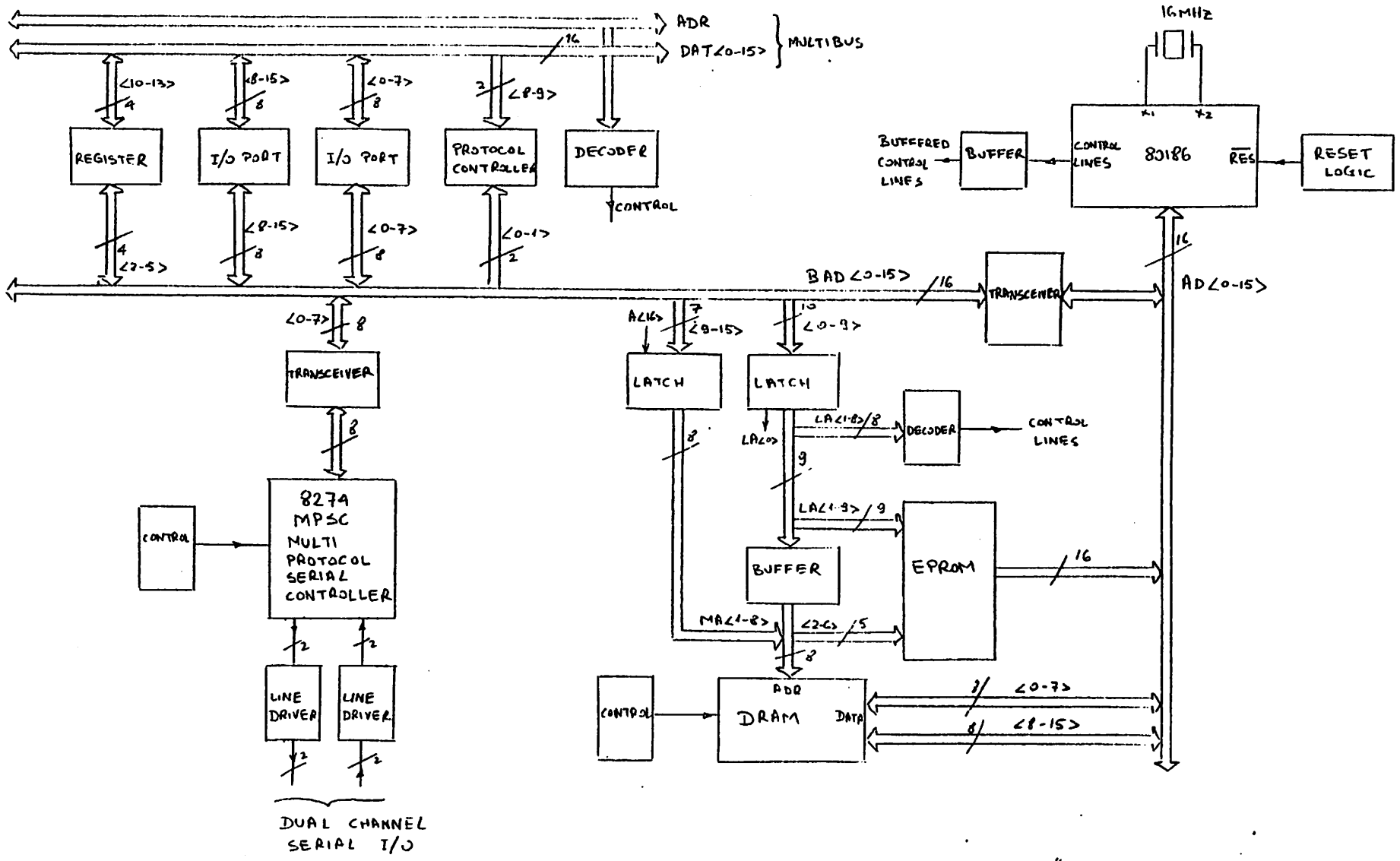



FIGURE 2: GPP BOARD - DIGITAL SECTION ("SPUDS") DETAILED BLOCK DIAGRAM

JULIAN DINUR
8/21/1984

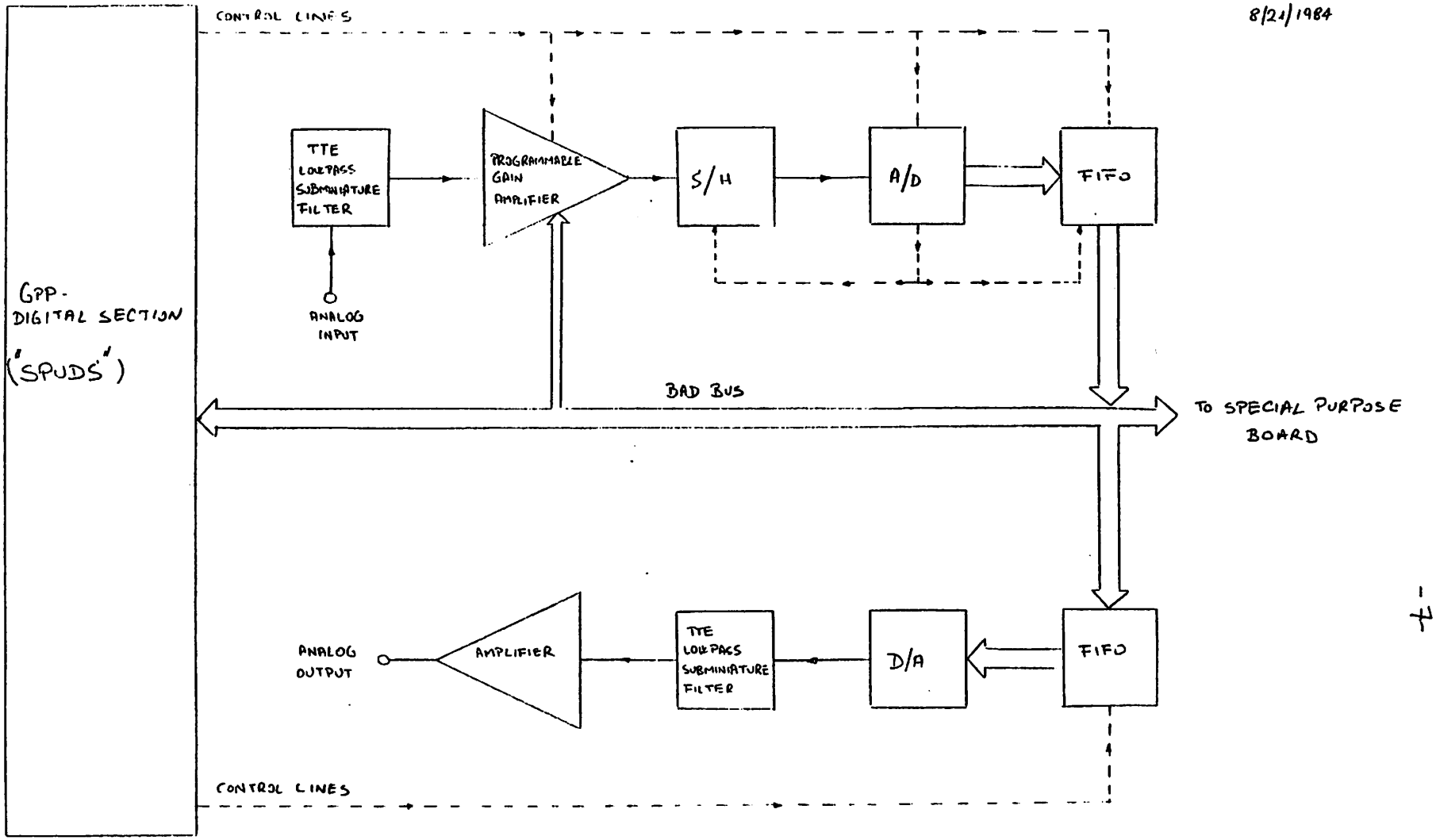



FIGURE 3: GPP BOARD - ANALOG SECTION
DETAILED BLOCK DIAGRAM

### 3.3. The DSP chips tester board

This board consists of the following functional blocks ( please see fig. 4 - fig.5 ) :

a. FIFO input memory - stores up to 16 words of 16 bits each of input data sent from the GPP to the DUT.

b. FIFO output memory - stores up to 16 words of 16 bits each of output data sent from the DUT to the GPP.

c. Clock generator - generates 2-phase non overlapping clocks (1/4 duty cycle).

Due to the small number of words in the FIFO, the minimum time between consecutive input data to the DUT is about 5 useconds. (A new faster board with a larger memory is under development).

### 3.4. The RAM chips tester board

This board contains the following functional blocks (please see fig. 2 ) :

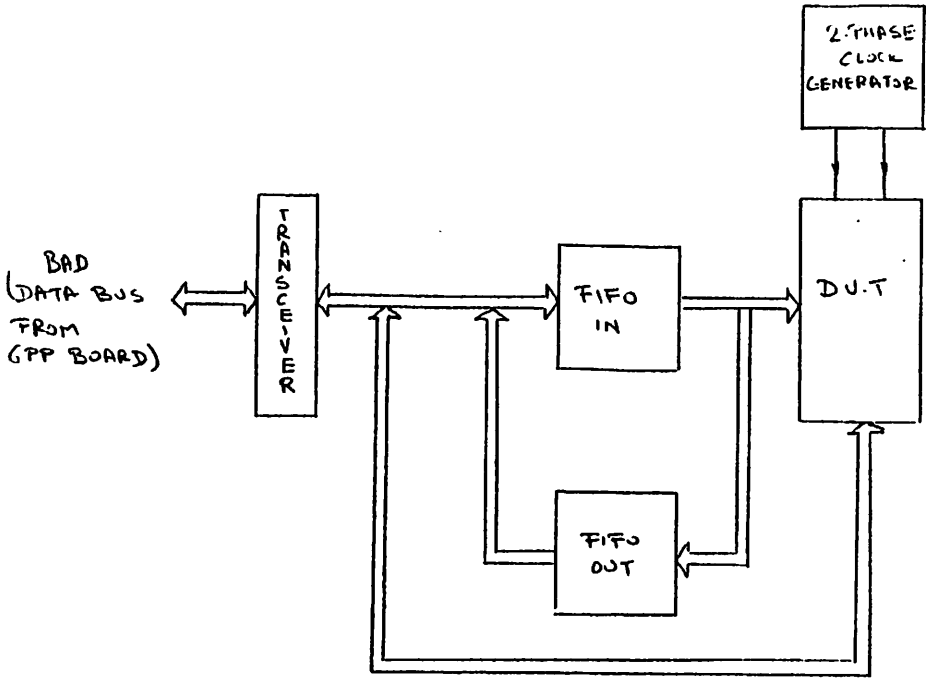
a. Address register - stores up to 8 bits of address sent from the GPP.

b. Input data register - stores up to 16 bits of input data sent from the GPP to the DUT.

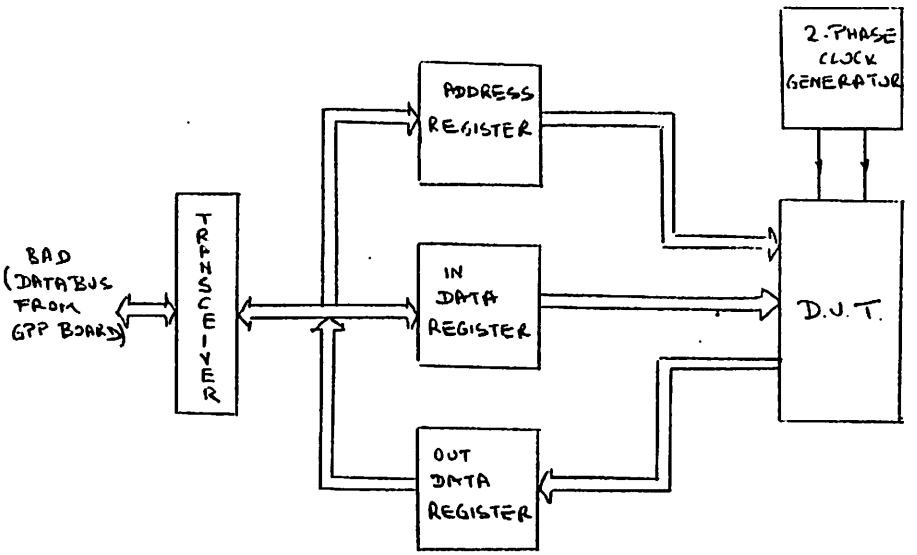
c. Output data register - stores up to 24 bits of output data sent from the DUT to the GPP.

d. Clock generator - generates 2-phase non overlapping clocks (3/8 duty cycle).

Because the GPP board sees this board as a mapped I/O device, the RAM addresses have to be sent first and then stored on the tester board. After that (about 1 usecond later), the data can be written or read from the RAM. ( A new faster board is under development).




A.




B.

FIGURE 4 : SPECIAL PURPOSE BOARD - GENERAL BLOCK DIAGRAM :

A. : FOR DSP CHIPS

B. : FOR RAM CHIPS

JULIAN DINUR
8/21/1984

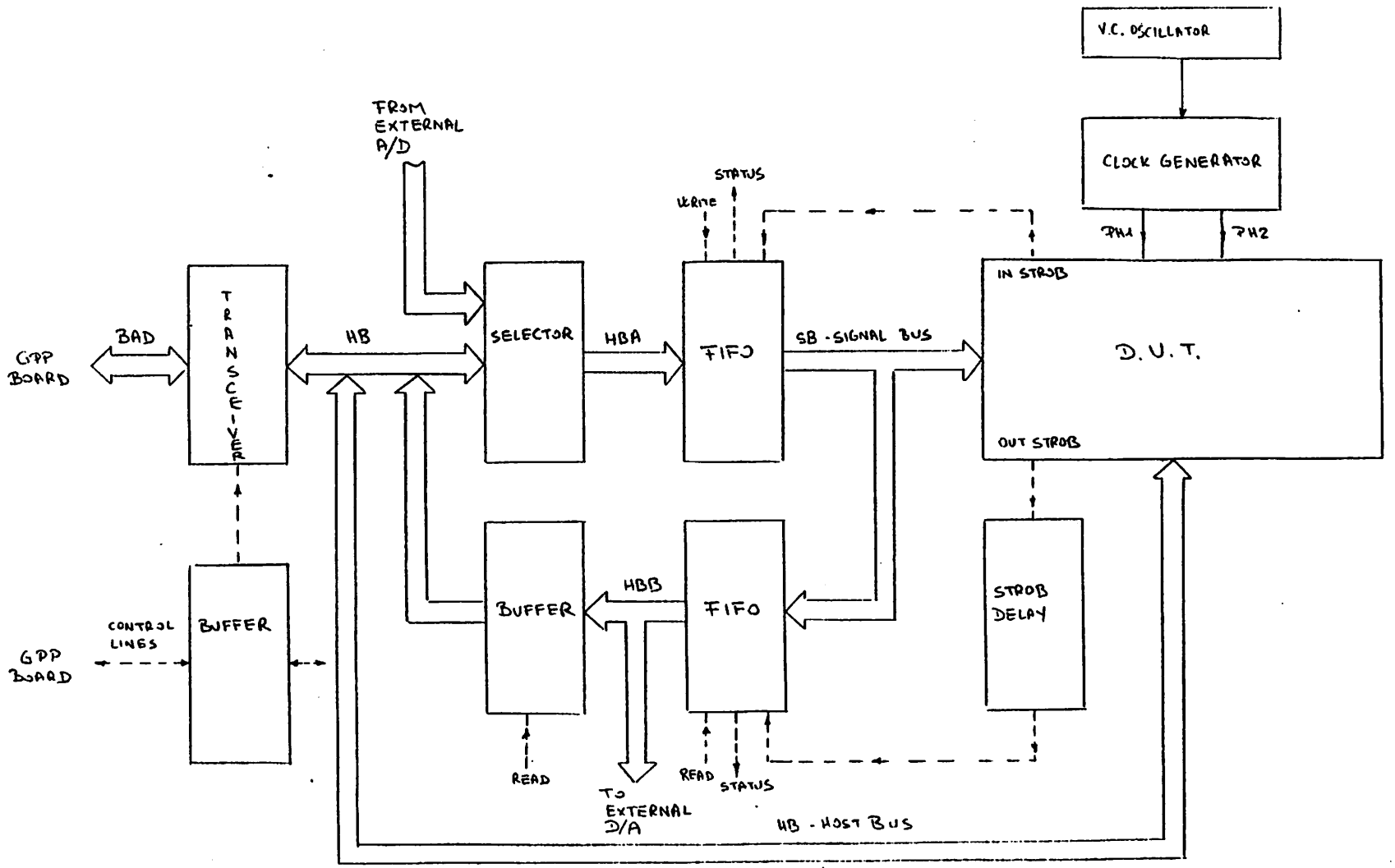



FIGURE 5 : SPECIAL PURPOSE BOARD FOR DSP CHIPS - DETAILED BLOCK DIAGRAM

## 4. FIRMWARE

The firmware in the EPROM on the GPP board was written so that a terminal may be connected to serial channel B and the host computer to serial channel A. The GPP board then acts in a "terminal emulator" mode where characters received from the host are passed on to the terminal's screen. However, if a certain string of control characters is received from the host computer, the code is not sent to the terminal, but stored in the DRAM on the GPP board. At the end of the transfer of the user's program, another set of control characters is sent as an end-of-text indicator, and program execution of the 80186 commences at the beginning of the new program in DRAM.

A typical program loaded into DRAM could allow control of the tester via the terminal, and send data back to the terminal for examination. Data may also be sent to the host computer for storing and further processing.

## 5. SOFTWARE

The test programs can be written in C and/or Intel 80186 assembly language.

The C programmer can use a large amount of pre-defined procedures (please, see appendix A) and an 8086 cross compiler .

### 5.1. The most useful commands

A program in assembly language should have the suffix .a86 . For example:

prog.a86

The assembler a86 will produce the file prog.b :

a86 prog.a86

The C cross assembler will produce the executable file xx.com ( xx is an arbitrarily name ) :

cc86 -1 -o xx prog.b

To assemble a C program, p.c, and an assembly program, prog.b, the following command should be used :

cc86 -1 -o xx p.c prog.b

To load the executable file in tester's memory, one should use the command:

pdploader xx.com /usr/local/86ldr

If using the C program presented in appendix c, the output results will be stored in the file "results" in the host computer, in the user's directory. To read this file on the terminal, the following command can be used :

dumpdata < results

dumpdata.c is a C dump program (please see appendix G). The user can modify this program in order to get a different output format.

## 6. THE MAIN STEPS OF TESTING A CHIP

a. Prepare the adaptor for the specific chip pinout (a special form has been prepared for this purpose, please see figure 6).

b. Prepare the software (the assembly language program and/or the C program with the input data file).

c. Begin to test in the "test mode" : load the testing program and examine the output result file.

d. In the case the the results are incorrect, the chip can be tested in the "debug mode". To do that a small hardware modification is needed and the software has to be updated.

e. The use of an oscilloscope and a logic analyzer for testing the waveforms and the timing of the chip's signals is strongly recommended.

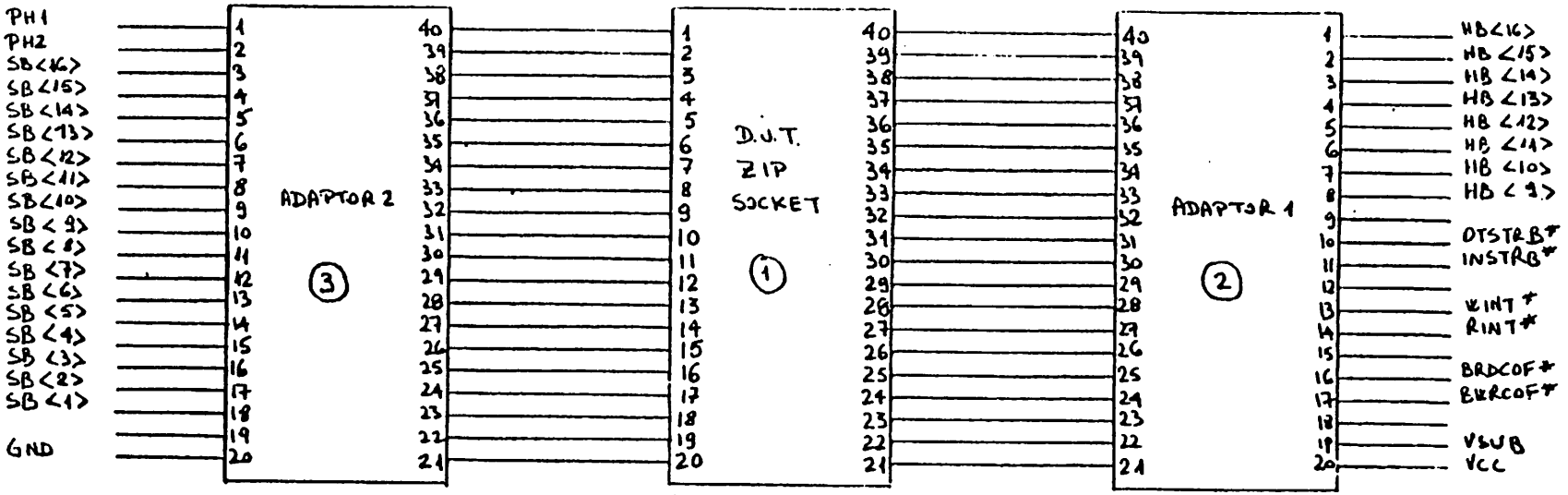



FIGURE 6: DSP CHIPS TESTER BOARD: ZIP SOCKET AND ADAPTORS

## 7. RECOMMENDED DESIGN RULES FOR TESTABILITY

a. Use only "active low" external signals.

b. Use only uninverted input clock.

c. Use built-in firmware for testing.

d. Use a reset pin to reset the chip. All internal flip flops and latches should be reset or set to known states :

- Counters
- Stack pointers
- Internal data busses
- Tristate output pads.

e. Being able to preset portions of a circuit can be very useful. The preset value can be hardwired into the circuit or programmed from the outside. The intent of presettability is to allow a portion of the circuit to be easily placed into a known state other than the reset state to activate a function to be tested.

## 8. APPENDIX A - Procedures

The procedures were prepared by Robert Kavaler. They are on ucboz in : ~kavaler/88/cc86mit/lib186.

**sys/**

```
    DmaSetup.a86     DmaSetup((long) source, (long) dest, count, control,
                       channel (FFC0/FFD0))
MemFill.a86  MemFill(Physical_Address, count, data0, data1, ...)
Refresh.c    Refresh()
int.a86      splow()        /* enable interrupts */
             splhigh() /* disable interrupts */
             splx(splhigh_return) /* return the old set-up for interrupts */
             SetInt(type, raddr)
io.a86       IoIn(port)     /* write data to port */
             IoOut(port, data) /* read data from port */
ios.a86      IoInsb(port, address, len)
             IoInsw(port, address, len)
             IoOutsb(port, address, len)
             IoOutsw(port, address, len)
lblt.c       lblt(to, from, len) /* long block transfer by DMA */
mem.a86      MemIn(Physical_Address) /* read words */
             MemOut(Physical_Address, data) /* write words */
misc.a86 GetDS()
             GetCS()
physaddr.a86 physaddr(addr)
sbrk.c       sbrk(incr)
sys.c        —
ttyio.c      /* serial communication with the TTY */
             _rcn_out(c) /* data : c - character */
             _rcn_ob()  /* status */
             _rcn_in()  /* status */
             _rcn_ib()  /* data */
vaxio.c      /* serial communication with the host computer */
             _rvx_out(c)
             _rvx_ob()
             _rvx_in()
             _rvx_ib()
```

**opsys/**

```
dispatch.a86 Dispatch()
             PromImm - from assembly language routines only, use jmp
menu.c       menu()
object.c     CreatObjects(length, number)
             GetObject(Objects)
             LinkObject(Objects, object)
             UnlinkObject(Objects, object)
plock.c      Lock(x)
             Unlock(x)
pmalloc.c    pmalloc(num)
             pfree(p)
process.c    ProcStart(Processes)
             ProcKill(Processes)
procinit.a86 InitDone()
```

```
ptime.c         TimeStamp()
queue.c         creatq(msize, nmessages)
          resetq(q)
          getq(q)
          igetq(q)
          getlq(q, b)
          putq(q)
          iputq(q)
          putlq(q, b)
          killq()
shell.c         shell(prompt, commands)
```

**messages/ /* for Multibus communication */**

```
MesgInit.c      MesgInit()
MesgOut.c       MesgOut(data, length)
MesgQIn.c       MesgIn(p)
          MesgFlush()
          NewMgInQ(type, q)
MesgQOut.c      MesgQOut(p)
          SendObject(Objects, obj, len)
          SendQueue(q, len)
```

```
multibus/       -- low level stuff
gen/            -- standard C library
stdio/          -- *printf, and getline (my own creation, used by shell)
```

Standard I/O is difficult to explain, just follow these rules. Before any input/output use one of the following calls:

```
#include <stdio.h>
     extern SIOSYSTEM sio_oconsole;
     sio_new(&sio_oconsole)          if using RS232 ports

                    -or-

#include <stdio.h>
     extern SIOSYSTEM sio_multibus;
     sio_new(&sio_multibus)          if using SUN "mbhost" program
```

## 9. APPENDIX B - An example of a program in assembly language

```
        .data                   |a program to test the cmos ram (cram)
        .comm   _a,18           |a contains 18 bytes
        .comm   _b,12           |b contains 12 bytes
        .comm   _c,18
        .comm   _addra,2
        .comm   _addrb,2
        .text                   |
        .globl  _first          |_first is the name of the program
        .globl  _addra          |address of input data
        .globl  _addrb          |address of output data
_first:
|
        pusha                   |push all registers on stack
|
MAXDAT=3                        |the max no. of blocks (3 words each)
                        |in input file
|
        mov bx,*1               |
        mov _addra,#_a
        mov _addrb,#_b
|
|
|
init1:      mov cx,#MAXDAT
        mov si,#_a
        mov di,#_b
|
|
        mov dx,#0x028E          |debug pulse
        outw                    |
|
|
l1:     mov ax,(si)             |read data from (80186)memory
        mov     dx,#0x028C      |write data to addr. reg.
        outw                    |
|
        add  si,*2              |prepare for next addr.
|
        mov ax,(si)             |read data from (80186)memory
        mov dx,#0x0284          |write data to data reg.
        outw                    |
|
|
        mov dx,#0x0282          |write to cram
        outw                    |
|
        add  si,*2              |
|
        loop    l1              |repeat for each cram addr.
|
|
```

```
        mov cx,#MAXDAT  |
|
12:     mov ax,(si)             |read data from memory
        mov dx,#0x028C          |write data to addr. reg
        outw                    |
|
        add si,*2       |
|
        mov dx,#0x028A  |
        inw             |read data from cram
|
        mov dx,#0x0286  |read M.S.Part of data from cram
        inw             |4 m.s.b.
|
        and ax,#0x000F  |clear non-relevant bits
|
        mov (di),ax             |store data in memory
|
        add di,*2       |
|
        mov dx,#0x0288  |read L.S.Part of data from cram
        inw             |16 l.s.b.
|
        mov (di),ax             |store data in memory
|
        add di,*2       |
|
|
        loop 12                 |repeat for each addr. in cram
|
|
        jmp init1       |start from the beginning
        dec bx
        jnz  init1
        popa                    |restore all registers
        ret
        .even
        .data
_a:
        .word   7
        .word   0
        .word   10
        .word   256
        .word   63
        .word   32767
        .word   7
        .word   10
        .word   63
_c:
        .word   0,0,0,0,0,0,0,0,0
_b:
        .word   0,0,0,0,0,0
```

## 10. APPENDIX C - An example of a program in C

This C program should be assembled with a program in assembly language in orde to enable the up-loading of the output file from the tester memory.

```
#define  SIO  sio_oconsole
#include <stdio.h>
#include <sys.h>
#define  LENGTH  48

extern   first();
extern int *addra;
extern int *addrb;

main()
{
     splhigh();                    /* disable interrupt */
     first();                  /* call the assembly program */
     splow();                  /* enable interrupt */
     initialize();                 /* init. the tty */
     printf("start up-loading\n ");
     r_write("results",addra,LENGTH);     /* up-load the output file */
     /* "results" : the file name in te host */
     /*  addra    : the starting address in the DRAM on GPP board */
     /*  LENGTH   : the number of bytes in the file */
     printf("end up-loading\n ");
}

initialize()
{
     extern SIOSYSTEM SIO;

     sio_new(&SIO);
     vaxrawmode();
}

vaxrawmode()          /* the following code initializes the serial */
               /* controller on the tester board for writing */
               /* to the host computer                       */
{
     IoOut(VAXCSR, 0x05);
     IoOut(VAXCSR, 0x68);
     IoOut(VAXCSR, 0x03);
     IoOut(VAXCSR, 0xC1);
     IoOut(VAXCSR, 0x04);
     IoOut(VAXCSR, 0x4E);
}
```

## 11. APPENDIX D - The I/O addresses for the special purpose board

```
          .globl _main        |adrtest1.a86
_main:
loop:  mov    dx,#0x0280
      in              |read status FIFO 1 (A/D)
       out            |set amp. gain
       add dx,*2
      in              |read status FIFO 2 (D/A)
      out             |write data to FIFO 3 (host to D.U.T.)
      add    dx,*2
      in              |read status FIFO 3 (host to D.U.T.)
      out             |write coef. to D.U.T.
       add    dx,*2
      in              |read status FIFO 4 (D.U.T. to host)
      out             |clear FIFO 1 (A/D)
       add    dx,*2
      in              |read coef. from D.U.T.
      out             |clear FIFO 2 (D/A)
      add    dx,*2
      in              |read data from FIFO 4 (D.U.T. to host)
      out             |write data to FIFO 2 (D/A)
      add    dx,*2
      in              |N.D
      out             |N.D
      add    dx,*2
      in              |read data from FIFO 1 (A/D)
      out             |N.D
      nop;nop;nop
     jmp loop
```

## 12. APPENDIX E - I/O Drivers

UNIX 4.2BSD I/O driver for GPP board

Hardware:

There are three distinct ports on the GPP board: CSR, DATA, and RESET. A read from the RESET port will reset the GPP board and jump to the on-board PROM. This is considered a hard-reset. The CSR and DATA ports are both readable and writable, but they have a different meaning if read or written. Thus code like:

```
CSR |= ENABLE;
```

will not do the obvious thing. Instead a variable is kept in memory that contains what is in the CSR and was last written. The data port is designed so that strings of characters will not have their bytes reversed, while strings of shorts will end up with reversed bytes. This is because of the byte ordering incompatibility between the SUN and 186. In addition, the CSR port is active low on the SUN side, so every read and write to the SUN CSR should be complemented to get active high signals. All of these concerns are handled correctly by the driver.

UNIX driver:

The driver implements 4 system calls: open, read, write, and ioctl. Open just checks that its arguments are legal. The read and write system calls receive and send "messages" to the GPP board through the MULTIBUS. Message lengths must be even (in bytes). A message consists of any number of data words followed by a unique word (called a header). The header word is distinguished from data through the use of the CSR port. From the UNIX program point of view a message is just a variable length data stream that is sent to the GPP board. The driver handles all handshaking and header generation. The CSR bits are defined as follows:

```
/* description of CSR bits */
#define  SP_OUTEN     0x000100
#define  SP_OUTRDY    0x000100
#define  SP_INEN      0x000200
#define  SP_INRDY     0x000200
#define  SP_PROGMASK      0x003C00

/* message bits */
#define  SP_MGMASKTYPE  0x000C00
#define  SP_MGTYPEH 0x000400
#define  SP_MGTYPEM 0x000800
```

TYPEM is the CSR bits for messages, TYPEH is the CSR bits for headers. All headers also have a data word associated with them. Thus there are 65536 distinct headers, only 2 are currently

used, 0 and 1. Most headers are 0, but the down-load PROM on the GPP board recognizes a header of 1 to mean start the loading the incoming message into RAM, and execute from there. To change the header that is sent out at the end of a message one uses the SP_CHHEADER ioctl call:

```
int i;
i = (header);
ioctl(sp_fn, SP_CHHEADER, &i);
```

Additional ioctl calls are:

SP_RESET - read the RESET port, returning control to the spuds PROM.

SP_FLUSH - called to reset the driver if read or write terminates early (i.e. from a kill signal).

SP_{RD,WR}{DATA,CSR} - read/write the DATA/CSR registers directly. These calls should never be used except to debug things. CSR bits come out active high.

The read system call should be used as:

```
actual_message_size = read(sp_fn, buf, MAX_MESSAGE_SIZE);
```

only one process should execute this command since one never knows what messages will be received.

The write system call can be used by any process. Serialization of the calls is perform by the driver. The write system call is:

```
errcode = write(sp_fn, outmessage, outmessage_length);
```

the value returned should always be outmessage_length or an error occured.

## 13. APPENDIX F - The communication protocol over the serial channels

```
    #include <stdio.h>
#include <host.h>

/*              To READ or WRITE UNIX files over serial lines

    The following protocol is used on the client end:

  in read() or write()
    - send a SYNC FILE_RD (or FILE_WR)
    - send the file name
    - send a SYNC END_OF_MESSAGE

  in r_read() or r_write()
    - wait for the client to respond with SYNC FILE_RD (or FILE_WR)
        (if we receive SYNC ERROR, there is a problem reading
         or writing the file)
    IN READ
        - read in characters from the host until a SYNC END_OF_MESSAGE
          is received
    IN WRITE
        - send BLOCK_SZ characters to the host
        - send a SYNC EOB
        - repeat until all the desired samples have been transmitted
        - send a SYNC END_OF_MESSAGE
                                                    */

/*  SHELL ESCAPE

    works similarly to the read/write functions.
    1) client sends SYNC COMMAND_MODE
    2) client sends command
    3) client sends END_OF_MESSAGE
    4) client goes into "terminal mode" which it stays in until
        it gets a SYNC END_OF_MESSAGE from the pdp
    5) host executes the command taking stdin from the client
        which it now treats as a standard terminal
        (it has temporarily discontinued raw mode)
    6) host transmits SYNC END_OF_MESSAGE when the forked of
        command has finished
    7) client sends END_OF_MESSAGE to acknowledge
*/
```

## 14. APPENDIX G - The dumpdata.c program

This program is used to dump the output data results stored in a ASCII file, by the testing program, to a terminal.

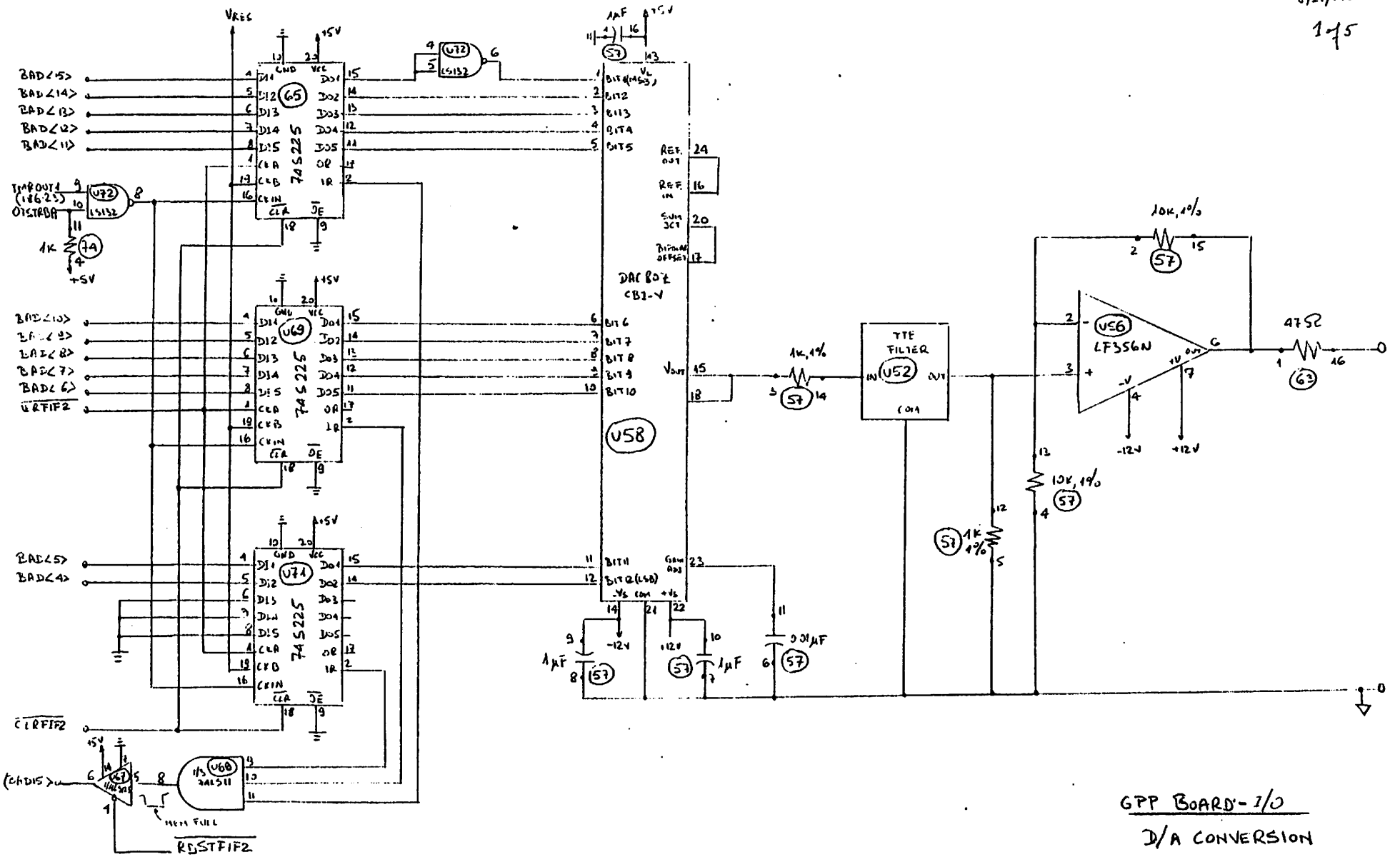
```
#include <stdio.h>

main(argc, argv)
     char *argv[];
     int  argc;
{
     printblock(10, 8);  /* dump the input data file */
     while(1) {
          printblock(36, 8); /* dump the output results */
     }
}

getshort(fp)
     FILE *fp;
{
     int c1, c2;

     c1 = getchar(fp);
     c2 = getchar(fp);
     if(c2 == EOF) {
          printf("0);
          exit(0);
     }
     return (int) (short) ((c1&0xFF)+((c2&0xFF)<<8));
}

printblock(blocksize, linesize)
     int  blocksize, linesize;
{
     int  i, col;

     col=0;
     for(i=0; i<blocksize; i++) {
          if(col++ >= linesize) {
               col=1;
               printf("0);
          }
          printf("%8d", getshort(stdin));
     }
     printf("0);
}
```

# 15. APPENDIX H - SCHEMATICS

## 15.1. GPP Board - I/O Part

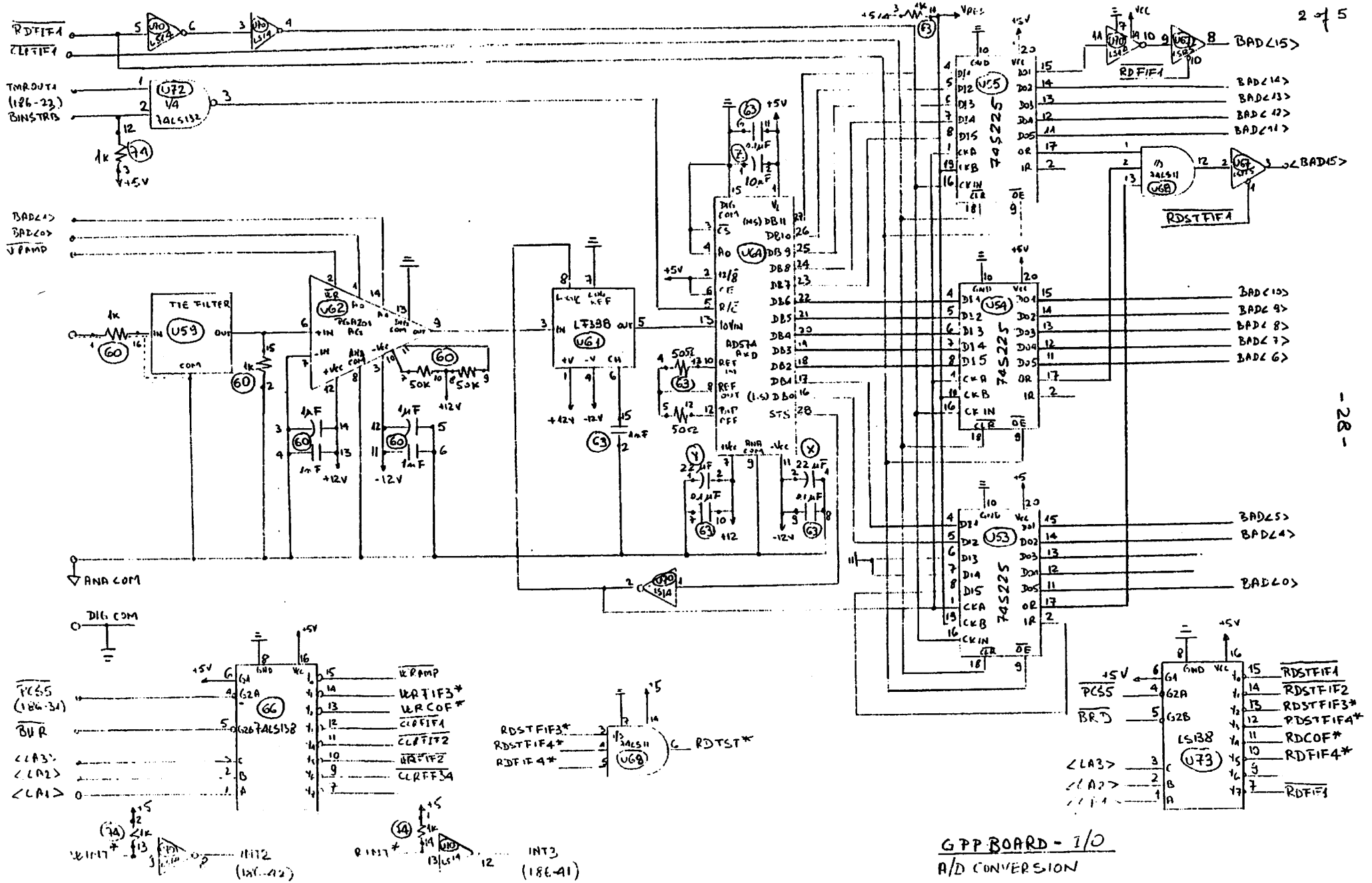
JULIAN DINUR
8/21/1984





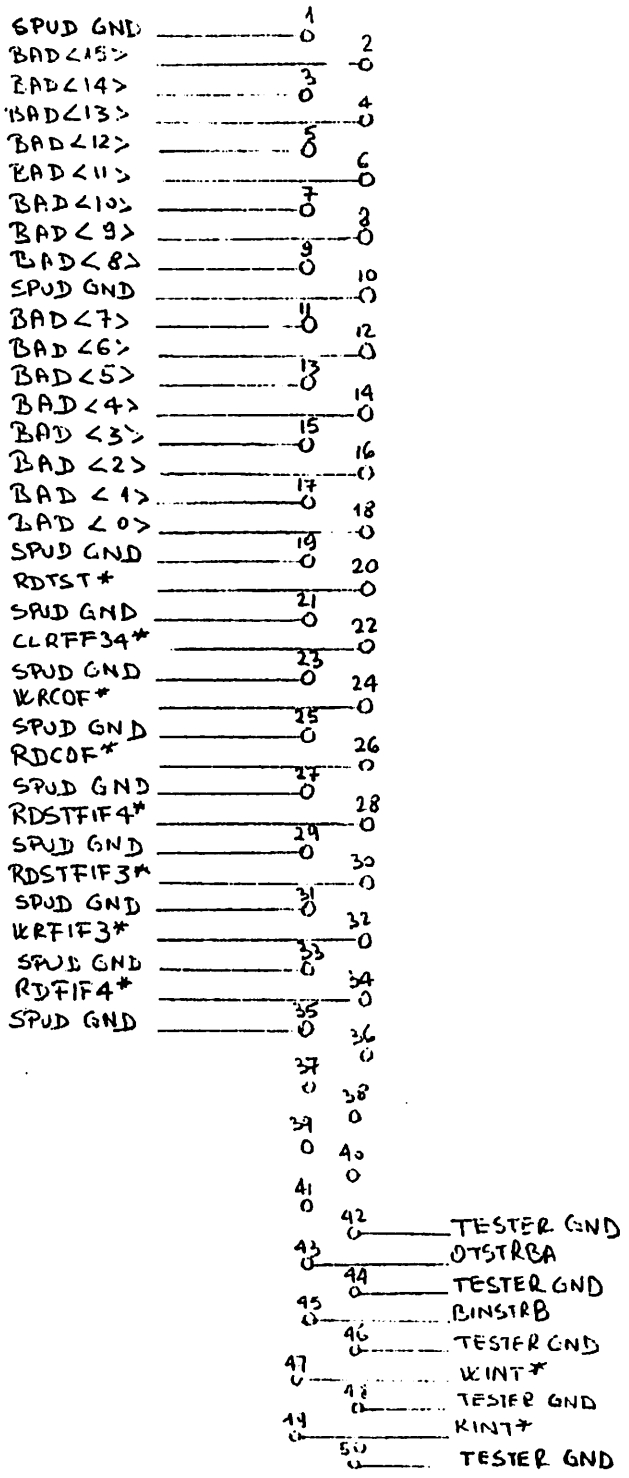

GPP BOARD - I/O

D/A CONVERSION



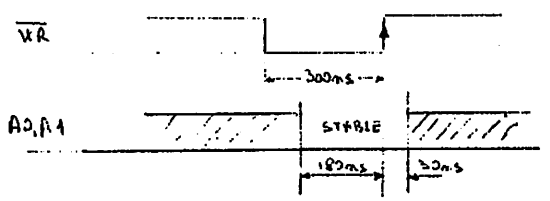

GPP BOARD - I/O

A/D CONVERSION

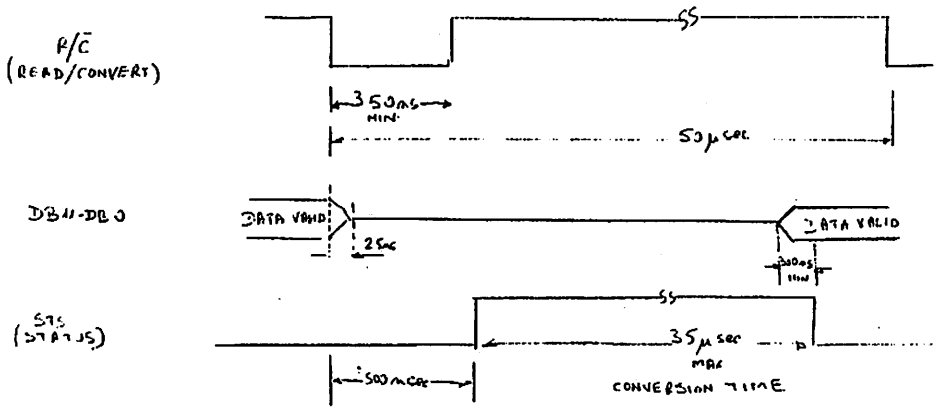



SPUD'S VLSI TESTER
PIN DESIGNATION FOR 50-PIN
RIBBON CABLE CONNECTOR (A)

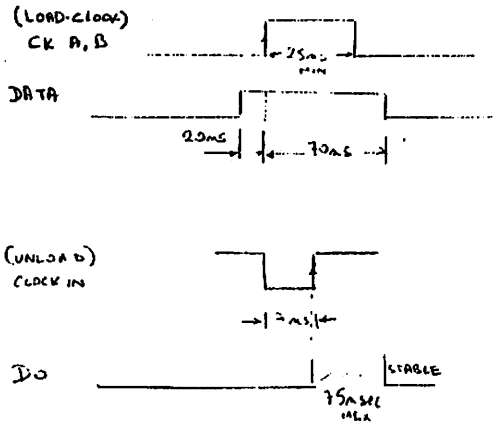
PGA 2041AG




AD574 AKD




SN74S225

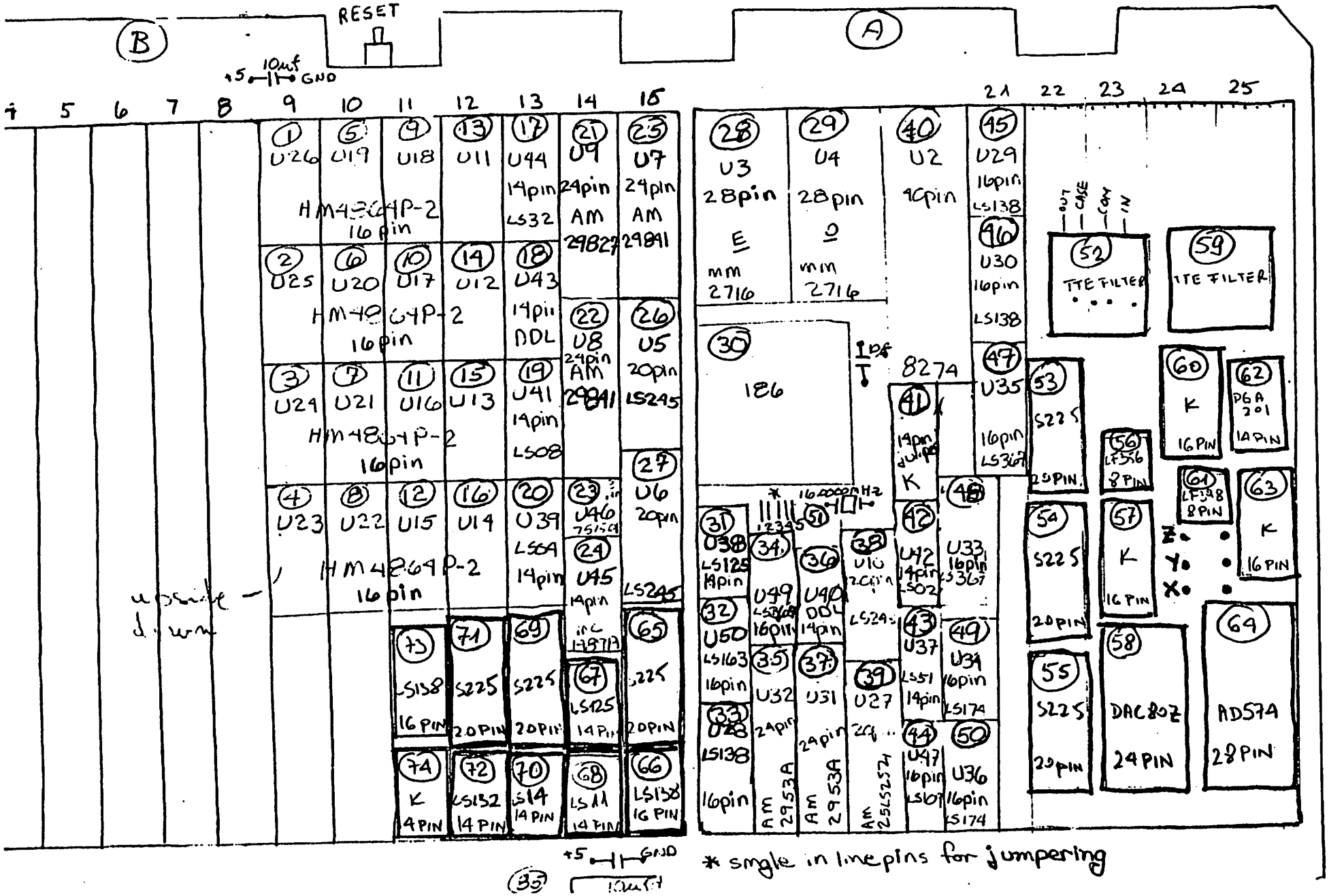



GPP BOARD I/O

TIMING DIAGRAMS

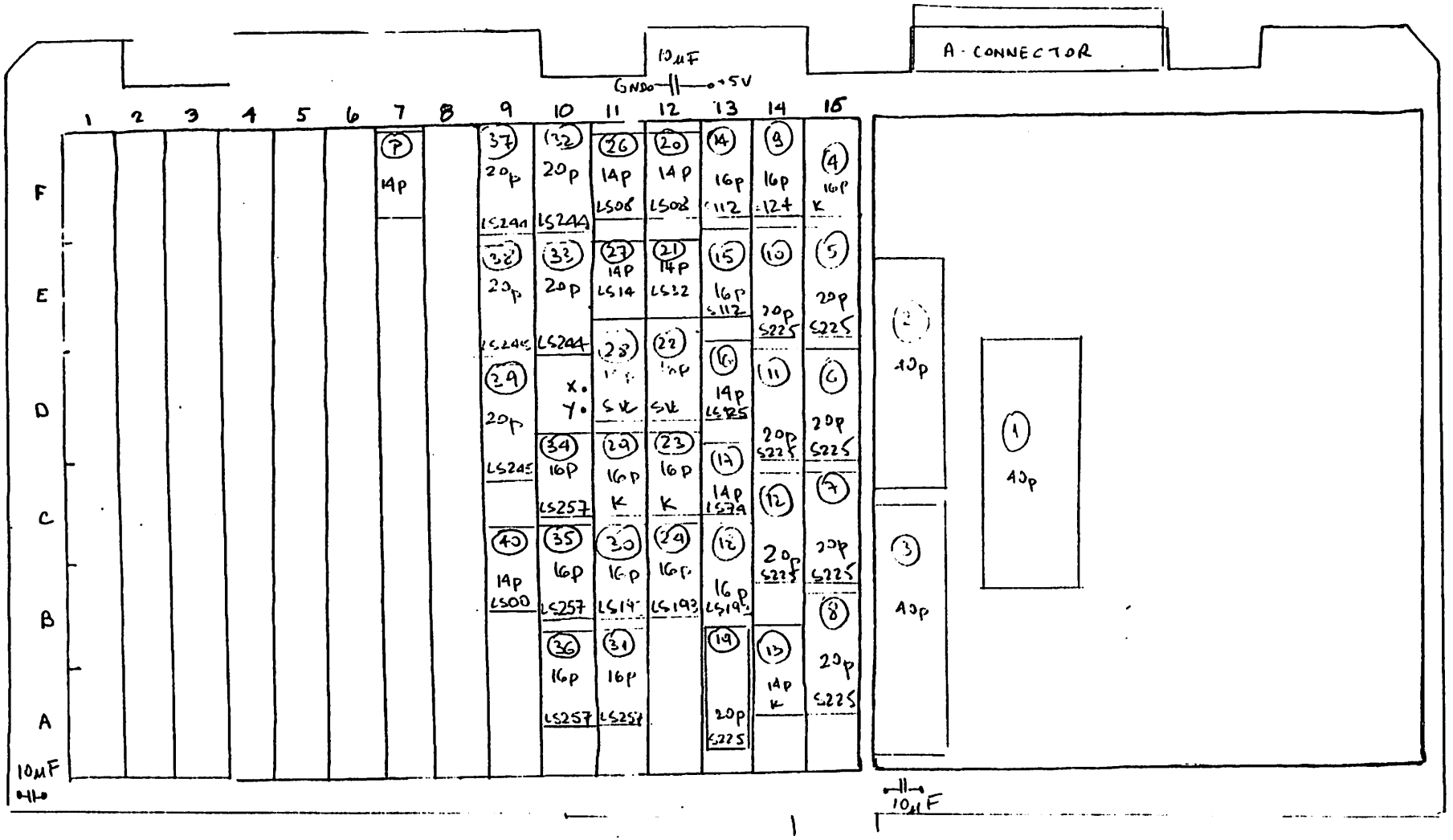
# GPP BOARD - LAYOUT FOR WIRE WRAPPING

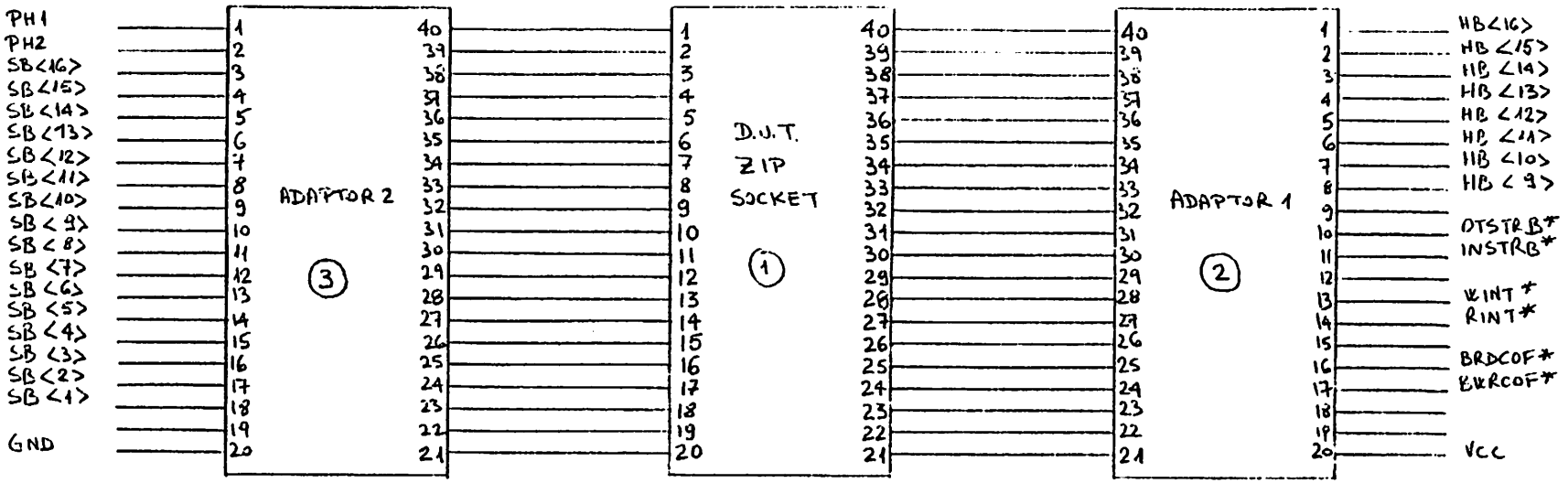
Front View

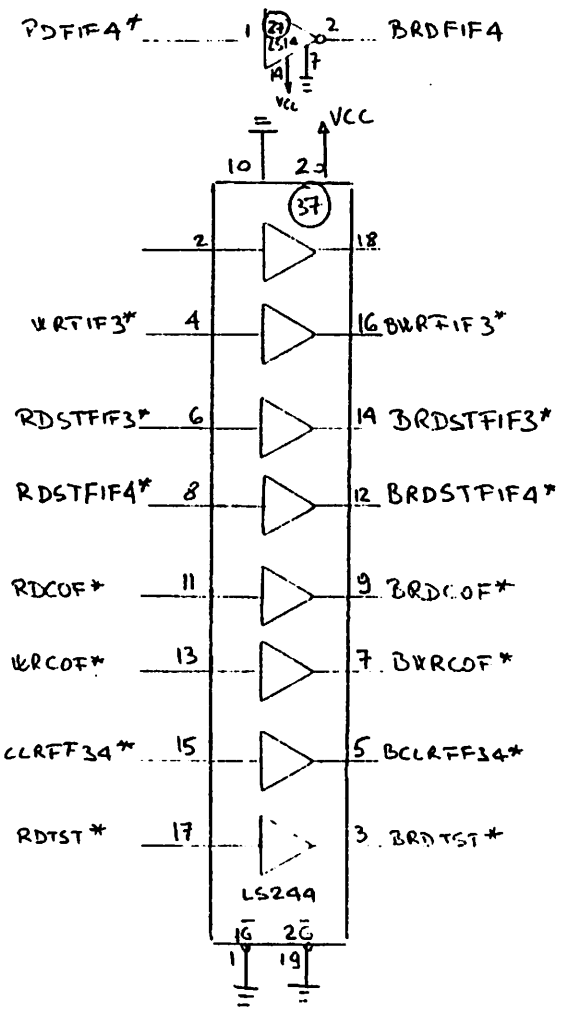


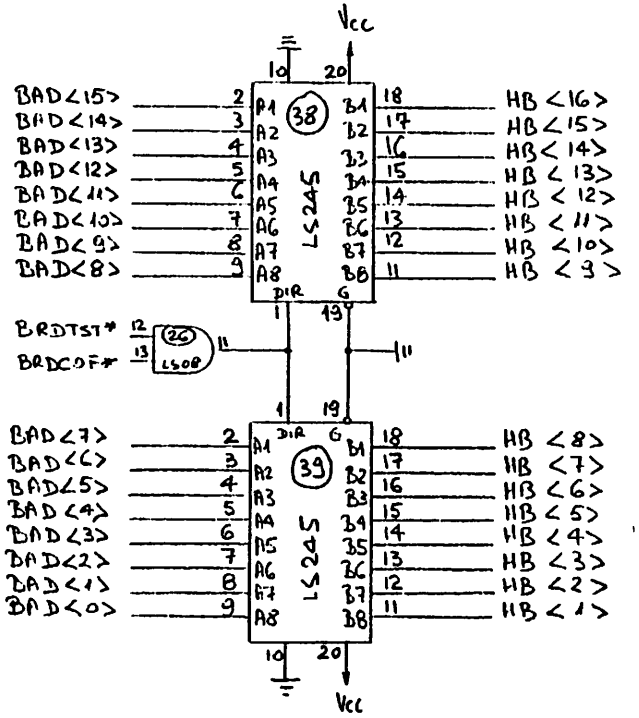
## 15.2. DSP Chips Tester Board

# VLSI TESTER LAYOUT (FOR DSP CHIPS)
FRONT VIEW



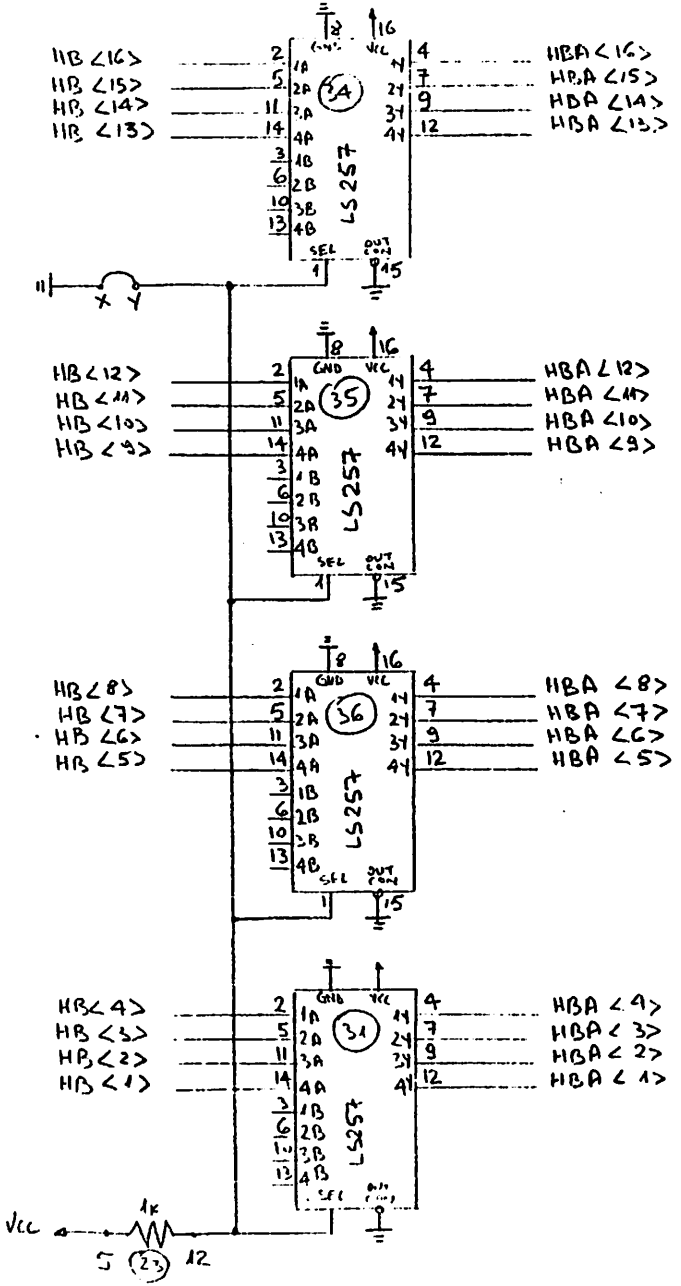

PH1
PH2
SB<16>
SB<15>
SB<14>
SB<13>
SB<12>
SB<11>
SB<10>
SB<9>
SB<8>
SB<7>
SB<6>
SB<5>
SB<4>
SB<3>
SB<2>
SB<1>
GND
ADAPTOR 2
3
D.U.T.
ZIP
SOCKET
1
ADAPTOR 1
2
HB<16>
HB<15>
HB<14>
HB<13>
HB<12>
HB<11>
HB<10>
HB<9>
OTSTRB*
INSTRB*
WINT*
RINT*
BRDCOF*
BWRCOF*
VCC

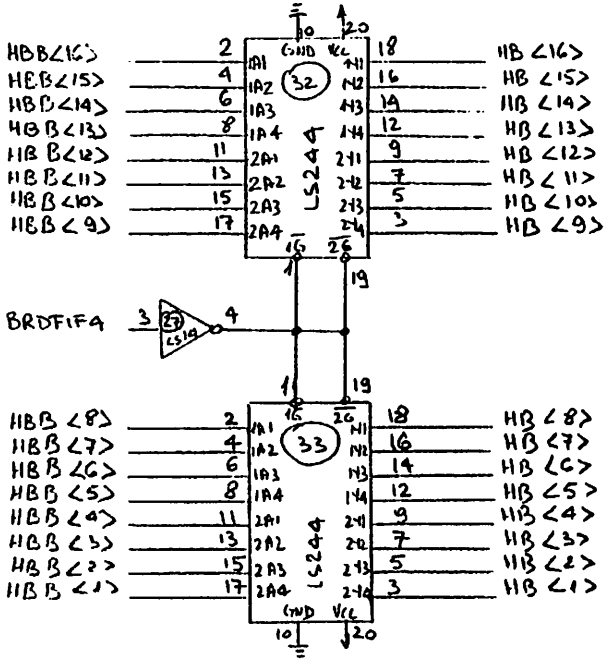



VLSI TESTER

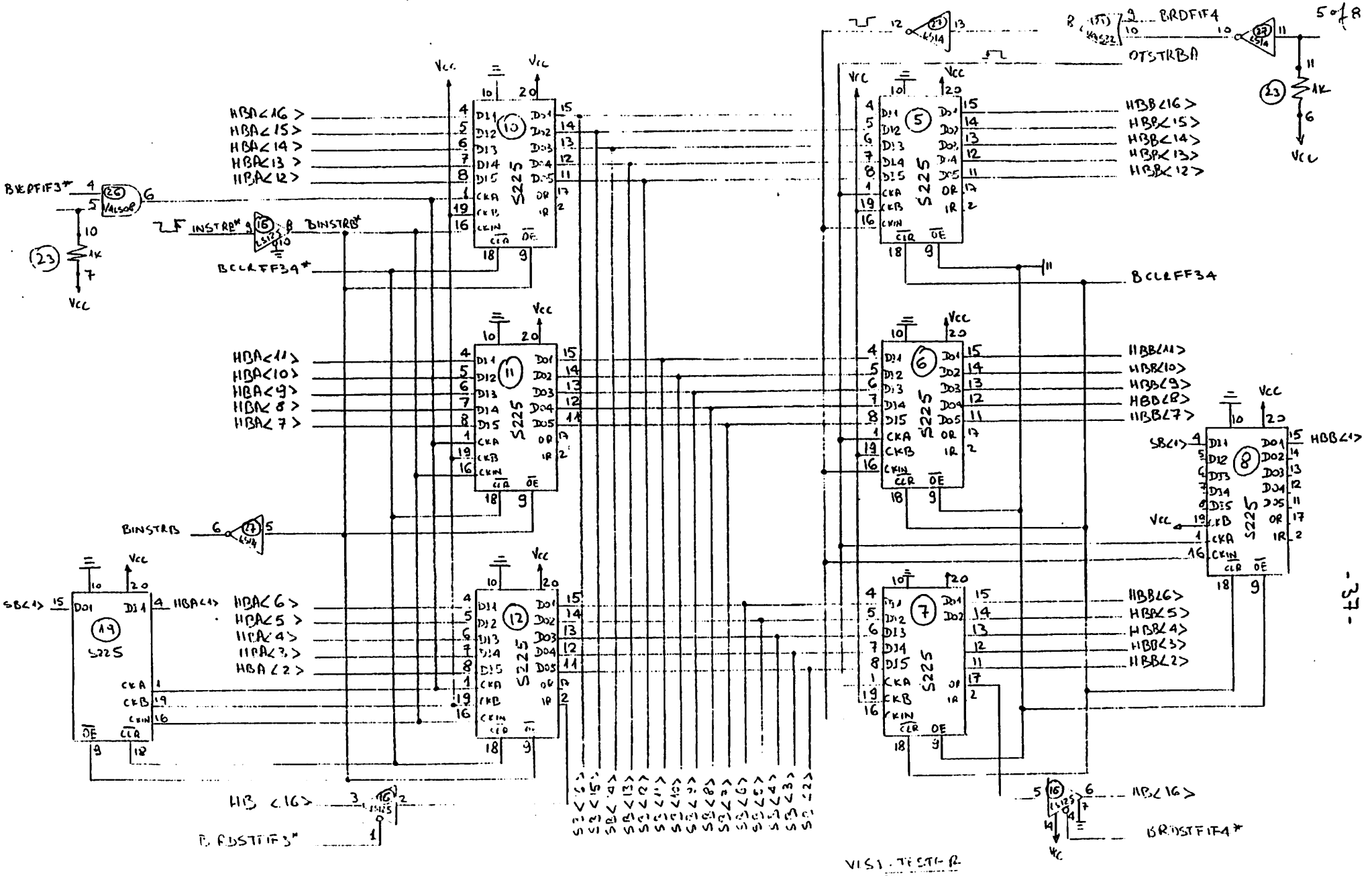
I/O BUFFERS

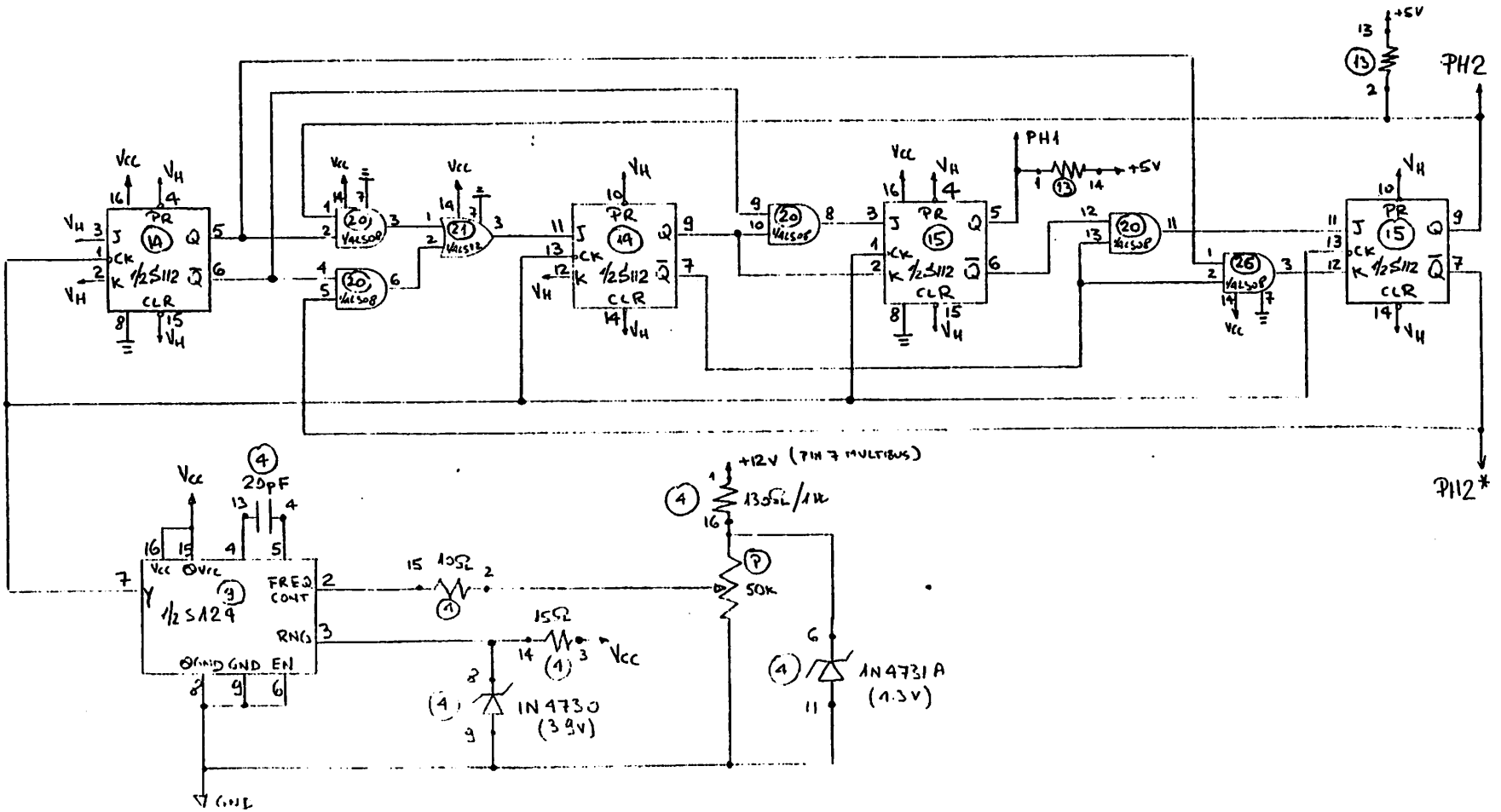


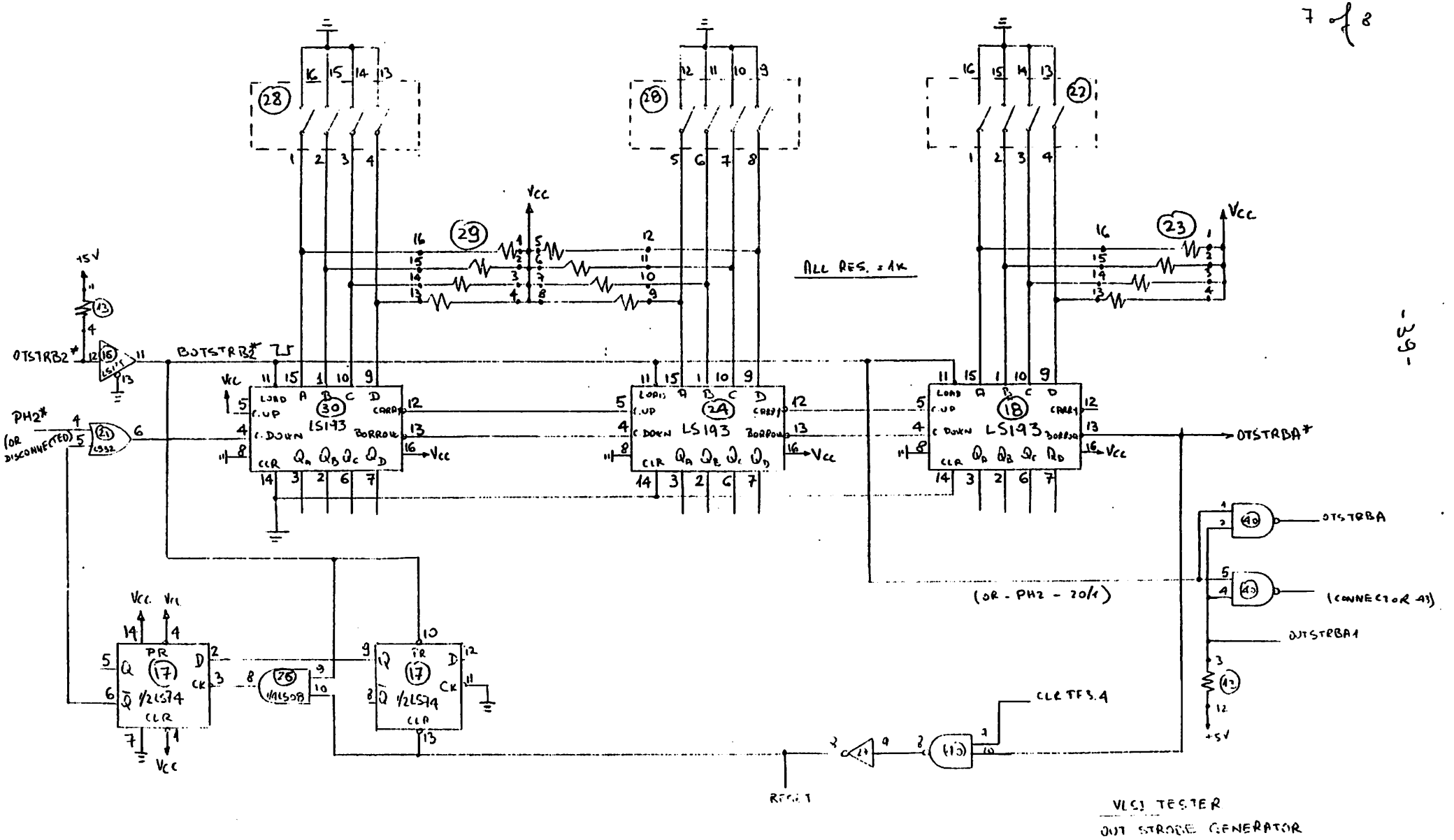

VLSI TESTER

SELECTORS & BUFFERS


5 of 8
BRDFIF4
OTSTRBA
BCLRFF34
BINSTRB
BINSTRB*
BCLRFF34*
HBA<16>
HBA<15>
HBA<14>
HBA<13>
HBA<12>
HBA<11>
HBA<10>
HBA<9>
HBA<8>
HBA<7>
HBA<6>
HBA<5>
HBA<4>
HBA<3>
HBA<2>
HBB<16>
HBB<15>
HBB<14>
HBB<13>
HBB<12>
HBB<11>
HBB<10>
HBB<9>
HBB<8>
HBB<7>
HBB<6>
HBB<5>
HBB<4>
HBB<3>
HBB<2>
HBB<1>
HBB<16>
BRDSTFIF4*
HIB<16>
BRDSTFIF3*
VCC
S225

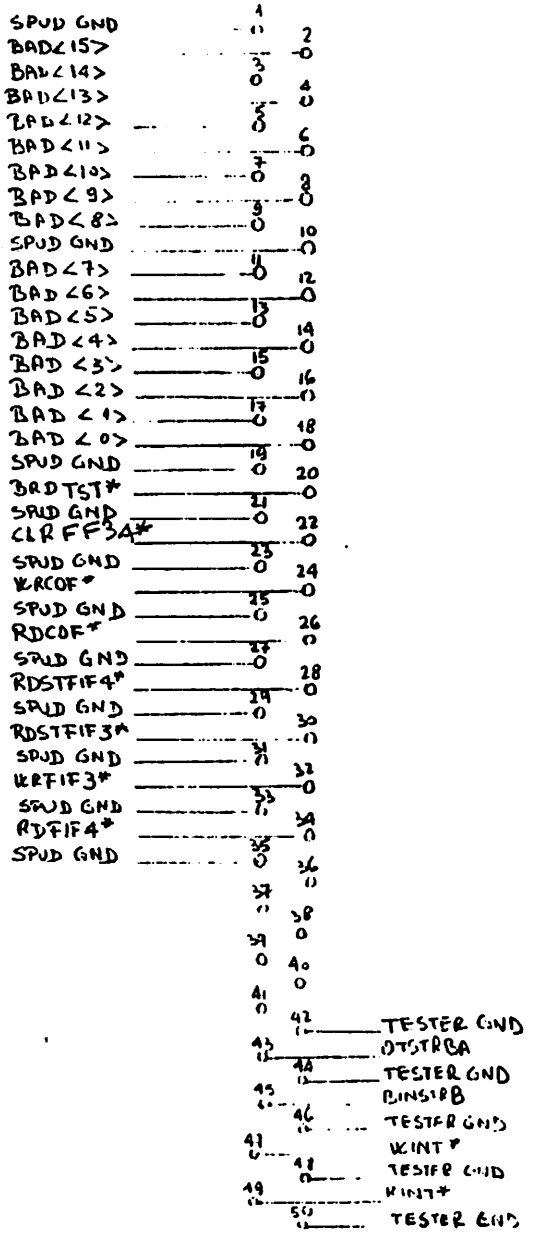



VLSI TESTER
CLOCK GENERATOR



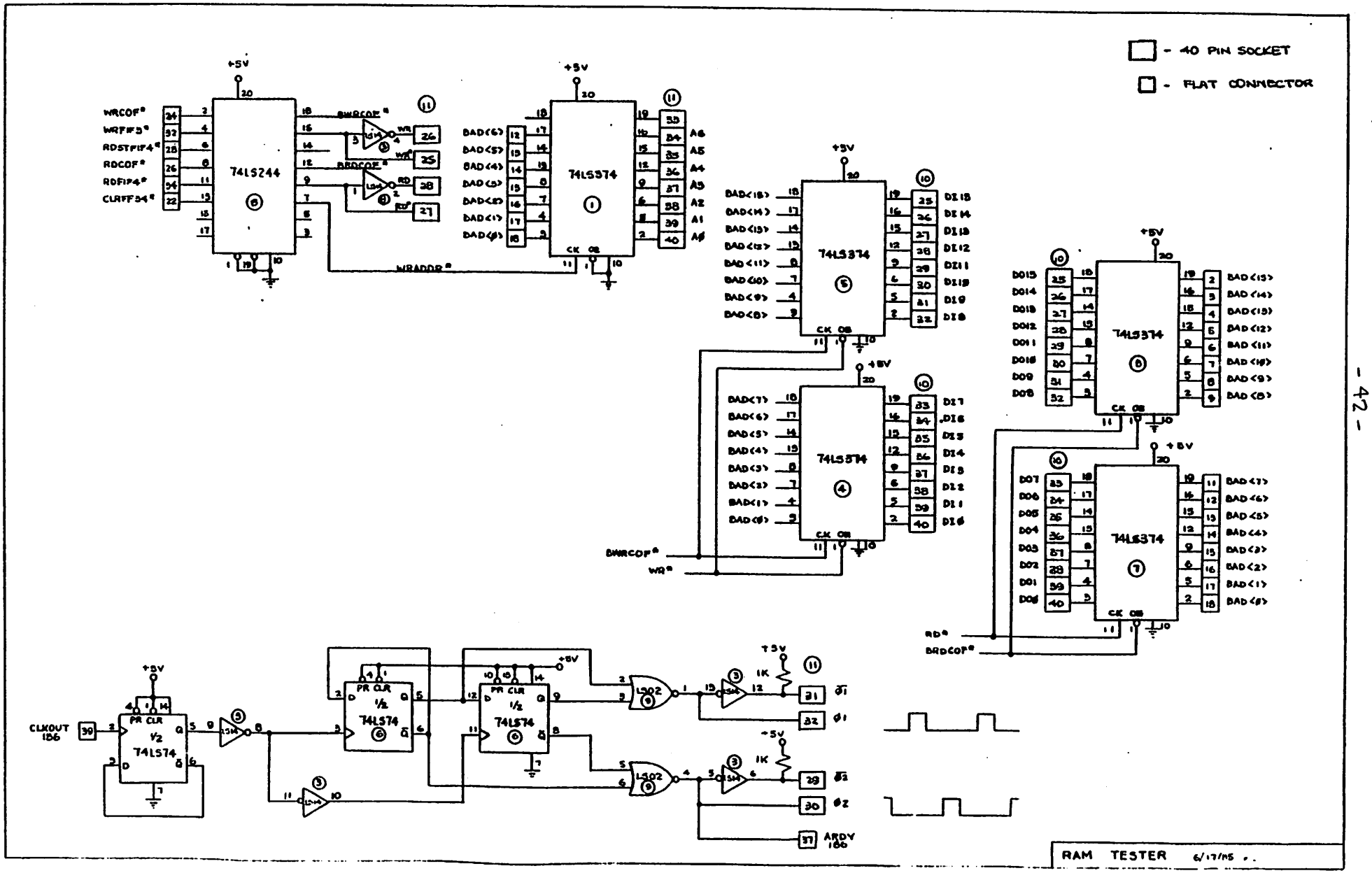

VLSI TESTER

OUT STROBE GENERATOR




SPUD'S VLSI TESTER
PIN DESIGNATION FOR 50-PIN
RIBBON CABLE CONNECTOR (A)

## 15.3. RAM Chips Tester Board


- 40 PIN SOCKET
- FLAT CONNECTOR
74LS244
74LS374
74LS74
74LS02
WRCOF*
WRFIF3*
RDSTFIF4*
RDCOF*
RDFIF4*
CLRFF34*
BWRCOF*
WR*
RD*
BRDCOF*
WRADDR*
CLKOUT 186
ARDY 186
1K
RAM TESTER 6/17/85