Technical Report 85-014

The Terak Tester User's Guide

Vernon L. Chi

Microelectronic Systems Laboratory

The University of North Carolina at Chapel Hill
Department of Computer Science
New West Hall 035 A
Chapel Hill, N.C. 27514

1 Introduction

This document is an introduction to writing programs for chip testing using the Terak tester at the Microelectronic Systems Laboratory (MSL), at the University of North Carolina, Chapel Hill. It is intended to help you understand the testing environment and, more important, the use of the software tool **Gcel.**

An appendix is included which introduces the use of the hardware as well. These tools include a VAX host computer running UNIX, a Terak microcomputer running UCSD Pascal, the special tester interface and test head, and communications facilities between the VAX and the Terak.

2 Scope

This document illustrates the basics needed for using the Terak tester and how the tools are used. No attempt is made to illustrate the entire Gcel language; many test programs will only need to use a small part of it. An example is given, however, which illustrates how heavy use of Gcel can accelerate the execution of a test enormously.

For those needing the full power of Gcel, an appendix is included which presents a complete BNF description of the Gcel syntax, along with semantic annotations. A reasonable familiarity with programming and with reading of BNF should be sufficient for a user to make full use of Gcel.

3 How the software is organised

To test your chip using the Terak tester you must write a test program. This program performs the following basic steps:

1. prepares a set of stimulus vectors (test patterns),
2. presents the stimulus vectors to the DUT (device under test) pins,
3. reads response vectors (test results) from the DUT pins,
4. stores and/or evaluates the response vectors.

These steps are performed cooperatively by a driver program and a special interface subroutine.

The driver program, written in Pascal, performs step 1 and step 4. The special subroutine, written in Gcel, performs steps 2 and 3. This subroutine is declared to be an external procedure in the driver program. These entities are compiled/assembled independently, then linked together to create a complete test program.

In the following sections we discuss the Gcel language in detail and how the driver program and gcel subroutine communicate with each other. Those needing instruction in Pascal are referred to Jensen and Wirth [1], Wilson and Addyman [2], and in particular for UCSD Pascal, UCSD [3].

4 What is Gcel and Why?

4.1 What is Gcel?

Basically, Gcel is a simple programming language, developed specifically for use with the Terak Tester. A Gcel program implements the body of an external procedure of a Pascal driver program. The Gcel procedure provides an interface between the driver program and the tester hardware as follows:

1. the driver passes an array of test vectors as a parameter to the gcel procedure;
2. Gcel "assert" statements transfer the test vectors to the outputs of the tester hardware which are connected to the DUT socket.
3. Gcel "read" statements transfer the response vectors from the inputs of the tester, which are also connected to the DUT socket, to an array.
4. Upon returning, Gcel passes the response vector array back to the driver program as a parameter. A Gcel program consists of a sequence of gcel statements. For example, some Gcel statements, and their associated actions are:

* "assert": present a stimulus vector to the DUT pins.
* "lo", "hi": modify the states of selected DUT pins.

* "if": conditional execution of other statements.
* "while": looping and loop closure.
* "read": capture a response vector from the DUT pins.

4.2 Why Gcel?

Implementing meaningful tests for complex circuits is at best an arduous task. One would like to use the structure and facilities of a high level language in expressing the test. On the Terak hardware, UCSD Pascal is an available tool to address needs for managing the complexity of the testing task.

On the other hand, many tests are speed-critical, or at least have critical segments. Therefore, one would like to be able to run the tester hardware as fast as possible. For this reason, one is tempted to write assembly code to implement speed-critical test segments.

Gcel was designed to provide higher level programming structures while generating high performance object code. It was also designed to hook nicely into a Pascal environment to take advantage of its available tools.

Thus, a highly complex, but non speed-critical test can be written almost entirely in Pascal, with a "one-liner" Gcel program as an interface. Alternatively, a speed-critical segment of a test can be extensively programmed in Gcel, and executed in its entirety by a single call to the Gcel procedure.

In this way, Gcel attempts to offer the user all the power of a high level programming language, while providing a structured escape hatch to achieve the speed performance typical of assembly code.

5 Communication between the Driver Program and the Gcel Routine

A test program consists of a driver program and a Gcel program. The driver program exercises the chip by calling the Gcel interface subroutine. The Gcel subroutine is just an external subroutine of the driver program. The communications are done through the normal parameter passing mechanism.

5.1 How to use Gcel in the driver

In the driver program, you should declare an external procedure with four parameters, as follows:

```
procedure exercise ( var control: array[1..MAXCONTROL] of integer;
                     var stimulus: array[1..MAXINPUTS]  of integer;
                     var response : array[1..MAXOUTPUTS] of integer;
                     var termcd  : integer);
    external;
```

The name of this procedure must be "exercise". The driver calls this procedure in the ordinary way.

5.2 Meanings of the Parameters

A description of the semantics of the parameters passed to the procedure "exercise" follows.

5.2.1 control[i]

This parameter allows the driver program to parameterize the execution behavior of the Gcel procedure, "exercise". An example is the use of the values of "control[i]" as loop closure counts in the Gcel code. This is just one example of many ways in which "control[i]" values can be used to control the execution of a Gcel program.

```
/* This is a Gcel code fragment */
        ...
        repeat control times        /* execute statement 1 control[1] times */
                <statement 1>
        repeat control hold times   /* execute statement 2 control[2] times */
                <statement 2>
        repeat control times        /* execute statement 3 control[2] times */
                <statement 3>
        ...
```

Use of the Gcel "control" statement to access a "control[i]" value will auto-increment through the "control" array passed by the driver program, unless prevented by the "hold" statement.

5.2.2 stimulus[i]

Stimulus signals for the DUT are contained in "stimulus[i]", the test vectors passed to the procedure "exercise". The Gcel "assert" statement performs the transfer of a test vector to the DUT pins via the tester ports.

```
/* Gcel code fragment */
        ...
        assert;             /* send stimulus[1] to tester port 0 (pins 1..16) */
        assert hold @ 2;    /* send stimulus[2] to tester port 2 (pins 33..48) */
        assert @ 1;         /* send stimulus[2] to tester port 1 (pins 17..32) */
        ...
```

Autoincrementing is the default here, as well. The "@" construct establishes a base pin address for transfer of the stimulus vectors. The default base is zero.

5.2.3 response[i]

The response signals of the DUT are returned in "response[i]" by the procedure "exercise". The Gcel "read" statement performs the transfer of the response signal to the "response[i]" array.

```
/* Gcel code fragment */
        ...
        read;               /* copy tester port 0 (pins 1..16) into response[1] */
        read hold @ 3;      /* copy tester port 3 (pins 33..48) into response[2] */
        read @ 2;           /* copy tester port 2 (pins 17..32) into response[2] */
        ...
```

This behavior is symmetric with that of the "assert" statement. The "@" and "hold" constructs behave the same way.

5.2.4 termcd

This is the return status of the procedure "exercise". A value of 0 indicates a normal return, whereas a value of 1 indicates an error return.

```
/* Gcel code fragment */
        ...
        if (pin 9)
                exit;       /* normal return if pin 9 of DUT is high */
```

```
    else
          error;          /* error return otherwise */
    ...
```

6 The Software Environment

This section introduces the software environment in which the test program is developed and run.

Testing is performed on the Terak at MSL. The Terak tester is an LSI-11 based microcomputer system with a modified four-port parallel I/O interface connected to a test head (DUT testing fixture) by ribbon cables. The Terak runs under UCSD Pascal System V2.0; see UCSD [3]. For Terak specific documentation, refer to TERAK [4].

The Pascal driver program is compiled on the Terak under the UCSD system. The driver source code may be written and maintained either on the Terak, or on a VAX under UNIX; communications facilities for file transfer are necessary to support Gcel code, in any event.

The Gcel program must be compiled under UNIX, typically on the Department VAX (Dopey) or the MSL VAX (John), as Gcel is only supported under UNIX. Refer to the manual entry for gcel(local), for details on using the gcel compiler. The object code generated by the gcel compiler must be assembled and linked to the driver program on the Terak, using the UCSD tools.

A utility program called "terminal", running on the Terak, can transfer files to and from a UNIX system, as well as acting as a terminal.

6.1 The general procedure

1) Write your drivers, Gcel programs, (and test input files, if any) on the Dvax;
2) compile the Gcel program on the DVax;
3) transfer the driver source code, compiled Gcel code, and test input files to the Terak;
4) compile the driver program on the Terak;
5) assemble the Gcel code on the Terak;
6) link the driver and the Gcel modules together on the Terak;
7) perform tests on the Terak;
8) transfer results back to the Dvax.

Appendix B works through an example of compiling, transferring and linking.

6.2 UCSD Pascal tools

Each user of the Terak tester should create and maintain a personal working disk. The Terak disk drives are single-sided double-density drives accepting 7 inch soft-sectored floppy disks.

Before being useful, a new disk must be formated. If it is to be used "stand-alone", it must have a bootstrap program written on its first two blocks. To support the tester, it must have at minimum the following files written on it:

```
SYSTEM.PASCAL
SYSTEM.FILER
SYSTEM.8510.QB
SYSTEM.CHARSET
SYSTEM.EDITOR
SYSTEM.COMPILER
SYSTEM.LIBRARY
SYSTEM.MISCINFO
SYSTEM.LINKER
SYSTEM.ASSMBLER
```

```
TERMINAL.CODE
FORMAT.CODE
11.OPCODES
11.ERRORS
```

Appendix B contains a cookbook procedure for configuring a new disk for use with the Terak tester.

6.3 Gcel and UNIX

Gcel is a compiler which is resident on a host computer running UNIX, such as the UNC Computer Science Department VAX (Dopey). A Gcel source code file, "filename.g" is compiled to an object code file, "filename.t". The object module is actually LSI-11 assembly code which is human-readable.

All of the standard UNIX tools are available, such as the editors, version control, makefiles, etc. are available to write and maintain the Gcel source and, if desired, the Pascal source code.

Refer to Appendix B for a detailed example of the entire process of generating a complete test program.

7 Hardware Environment

This section introduces the hardware environment for chip testing.

7.1 Description of the Terak Tester hardware

The Terak, test fixture, and the DUT (your chip) are the three basic components of the hardware configuration. The test fixture consists of a test interface card in the Terak and an external test head containing the DUT socket. The test head is connected to the interface with ribbon cables.

There are several interchangeable test heads to accommodate different packages such as 24-, 40-, and 64-pin DIPs, and 84-pin PGA packages. Standard test heads for these packages are available, while custom test heads can be built by contract to the MSL. The standard test heads map the logical tester pins sequentially onto the physical pins of the DUT socket, i.e., tester pin 1 to DUT pin 1, tester pin 2 to DUT pin 2, etc. Custom fixturing with different mappings may be used if the ultimate performance is required of the tester.

The Terak is the control station on which the test program is executed. The test interface is a four port bidirectional parallel interface to the Terak Q-bus. It is accessed by the Terak as eight consecutive 16-bit words in memory; thus chips with up to 128 pins can be tested. The test program in the Terak sends signals to the interface ports. These signals then pass through the ribbon cables to the test head and finally to the pins of the DUT socket.

The DUT socket on a standard test head is a zero-insertion-force (ZIF) socket. It is typically green, and has a release lever to allow easy insertion and removal of the DUT. The lever should always be in the up position while inserting or removing a chip. Moving the lever to the down position will firmly lock the chip in place, ready for testing.

The standard test heads can accommodate any chips packaged in DIP's up to 64 pins, and 84-pin PGAs. The heads available at this time are for 40-pin (can be used for 24-pin packages) and 64-pin DIPs, and for 84-pin PGAs. For DIPs, the standard head pins are numbered sequentially, starting with pin 1 being nearest the release lever of the ZIF socket, an proceeding counter-clockwise. For PGAs, the pins are numbered sequentially according to the MOSIS bond-out pad specification.

Five volt power and ground are available on the test head on two banana jacks. The red jack is +5 volt power, and the black is ground. On some test heads, further provision is made for a second ("V.hot") power supply. Each standard test head has a set of access points corresponding to the pins of the ZIF socket. Power and ground connections must be made by the user by connecting the appropriate access point to these banana jacks and/or to external power source(s).

Note that the actual pin numbers as used in the testing programs are referring to the pins of the test interface. They do not refer to the pins of the ZIF socket. The mapping between these sets of pins is established by the test head wiring. The standard test heads have hidden this distinction by providing an identity mapping.

Some applications will need a custom mapping, for which a custom test head is required. If at all possible, however, the user should use the available standard test heads. ZIF sockets are in the $75 range, and there is a cost in building a special test head. While it is up to the user to cover the costs, MSL personnel can build a custom test head to specifications, or help the user to build one.

7.2 Procedure for Setting up the Chip for Testing

1) Connect the cables from the Terak to the test head: the *J1* cable to the *J1* socket and the *J2* cable to the *J2* socket, etc., making sure that the pin 1 marks of the cable plugs match those of the sockets.
2) Set the switch on the test head to the "off" position and set the release lever on the ZIF socket to the "open" (up) position.
3) Connect *power supply* and *ground* to the appropriate access pins. (Note: the power supplied by the Terak through the J1 and J2 cables is +5 volts). **Be sure to make the connections to the correct pins or you can fatally damage the chip.**
4) Plug the chip into the test socket, making sure the chip is seated properly, and that pin 1 (marked with a dot on the DIP, or at the notched end of the DIP) is adjacent to the release lever; then lock the chip in place by moving the release lever to the "closed" position. **Handle the chip with care to avoid damage; a static discharge from your body to a chip's pin can kill it.**
5) Turn on the power switch.
6) Perform your tests.
7) Turn off the power switch, open the release lever, and remove the chip from the test socket.

8 Programming Examples

The examples in this section are intended to introduce gcel constructs and driver program techniques. These examples are all real in the sense that they actually operate as advertised on the Terak tester. The gcel compiler as implemented under 4.2BSD UNIX will generate code which can be assembled and linked to the driver program by the Terak under the UCSD Pascal system.

The simple driver programs in examples 1 and 2 illustrate how to assemble and analyze stimulus and response vectors, and how these vectors are passed to and from the gcel procedure, "exercise".

Example 3 introduces some gcel control flow facilities which allow one to move speed-critical test segments out of the driver program, which is slow, and into the gcel procedure, which is relatively fast. A listing of the compiled gcel code, i.e. an LSI-11 assembly code segment which is linkable as a UCSD Pascal procedure, is provided for the benefit of hackers desiring to analyze the gcel code generator.

Examples 4 and 5 are taken from tests which were performed on real custom VLSI circuits. Example 4 shows only the gcel code, and emphasizes how gcel was used to maximize the speed of the Terak tester. Example 5 represents how a simple gcel program can be used with a very baroque driver to enable very complex and intricate tests to be performed.

8.1 Example 1

This example consists of a gcel program and its associated PASCAL driver program. It implements a check of the tester drive and sense electronics and associated device fixturing. The response analysis is intentionally kept minimal; just sufficient to illustrate how to access response vectors. This test will detect stuck-at faults and inter-pin shorts, reporting only that a fault was detected in a given port.

The Gcel Program

```
     /*************************************************/
     /* The gcel program is executed by each call of the procedure "exercise" in the driver */
     /* program.  In this case, stimulus and response vectors are passed as parameters, as is */
     /* (automatically), a termination condition code. The control parameter is not used in this */
     /* example. During compilation, the gcel program is passed through the C preprocessor, */
     /* so comments such as this one can be included in the gcel text.                        */
     /*************************************************/
                    {
                    assert @0;
                    assert @1;
                    assert @2;
                    assert @3;
                    read @0;
                    read @1;
                    read @2;
                    read @3;
                    }
```

Explanation:

Each "assert" sends signals to a different word and each "read" reads from a different word. Each word is sixteen bits and is mapped onto sixteen pins of the DUT. The standard test head maps bits 0..15 of word[0] to pins 1..16 of the DUT, bits 0..15 of word[1] to pins 17..32 of the DUT, etc.

The order of the statements must be related to the order of the test vectors stored in the stimulus array. That is, the first vector in the array will be asserted at word[0], the second at word[1], etc. Similar comments apply to the response vector array.

The Driver Program

```
     /*************************************************/
     /* A useful feature of this driver program is the use of a variant record to specify a pin */
     /* within a given port, and to cast this into an integer type for communication to the */
     /* procedure "exercise"  This is a convenient way to circumvent the tendency for UCSD */
     /* PASCAL to interpret a most-significant pin (bit) specification as an integer sign-bit.  */
     /*************************************************/
{$S+} /* UCSD compiler directive to enable swapping */
program porttest(input,output);

     type  control = array[1..1] of integer;
           vector = array[1..4] of integer;
           pintype = record
                 case integer of
                       0: (int: integer);
                       1: (bit: set of 0..15)
                 end;
```

```
var     ctrl: control;
        stimulus: vector;
        response: vector;
        portno, pinno, termcode: integer;
        pin: pintype;

procedure exercise (var ctrl: control; var stimulus: vector;
                        var response: vector; var termcd: integer);
        external;

procedure initialize;
        begin /* initialize */
        stimulus[1] := 0; stimulus[2] := 0; stimulus[3] := 0; stimulus[4] := 0;
        writeln('This is a program to test the test head and pin drive');
        writeln('electronics.  You may select any (port-number, pin-number)');
        writeln('pair to test, within valid range (ports: 1..4; pins: 1..16).');
        writeln('To terminate test, select out of range (e.g. port 0).');
        writeln;
        end;   /* initialize */

procedure dotest;
        var     i: integer;
        begin /* dotest */
        stimulus[portno] := pin.int;
        exercise(ctrl, stimulus, response, termcode);
        if termcode <> 0 then
                writeln('Gcel execution error.')
        else
        for i := 1 to 4 do
                if stimulus[i] <> response[i] then
                        writeln('       Fault detected in port', i, '.');
        end;   /* dotest */

begin /* porttest */
initialize;
write('Enter port number and pin number to be tested: '); readln(portno, pinno);
while (portno >= 1) and (portno <= 16) and (pinno >= 1) and (pinno <= 16) do
        begin
        pin.bit := [pinno - 1];
        dotest;
        pin.bit := [ ];
        dotest;
        writeln;
        write('Enter next port and pin number: '); readln(portno, pinno);
        end;
end.   /* porttest */
```

8.2 Example 2

This is a gcel program used to test a 40-pin test head. It is similar to example 1, but shows another feature of the gcel language: declaration of ensembles of pins to be driven or sensed as groups.

The Gcel Program

```
/***********************************************************************/
/* If we were to delete the "assert @3" and "read @3" of the Gcel program in example 1, */
/* it would become equivalent to this program.                                          */
/***********************************************************************/
                stimulus    40    pins;
                response    40    pins;

                    {
                    assert;
                    read;
                    };
```

Explanation:

The example here shows the "stimulus" and "response" declarations. These declarations tell the Gcel compiler that when we use an assert or read statement, we want to operate on all 40 pins together. Without these declarations, we must assert 3 times and read 3 times to accomplish the same thing.

While we assert signals to all 40 pins by one gcel "assert" statement, the driver program must assemble a 3-byte array of test vectors in the correct order in the stimulus array. Similarly, the single gcel "read" passes a 3-byte array back to the driver program.

The Driver Program

```
    /***********************************************************************/
    /* The use of the variant record is extended here to aid in the formatting of signals for */
    /* the entire 40-pin group.  Note that use of the variant record to cast types exhibits */
    /* implementation-dependent behavior.  On the Terak tester, UCSD Pascal maps the */
    /* "set" type onto the "integer array" type in the appropriate order.              */
    /***********************************************************************/
{$S+}
program tst40pin(input,output);
    const  msb = 39;    /* bit corresponding to pin 40 */

    type   control = array[1..1] of integer;
           vector = array[1..3] of integer;
           pintype = record
                case integer of
                     0: (vect: vector);
                     1: (bits: set of 0..msb)
                end;
    var    ctrl: control;
           stim, resp: pintype;
           bitnum, termcode: integer;

    procedure exercise (var ctrl: control; var stimulus: vector;
                        var response: vector; var termcd: integer);
           external;

    procedure initialize;
           begin  /* initialize */
           writeln('This program tests a 40-pin test head and pin drive');
           writeln('electronics.  It uses a "marching one/zero" test');
           writeln('pattern to detect stuck-at and other faults.');
```

```
          writeln;
          end;   /* initialize */

    procedure dotest;

          procedure binwrite (pin: pintype);
                var    i: integer;

                begin /* binwrite */
                for i := 0 to msb do
                      begin
                      if i mod 8 = 0 then write(' ');
                      if i in pin.bits then write('1')
                      else write('0');
                      end;
                writeln;
                end; /* binwrite */

          begin  /* dotest */
          exercise(ctrl, stim.vect, resp.vect, termcode);
          if termcode <> 0 then
                writeln('Gcel execution error on pin ', bitnum, '.')
          else if resp.bits <> stim.bits then
                   begin
                      write('stimulus:');
                      binwrite(stim);
                      write('response:');
                      binwrite(resp);
                   writeln;
                      end;
          end;   /* dotest */

    begin  /* tst40pin */
    initialize;
    for bitnum := 0 to msb do
          begin
          stim.bits := [bitnum];
          dotest;
          stim.bits := [0..msb] - [bitnum];
          dotest;
          end;
    end.   /* tst40pin */
```

8.3 Example 3

The previous examples involved calling the gcel procedure "exercise" once for each stimulus-response vector pair. This is exceedingly slow for two reasons. First, because a procedure call involves significant overhead; second, because UCSD Pascal is implemented on a p-machine emulator, which incurs even more overhead.

Gcel provides an escape hatch akin to writing assembler code for the LSI-11 processor in the Terak, allowing the user to migrate speed-critical code segments from the driver program to the gcel program, where they can be executed at the full speed of the LSI-11.

Gcel allows for long high-speed sequences by implementing loop constructs. Several loop closure mechanisms are featured, as delineated in Appendix A. This example will introduce the use of loops in gcel using the "repeat" statement and the "control" parameter for loop closure.

The Gcel Program

```
/*****************************************************************************/
/* Line numbers are used in this listing for purposes of discussion. The actual gcel code */
/* cannot have line numbers.  Remember that this entire gcel program is executed once */
/* for every call of "exercise" by the driver program.                                    */
/*****************************************************************************/

 1:    push sp;
 2:    repeat control times {
 3:            pop sp;
 4:            push sp;
 5:            repeat control hold times {
 6:                    assert @ 0;
 7:                    assert @ 1;
 8:                    read @ 2;
 9:            }
10:    }
```

Explanation:

line 1: Save the stimulus array index, pointing to "stimulus[1]". This allows us to re-use stimulus[1] and stimulus[2] on each pass around the subsequent loop.

line 2: Repeat the execution of lines 3 through 8 "control[1]" times. Since "hold" is not specified, the next control parameter will be "control[2]".

line 3: Reset the stimulus array index to "stimulus[1]",

line 4: and save it again for the next pass through the loop.

line 5: Repeat the execution of lines 6 through 8 "control[2]" times.

line 6: Assert signals from stimulus[1] on pins 1 through 16.

line 7: Assert signals from stimulus[2] on pins 17 through 32. Since "hold" wasn't specified in line 6, we know it's now stimulus[2].

line 8: Read the outputs from pins 33 through 48, and store them in response[X]. X auto-increments; X = 1 the first time through. Thus, the test results will be stored into the array "response[1]" through "response[control[1]*control[2]]".

Gcel compiles its source code to generate an LSI-11 assembly code module. The code generated by compiling the above program is shown here:

```
        .PROC   EXERCISE, 4
; INTERFACE
; ------
; EXERCISE(VAR CONTROL_ARRAY: ARRAY[1..?] OF INTEGER;
;          VAR STIMULUS_ARRAY: ARRAY[1..?] OF SET OF 1..STIMULUS PINS;
;          VAR RESPONSE_ARRAY: ARRAY[1..?] OF SET OF 1..RESPONSE PINS;
;          VAR TERM_CODE: BOOLEAN);
CNTLADR .EQU    8.          ; CONTROL BUFFER ADDRESS :
STIMADR .EQU    6.          ; INPUT BUFFER ADDRESS : SET OF 1..STIMWIDTH
RESPADR .EQU    4.          ; OUTPUT BUFFER ADDRESS : SET OF 1..RESPWIDTH
TERMCD  .EQU    2.          ; TERMINATION CODE 0=OK, 1=ERROR
RETN    .EQU    0.          ; RETURN ADDRESS IS ON TOP
NPARMS  .EQU    8.          ; 10 BYTES OF PARAMETERS
```

```
; PARALLEL PORT ADDRESS
PPORT  .EQU   0176540

; EMT TRAP ADDRESSES
EMTPC  .EQU   030
EMTPS  .EQU   032

; INITIALIZATION CODE
       MOV    R5,@#SR5        ; SAVE REGISTERS
       MOV    R4,@#SR4
       MOV    R3,@#SR3
       MOV    R2,@#SR2
       MOV    SP,@#OLDSP      ; SAVE STACK POINTER

       MOV    CNTLADR(SP),@#CP
       MOV    STIMADR(SP),R5  ; INITIALIZE REGISTERS
       MOV    RESPADR(SP),R4
       MOV    #PPORT,R1       ; KEEP PPORT ADDRESS IN R1 FOR SPEEDY ACCESS
; SET PRIORITY UP
       MOV    #START,@#EMTPC
       MOV    #0340,@#EMTPS
       EMT    0
; CLEAN UP CODE
       MOV    @#SR5,R5        ; RESTORE REGISTERS
       MOV    @#SR4,R4
       MOV    @#SR3,R3
       MOV    @#SR2,R2
       MOV    R0,@TERMCD(SP)  ; RETURN STATUS IN TERMCD
       MOV    (SP)+,R0        ; RETURN ADDRESS
       ADD    #NPARMS,SP      ; DISCARD PARMS
       JMP    @R0
START:
       MOV    SP,@#OLDSP      ; SAVE SP
       MOV R5,-(SP)
       MOV @#CP,R0
       MOV (R0)+,@#CN1
       MOV R0,@#CP
RB0:
       MOV (SP)+,R5
       MOV R5,-(SP)
       MOV @#CP,R0
       MOV (R0),@#CN2
RB1:
       MOV (R5)+,(R1)
       MOV (R5)+,@#PPORT+2
       MOV @#PPORT+4,(R4)+
       DEC @#CN2
       BEQ JSKP2
       JMP RB1
JSKP2:
       DEC @#CN1
       BEQ JSKP3
       JMP RB0
```

```
JSKP3:
      MOV    #0,R0
EXIT:
      MOV    @#OLDSP,SP    ; RESTORE THE SP
      RTI

; DATA AREA
      OLDSP      .WORD
      SR5   .WORD
      SR4   .WORD
      SR3   .WORD
      SR2   .WORD
      CN1   .WORD
      CN2   .WORD
      CN3   .WORD
      CN4   .WORD
      CN5   .WORD
      T     .WORD
      CP    .WORD
.END
```

8.4 Example 4

This example is an actual test. It was used to check the operation of Bishop's Self-Tracker[7]. This system is composed of a photo-sensor array integrated on the same chip with a processor; therefore, some of the inputs to the DUT were optical and therefore not visible, per se, in the test program code.

The purpose of this example is to illustrate how to use gcel to get the highest speed performance possible from the Terak tester. Secondarily, it introduces more features of the gcel language.

Notice that the gcel code in this example performs many clock cycles of activity for each call of "exercise" by the driver program (not included in this example).

```
/*************************************************************/
/* The gcel features introduced here are a macro facility for symbolic references, the direct */
/* pin setting statements "hi" and "lo", the phase declarations "phi1" and "phi2", and */
/* their associated "clock" statement.                                                  */
/*************************************************************/
            #define halfcells 108
            #define shift pin 3
            #define strobe pin 4
            #define reset pin 5
            #define top pin 6
            #define bot pin 7

            phi1 pin 1;
            phi2 pin 2;
            response 7 pins;

            lo shift strobe;
            hi reset;
            clock 20;
            lo reset;
```

```
clock control;
hi strobe;
clock 10;
lo strobe;
clock 10;
repeat halfcells times
        {
        read;
        hi shift;
        clock 10;
        lo shift;
        clock 10;
        }
hi reset;
```

Explanation:

Macros: Gcel code is first passed through the C pre-processor. Thus, all of its macro facilities are available. This example shows the use of the "define" facility to provide mnemonic references to pins of the DUT. Other facilities, such as file inclusion, parameterized macros, and conditional definitions are also provided. Refer to the UNIX manual [5] entries for "cc(1)" and "m4(1)" and to Kernighan and Ritchie[6], for detailed information regarding the pre-processor.

Setting pins: Gcel allows immediate, in-line specifications for setting specific pins high and low. A "hi" ("lo") command followed by a pin list (terminated by a ";") will set the listed pins high (low) without affecting any unreferenced pins.

Clocking: Gcel provides automatic clock generation for two phase non-overlapping clocks. The clock phases are bound to pin numbers using the "phi1" and "phi2" declarations. The "clock" statement causes phase 1 to be driven high, then low, then phase 2 high, then low. The value following "clock" specifies the number of repetitions of this clock cycle to execute.

8.5 Example 5

This a complete working example. It was used to test Pixel-Planes chips preparatory to integration of the chips into the (now functioning) experimental system. While much of the driver program may not be of general interest, it is included both for completeness, and to give the reader a flavor of how a very complex and involved test can be supported by the Terak Tester.

In this case, a simple Gcel procedure is used. The test is designed to demonstrate logical correctness, and was not speed-critical. The complexity of the test itself was therefore managed better in the Pascal driver, which is exactly how it was done.

This strategy separates test data generation from the actual testing. The stimulus array is read from external files. The test responses along with the input vectors are written onto an output file. This mechanism allows communication between a simulator and the tester. The simulator generates pre-processed test data, while the driver program performs the running of the test.

8.5.1 The Gcel program

```
#define phi1 pin 31
#define phi2 pin 30
#define px1 pin 29
#define px2 pin 28

stimulus    31    pins;
response    9     pins;

repeat control times
      {
      assert;
      hi phi1 px1;
      lo phi1 px1;
      hi phi2 px2;
      lo phi2 px2;
      read @2;
      };
```

8.5.2 The Driver Program

```
      /***********************************************************/
      /* Driver Program for chip testing: accepts data from an input data file, generates control */
      /* and stimulus vector arrays with proper values, executes the "exercise" procedure, reads */
      /* back response vectors and prints the input data and the test results in a useful format */
      /***********************************************************/

program pxpl30test();

const  ArraySize = 2000;
       HalfSize = 1000;
       CASize = 800;
       ComLine = 50;

type   CntlArray = array[1..1] of integer;
```

```
        StimArray = array[1..ArraySize] of integer;
        RespArray = array[1..HalfSize] of integer;

var     control:     CntlArray;
        stimulus:    StimArray;
        response:    RespArray;
        termcd:      integer;
        fin:         text;        /* input file pointer */
        fout:        text;        /* output file pointer */
        filename:    string; /* user-entered file name */
        ComArray:    array[1..CASize] of char;
        input1:      array[1..4]  of char;
        GlobalTOut:  char;
        TempCom:     integer;

procedure exercise( var control: CntlArray;
                    var stimulus: StimArray;
                    var response: RespArray;
                    var termcd: integer);
        external;

/* returns the integer value of a hex character */

function hexdec(ch: char): integer;
var i: integer;

begin /* hexdec */
        case ch of
                '0': i := 0; '1': i := 1; '2': i := 2; '3': i := 3;
                '4': i := 4; '5': i := 5; '6': i := 6; '7': i := 7;
                '8': i := 8; '9': i := 9; 'A': i := 10; 'a': i := 10;
                'B': i := 11; 'b': i := 11; 'C': i := 12; 'c': i := 12;
                'D': i := 13; 'd': i := 13; 'E': i := 14; 'e': i := 14;
                'F': i := 15; 'f': i := 15
        end;
        hexdec := i
end; /* hexdec */

/* gets the input data file name from the terminal, reads and converts hex data from file, */
/* and puts data into stimulus vector.                                                     */
procedure getdata(var control: CntlArray; var stimulus: StimArray);

var i, j, count: integer;       /* temporary storage */
        ch: char;                       /* temporary */

begin /* getdata */

        /* get test input data file name */
        writeln;
        writeln;
        write('Enter name of data file please: ');
        readln(filename);
        if pos('.', filename)=0 then
                filename := concat(filename, '.TEXT');
```

```
reset(fin, filename);

writeln;
write ('processing begins .');

j := 1;
while not eof(fin) do
        begin
        /* avoid looping */
        if (j > ArraySize) then
                begin
                close(fin);
                exit(getdata)
                end;

        /* skip comments */
        if (j = 1) then read(fin, ch);
        while (ch = '|') do
                begin
                readln(fin);
                read(fin, ch)
                end;

        /* read in the first stimulus vector, put into input1[ ] */
        input1[1] := ch;
        i := 2;
        while not eoln(fin) do
                begin
                read(fin, ch);
                input1[i] := ch;
                i := i + 1
                end;

        /* convert hex input vector in input1[ ] to integer
        and put it into stimulus[ ] */
        stimulus[j] := 0;
        i := 1;
        while (i <  5) do
                begin
                stimulus[j] := stimulus[j] * 16 + hexdec(input1[i]);
                i := i + 1
                end;

        /* read in the second stimulus vector, put into input2[ ] */
        i := 1;
        readln(fin);
        while not eoln(fin) do
                begin
                read(fin, ch);
                input2[i] := ch;
                i := i + 1
                end;
```

```
            j := j + 1;
            stimulus[j] := 0;
            /* convert hex input vector in input2[ ] to integer
                  and put it into stimulus[ ] */
            i := 1;
            while (i <  5) do
                  begin
                  stimulus[j] := stimulus[j] * 16 + hexdec(input2 [i]);
                  i := i + 1
                  end;

            /* ignore the response vector - a string of integers */
            readln(fin);
            while not eoln(fin) do read(fin, ch);
            readln(fin);
            read(fin, ch);


            j := j + 1;
            end;
      write ('   processing complete');
      close(fin);

      control[1] := j div 2;

      end; /* getdata */

function dechex(number : integer): char;
var   ch:    char;
begin
      case number of
            0: ch := '0'; 1: ch := '1'; 2: ch := '2'; 3: ch := '3';
            4: ch := '4'; 5: ch := '5'; 6: ch := '6'; 7: ch := '7';
            8: ch := '8'; 9: ch := '9'; 10: ch := 'A'; 11: ch := 'B';
            12: ch := 'C'; 13: ch := 'D'; 14: ch := 'E'; 15: ch := 'F'
      end;
      dechex := ch
end; /* dechex */

/*  print the parameter - number in hex to the output file  */
procedure pstring(number: integer);
var   i, temp:      integer;
      NegNum: boolean;
      TempStr:     array[1..4] of char;
begin

      /*  convert the number into hexdecimal  */
      i := 1;
      if (number < 0) then
            begin
            NegNum := true;
            number := number + 32767 + 1;
            end
      else
```

```
                NegNum := false;

        while (i < 5) do
                begin
                if (NegNum and (i = 4)) then
                        temp := (number + 8) mod 16
                else
                        temp := number mod 16;

                TempStr[i] := dechex(temp);
                number := number div 16;
                i := i + 1
        end;

        i := 1;
        while (i < 5) do
        begin
                write(fout, TempStr[5 - i]);
                i := i + 1
        end
end;

/*  print comment from position StartPos to the output file  */
procedure pcomment(StartPos: integer);
var     i, Temp:        integer;

begin
        if (StartPos = 1) then
                write (fout, '    | ')
        else
                write (fout, '    | ');

        if (TempCom > StartPos + 49) then
                Temp := StartPos + 49
        else
                Temp := TempCom;
        i := StartPos;
        while (i <= Temp) do
                begin
                write(fout, ComArray[i]);
                i := i + 1
                end
end;

procedure printout(var response: RespArray; var j: integer);
var i, Temp: integer;

begin

        /* print input data - 1st line  of output */
        i := 1;
        while (i < 5) do
                begin
```

```
          write(fout, input2[i]);
          i := i + 1
          end;
  write(fout, ' ');

  i := 1;
  while (i < 5) do
          begin
          write(fout, input1[i]);
          i := i + 1
          end;

  write (fout, '   ');
  write (fout, GlobalTOut);
  write (fout, ' ');

  i := 8;
  while (i > 0) do
          begin
          write (fout, OutStr[9 - i]);
          i := i - 1
          end;

  if (TempCom > 1) then pcomment(1);
          writeln(fout);

  /* print the second line to output file */

  /* print stimulus vector */
  Temp := 2 * j;
  pstring(stimulus[Temp]);
  write(fout, ' ');
  pstring(stimulus[Temp - 1]);
  write(fout, '   ');

  GlobalTOut := chr((response[j] mod 2) + 48);
  response[j] := response[j] div 2;
  write (fout, GlobalTOut);
  write (fout, ' ');

  i := 8;
  while (i > 0) do
          begin
          response[j] := response[j] div 2;
          OutStr[i] := chr((response[j] mod 2) + 48);
          i := i - 1
          end;

  i := 8;
  while (i > 0) do
          begin
          write(fout, OutStr[9 - i]);
          i := i - 1
          end;

     if (TempCom > ComLine) then pcomment(ComLine + 1);
          writeln(fout);

     Temp := 2 *  ComLine;
     while (Temp < TempCom) do
          begin
          write(fout, '                    ');
          pcomment(Temp + 1);
          writeln (fout);
          Temp := Temp + ComLine
          end;

     writeln(fout)
end;   /* printout */


procedure output(var response: RespArray);
var
     i, j, count:     integer;
     chipid:      integer;
     ch:            char;
     outfile:     string;
     flag:        boolean;
     outfilefull:  boolean;
begin /* output */

     reset(fin, filename);

     /* get the output file name */
     writeln;
     writeln;
     write('Enter name of output data file please: ');
     readln(outfile);
     if pos('.', outfile)=0 then
          outfile := concat(outfile, '.TEXT');

     rewrite(fout, outfile);

     /* get the chip id # under test */
     writeln; writeln;
     write('Enter chip id number please: ');
     readln(chipid);

     /* print output heading into the output file */
     writeln(fout, 'PXPL 3.0 - - Chip #', chipid:2, ' tested on file ', filename);
     writeln(fout);
     writeln(fout, 'input      output      comments');
     writeln(fout);

     writeln;
     write ('processing begins .');
     j := 1;
```

```
while (j <= control[1]) do
        begin
        /* avoid looping */
        if (j > ArraySize) then
                begin
                close(fin);
                exit(output)
                end;

        /* put comments into ComArray for CASize characters at most */
        TempCom := 1;
        if (j = 1) then read(fin, ch);

        flag := false;
        while (ch = '|') do
                begin
                if (flag = true) then
                        begin
                        ComArray[TempCom] := ';';
                        TempCom := TempCom + 1;
                        ComArray[TempCom] := ' ';
                        TempCom := TempCom + 1;
                        ComArray[TempCom] := ' ';
                        TempCom := TempCom + 1
                        end
                else flag := true;

                while ((not eoln(fin)) and (TempCom < CASize - 4)) do
                        begin
                        read(fin, ch);
                        ComArray[TempCom] := ch;
                        TempCom := TempCom + 1
                        end;
                readln(fin);
                read(fin, ch)
                end;
        TempCom := TempCom - 1;

        /* read in the first stimulus vector, put into input1[ ] */
        input1[1] := ch;
        i := 2;
        while not eoln(fin) do
                begin
                read(fin, ch);
                input1[i] := ch;
                i := i + 1
                end;

        /* read in the second stimulus vector, put into input2[ ] */
        i := 1;
        readln(fin);
        while not eoln(fin) do
                begin
```

```
                    read(fin, ch);
                    input2[i] := ch;
                    i := i + 1
                    end;

                /* read in the response vector, put into OutStr[ ] */
                readln(fin);
                read(fin, ch);
                GlobalTOut:= ch;
                i := 1;
                repeat  read(fin, ch);
                        OutStr[i] := ch;
                        i := i + 1
                until   i > 8;

                printout(response, j);

                readln(fin);
                read(fin, ch);

                j:= j + 1;
        end;    /* while */

        write('   processing complete');

        close(fin);
        close(fout, lock)
end;    /* output  */

procedure init(var stimulus: StimArray; var response: RespArray);
var     i:      integer;

begin
        for i := 1 to Arraysize do
                stimulus[i] := 0;
        for i := 1 to HalfSize do
                response[i] := 0;

end;

begin   /* pxpl30test */
        init(stimulus, response);
        getdata(control, stimulus);

        exercise(control, stimulus, response, termcd);

        if (termcd <> 0) then
                writeln('error in Gceltest', control[1])
        else
                output(response)
end.  /* pxpl30test */
```

9 Appendix A: Annotated Gcel syntax

Words enclosed in ' ' are key words of the Gcel language. Words enclosed in < > are nonterminal symbols in the Gcel grammer. Expressions enclosed in [ ] are optional.

Grammer rules are written as "<nonterminal> : <alternative> | <alternative>;" just as in YACC. Notes indented under a rule are a description of the semantics of that rule.

```
<prog>      :   <stmts>
                    A program is a series of statements.
            ;

<stmts>     :   <stmt>
            |   <stmts> <stmt>
            ;

<stmt>      :   'lo' <pins> ';'
                    Set the pins in the list to 0.  If all of the pins are in
                    the same word, they will be set to 0 simultaneously.
                    No other pins are affected.
            |   'hi' <pins> ';'
                    Set the pins in the list to 1.  If all of the pins are in
                    the same word, they will be set to 1 simultaneously.
                    No other pins are affected.

            |   'assert' [hold] ['@' <number>] ';'
                    Transfer words from the stimulus array position determined
                    by the stimulus pointer to the pins.
                    The number of words transferred is dependent on the number
                    of pins declared in the stimulus declaration.
                    The first word is transferred to a contiguously numbered
                    sixteen pin group, starting with pin number 16* < number > +1;
                    the next word is transferred to the next sixteen pin group,
                    i.e., starting with pin number 16 * (< number > +1) + 1, and so on.
                    The default value for <number> here is zero.
                    If the "hold" option is specified, the stimulus pointer is
                    not incremented, otherwise it is incremented for each word
                    transferred.
            |   'read' [hold] ['@' <number>] ';'
                    Transfer words from the pins to the response array position
                    determined by the response pointer.
                    The number of words transferred is dependent on the number
                    of pins declared in the response declaration.
                    The first word is transferred from a contiguously numbered
                    sixteen pin group, starting with pin number 16* < number > +1;
                    the next word is transferred from the next sixteen pin group,
                    i.e., starting with pin number 16 * (< number > +1) + 1, and so on.
                    The default value for <number> here is zero.
                    If the "hold" option is specified, the response pointer is
                    not incremented, otherwise it is incremented for each word
                    transferred.
            |   'clock' <value> ';'
                    Run the specified number of non-overlapping 2-phase clock
                    cycles.  The pins affected are those specified in the
```

```
                        phi1 and phi2 declarations.
             |    'bump' <reg> ';'
                        Increment the register by two.  This is used for addressing
                        purposes; therefore the register will point to the next word.
             |    'buzz' <value> ';'
                        Execute a tight loop for this number of cycles.  Used to
                        get a delay when nothing else is going on.
             |    'repeat' <value> 'times' <stmt>
                        Repeat the <stmt> the number of times given in <value>.
                        These nest up to up to 5 deep.
             |    'repeat' <stmt>
                        Repeat the <stmt> forever.
             |    'do' <stmt> 'while' '(' <cond> ')' ';'
                        Repeat the <stmt> until the <cond> is FALSE.  The condition
                        is tested after the execution of the <stmt>.  This is
                        slightly more efficient than the while statement.
             |    'while' '(' <cond> ')' <stmt>
                        Repeat the <stmt> until the <cond> is FALSE.  The condition
                        is tested before each execution of the <stmt>
             |    'if' '(' <cond> ')' <stmt>
                        Execute the <stmt> only if the <cond> is true.
             |    'if' '(' <cond> ')' <stmt> 'else' <stmt>
                        Execute the first <stmt> if the <cond> is TRUE, otherwise
                        execute the second <stmt>.
             |    'push' <reg> ';'
                        Push the indicated register on the pushdown stack.  This
                        is the only way to save values.  This is useful, for
                        example if you want to reuse values from the stimulus
                        vector in a loop.
             |    'push' <value> ';'
                        Push the indicated value on the stack.  This is the only
                        way to get strange values into registers.
             |    'pop' <reg> ';'
                        Pop the top value from the stack and place it in the
                        indicated register.
             |    'pop' ';'
                        Throw away the top value from the stack.
             |    'exit' ';'
                        Exit with a zero TERMCD.
             |    'error' ';'
                        Exit with a one TERMCD.
             |    '{' <stmts> '}'
                        Allow a group of <stmts> wherever a single <stmt> is
                        allowed.
             |    <decl> ';'
                        These are the declarations.
             |    ';'
                        Null statements are legal.
             ;

<pins>       :    'pin' <number>
                        Specifies a particular pin on the chip.  Legal values
                        range from 1 to 64.
```

```
            |   <pins> 'pin' <number>
            ;

<cond>      :   'pin' <number>
                    TRUE if the indicated pin is 1, FALSE if it is 0.
            |   'not' <cond>
                    Inverts the value of the condition.  Highest precedence.
            |   <cond> 'and' <cond>
                    Just what you would expect. Lower precedence than "not"
                    and higher than "or".
            |   <cond> 'or' <cond>
                    Just what you would expect.  Lowest precedence.
            |   '(' <cond> ')'
                    You can have parenthesized expressions.
            ;

<decl>      :   'phi1' 'pin' <number>
                    Specifies the pin to be used as phi1 for 2-phase clocking
                    with the clock command.
            |   'phi2' 'pin' <number>
                    Specifies the pin to be used as phi2 for 2-phase clocking
                    with the clock command.
            |   'stimulus' <number> 'pins'
                    Specify the number of pins of stimulus for the chip.
                    This value is used to compute the number of words
                    transferred from the stimulus array for each assert
                    command.  Pins up to the next 16 bit boundary are
                    affected by the transfer.
            |   'response' <number> 'pins'
                    Specify the number of pins of response for the chip.
                    This value is used to compute the number of words
                    transferred to the response array for each read command.
            ;

<value>     :   <number>
                    The specified number is used as the value.
            |   'control' [hold]
                    The value at the position of the control array determined
                    by the control pointer, is used as the value.  If "hold"
                    is specified, the control pointer is NOT incremented.  This
                    is useful for specifying variable repeat counts from your
                    Pascal driver program.  For example you might want
                    stimulus vectors of different lengths on different runs.
            |   'top' [hold]
                    Use the value at the top of the stack as the value.  If
                    "hold" is specified, the stack is not popped.
            ;

<reg>       :   'sp'
                    The stimulus pointer; it is initialized to the first
                    position of the stimulus array.
            |   'rp'
                    The response pointer; it is initialized to the first
```

```
                                position of the response array.
                  |     'cp'
                                The control pointer; it is initialized to the first
                                position of the control array.
                  |     't'
                                A temporary register.
                  ;

<number>          :     <digit>
                  |     <number> <digit>
                  ;

<digit>           :     '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' | '0'
                  ;
```

10 Appendix B: Nitty-gritty details and cookbook procedures

There are numerous procedural details to deal with in order to effectively use the Terak Tester to test your chips. These include setting up your UNIX and UCSD environments, and communication between them. This appendix provides cookbook procedures and step-by-step examples to help overcome the learning effort for the first time user.

In examples throughout this appendix, the following font conventions are used:

* UNIX prompts are shown in a `Typewriter` font.
* UCSD Pascal prompts are shown in a Sans Serif font.
* User responses are shown in a *Slanted Roman* font.
* The symbol "↵" indicates a user-entered "return" character.
* Comments are enclosed in "/* */" delimiters.
* A character preceded by a "↑" indicates "control" character.

10.1 The UNIX environment

Documents are commercially available for your reference [5, 6]. It is presumed you are sufficiently conversant with UNIX to be able to use the command shell, an editor, and other basic UNIX utilities. Instruction on UNIX, being beyond the scope of this document, is best obtained from your local UNIX hackers, if you need it.

As a Terak Tester user, you should login to UNIX as "terak", unless you have your own login. A password is available from MSL personnel. When you first login, create a sub-directory with a name of your choice; subsequently, always work within this sub-directory. There is no mechanism to keep users from walking on each other's files other than common courtesy and good sense. By working only within your own directory, you are protecting other users from your mistakes, as well as providing yourself a barrier against theirs. **This will only work if every user observes this protocol.** Do it. It's for your own protection.

10.2 The UCSD/Terak environment

General documentation regarding the UCSD Pascal system [3] and the Terak implementation [4] are available. These should be referenced, as necessary while using the Terak. This section is intended to speed you through some of the effort common to all users of the Terak Tester.

Every user should have at least one "working" disk, and should **refrain from using anyone else's.** The following procedure will configure a disk which is ready for you to use in your testing efforts.

You need a 7-inch, single-sided, double-density, soft-sectored (with the sector sync hole near the center rather than near the outside of the disk) flexible disk. A master template disk, named "TESTER:" is available in the MSL, adjacent to the Terak. Insert this disk into the Terak's drive #0, and your new working disk into drive #1. Power up the Terak. When bootstrapping is complete, you can format your disk as follows:

```
Command:E(dit.R(un.F(ile.C(omp.L(ink.X(ecute.A(ssem. D(ebug.?[II.0]  X

Execute what file?  FORMAT↵

TERAK DISK FORMAT UTILITY * VER. 1-01

0 = Single-sided. Single-density (494 blocks)
1 = Single-sided. Double-density (1140 blocks)
2 = Double-sided. Double-density (2280 blocks)
S = STOP
?  1  -OK

Drive Number (0.1.2 or 3)
```

```
? 1 -OK

PLACE DISK IN DRIVE 1 - TYPE ANY KEY WHEN READY↵

Cylinder (x10): 01234567
FORMAT COMPLETE - NO ERRORS DETECTED

0 = Single-sided, Single-density (494 blocks)
1 = Single-sided, Double-density (1140 blocks)
2 = Double-sided, Double-density (2280 blocks)
S = STOP
? S  /* The "S" must be explicitly upper case here. */

?REEBOOT ADVISED? /* Do reboot the system. */
```

You will also need a basic set of supporting files on your disk. These are exactly the same as those on the "TESTER:" disk. In order to get them on your disk, do a "volume to volume transfer". Assuming you have just formated your disk and rebooted the system, the procedure is as follows:

```
Command:E(dit,R(un,F(ile,C(omp,L(ink,X(ecute,A(ssem, D(ebug,?[II.0]  F
Filer: G(et, S(ave, W(hat, N(ew, L(dir, R(em, C(hng, T(rans, D(ate, Q(uit [C.4]  T
Transfer what file ?   TESTER:↵
To where ?   #5↵
Transfer 1140 blocks ? (Y/N)   Y
TESTER:       --> #5: /* This is actually below the prompt line on the Terak screen. */
Filer: G(et, S(ave, W(hat, N(ew, L(dir, R(em, C(hng, T(rans, D(ate, Q(uit [C.4]

/* Remove the "TESTER:" disk from Drive #0 and re-file it. Move your working disk */
/* from drive #1 to drive #0 and reboot the system.                              */
Command:E(dit,R(un,F(ile,C(omp,L(ink,X(ecute,A(ssem, D(ebug,?[II.0]  F
Filer: G(et, S(ave, W(hat, N(ew, L(dir, R(em, C(hng, T(rans, D(ate, Q(uit [C.4]  C
Change what file ?   TESTER:↵
Change to what ?   NAME:↵ /* Name of your choice; ":" is required. */
TESTER:       --> NAME: /* This is actually below the prompt line on the Terak screen. */
Filer: G(et, S(ave, W(hat, N(ew, L(dir, R(em, C(hng, T(rans, D(ate, Q(uit [C.4]  Q
```

If all goes according to this scenario, your disk is now ready to use. The UCSD system is relatively friendly, and if things don't go exactly as shown, error recovery is generally pretty straightforward. In a pinch, refer to the documents cited above.

10.3 Host to Terak communications

Communications between the UNIX Host and the Terak is via a 4800 b/sec serial link. Connect the Terak's RS-232 port (a DB-25 connector) to a suitable terminal port which can access the UNIX Host. This port may or may not be routed through the UNCCC PACX (the Gandalf switch).

The "TERMINAL" program on your working disk supports interactive (terminal) communications, as well as file transfers between the UCSD Pascal and UNIX file systems. Assuming you have booted the Terak on your working disk, connected the terminal line, and switched on the connection switch (if any):

```
Command:E(dit,R(un,F(ile,C(omp,L(ink,X(ecute,A(ssem, D(ebug,?[II.0]  X
    Execute what file?  TERMINAL↵

    TERMINAL V02-01 ..  terminal emulator and file transfer utility.
       type DC2 to alpha case toggle. ↑ EOM to command. ↑ NUL to break.

   /* If Host connection is direct, ignore these commented steps. */
   /* PACX prompts are in normal Roman font, comments are in "( )". */
   /*   ↵ (repeat until you get the PACX's attention) */
   /* enter service  21↵ ( this may not get echoed) */
   /* (an optional PACX "message of the day" may appear) */
   /* class 021 start (if "busy" or "unavailable", try again later) */

    UNC Microelectronic Systems Laboratory Vax/4.2 BSD (john) /* Depends on which UNIX
Host, of course. */

    login:    terak↵
    Password:    <password>↵  /* Your password is not echoed. */

   /* UNIX "message of the day" and other environment-dependent stuff will appear. */

    % /* This is the default UNIX shell prompt. */
```

The Terak and the UNIX Host are now connected and communicating. You can now interactively get into your working UNIX directory:

```
    %  cd <directory>↵
    %
```

Once you are in your working UNIX directory, you are ready to transfer files between the Tester and the UNIX Host. The steps are as follows:

```
   ↑ EOM   /* i.e., simultaneously press "ctrl" and "EOM" keys */
   Terminal: S(end file. R(eceive file. T(erminal mode. Q(uit   S

  /* Send a file from your local Tester disk to your UNIX directory */
   Send file: Text only so .TEXT automatically appended to TERAK filenames
   File name on this TERAK (ending .TEXT assumed):  <terakfile>↵
   Enter Unix filename:
   <unixfile>↵
   Transmitting... /* This may only briefly flash on the screen for short files. */
   Terminal: S(end file. R(eceive file. T(erminal mode. Q(uit   R

  /* Transfer a file from your UNIX directory to your local Terak disk. */
   Enter Unix filename:  <unixfile>↵
   File name on this TERAK (ending .TEXT assumed):  <terakfile>↵
   Receiving...

   <unix.file> transferred to <terak.file>.TEXT
   Terminal: S(end file. R(eceive file. T(erminal mode. Q(uit /* Do whatever comes next. */
```

10.4 Compiling your Gcel program

Assuming you have written your Gcel program and it is stored in "exercise.g" in your current working directory on the UNIX Host, the procedure for compiling it is straightforward.

```
/* You must be logged into UNIX and in your working directory */
 % /usr/cat/bin/gcel exercise.g↵
 %
```

If there are no syntax errors in your Gcel source code, Gcel should exit normally and leave the object file, "exercise.t" in your current working directory.

This object file must be transferred to the Terak for subsequent assembly and inclusion in the final test program. The Terak system will expect a ".TEXT" suffix to whatever file name you use for this file on your UCSD Pascal working disk. This suffix is automatically appended by the "TERMINAL" utility, as described above.

10.5 Assembling your Gcel program

Assuming you have a compiled Gcel file, "EXERCISE.TEXT" on your working disk. You can assemble this to LSI-11 machine code, as follows:

```
/* The Terak must be booted with your working disk mounted. */
 Command:E(dit,R(un,F(ile,C(omp,L(ink,X(ecute,A(ssem, D(ebug,?[II.0]  A
 Assembling...
 Assemble what text?  EXERCISE↵
 To what codefile?  EXERCISE↵ /* May be another name. */
 11 Assembler II.0[d.4]
 Output file for assembled listing: (<CR> for none)↵

/* A bit map of the core image is displayed during assembly. */

...
/* Source code dependent messages here. */
...

 Assembly complete:    X lines /* X depends on your source code. */
    0 Errors flagged on this Assembly /* Hopefully, at least. */
 Command:E(dit,R(un,F(ile,C(omp,L(ink,X(ecute,A(ssem, D(ebug,?[II.0] /* At top of screen. */
```

If assembly is successful, the file "EXERCISE.CODE" will be left on your working disk. If there are errors flagged, or the assembler otherwise balks, the UCSD Pascal Reference Manual [3] should be consulted (see the last section of this guide).

10.6 Compiling your Driver program

Your Pascal driver program may be entered directly on the Terak, or may be edited and syntax checked by using the UNIX compiler on the UNIX Host. In the latter case, the Pascal source file must be transferred to the Terak for compilation to UCSD p-code. Assuming the driver source code is on your Terak working disk in the file "DRIVER.TEXT", the procedure for compiling it is:

```
/* The Terak must be booted with your working disk mounted. */
 Command:E(dit,R(un,F(ile,C(omp,L(ink,X(ecute,A(ssem, D(ebug,?[II.0]  C
Compiling...
Compile what text?  DRIVER↵
To what codefile?  DRIVER↵ /* May be another name. */

/* A bit map of the core image is displayed during compilation. */

PASCAL Compiler [II.0.A.1]
...
/* Source code dependent messages */
...
Command:E(dit,R(un,F(ile,C(omp,L(ink,X(ecute,A(ssem, D(ebug,?[II.0] /* At top of screen. */
```

If compilation is successful, the file "DRIVER.CODE" will be left on your working disk. If your code won't compile, the UCSD Pascal Reference Manual [3] should be consulted (see the last section of this guide).

10.7 Bringing it all together: linking the modules

When you have successfully gotten to this stage, you are almost done generating your test program. All that remains is to link together your two machine code modules. Assuming a compiled driver program, "DRIVER.CODE" and assembled Gcel program, "EXERCISE.CODE" are now on your Terak working disk, you can link them as follows:

```
/* The Terak must be booted with your working disk mounted. */
Command:E(dit,R(un,F(ile,C(omp,L(ink,X(ecute,A(ssem, D(ebug,?[II.0]  L
Linking... /* Seen only briefly; then */

/* a bit map of core is displayed. */

Linker [II.0 a.2]
Host file?  DRIVER↵
Opening DRIVER.CODE
Lib file?  EXERCISE↵
Opening EXERCISE.CODE
Lib file?↵
Map name?↵
Reading TESTNAME /* The Pascal source program name. */
Reading EXERCISE /* The Gcel procedure name. */
Output file?  TEST.CODE↵ /* May be another name. */
Linking TESTNAME #1
   Copying proc EXERCISE
Command:E(dit,R(un,F(ile,C(omp,L(ink,X(ecute,A(ssem, D(ebug,?[II.0] /* At top of screen. */
```

If linking is successful, the file "TEST.CODE" will be left on your working disk. If there are undefined globals or other errors, the UCSD Pascal Reference Manual [3] should be consulted (see the last section of this guide).

10.8 Running the test

The rest is up to you. Running the test itself is trivial. Writing a good one, and interpreting the results is not; however that's uniquely your problem. Assuming you have an executable test program, "TEST.CODE" on your working disk, and you have done the setup procedure given in section 7 of this guide, the test is run as follows:

```
/* The Terak must be booted with your working disk mounted. */
Command:E(dit.R(un.F(ile.C(omp.L(ink.X(ecute.A(ssem. D(ebug.?[II.0]  X
Execute what file?  TEST↵

/* Whatever happens now is what you programmed. */

/* Upon exit, (normal or abnormal) you will get back the UCSD prompt: */
Command:E(dit.R(un.F(ile.C(omp.L(ink.X(ecute.A(ssem. D(ebug.?[II.0] /* At top of screen. */
```

That's all, folks. If you've gotten this far, you are sufficiently well versed in the mechanics of using Gcel and the Terak Tester to do your tests. Don't be discouraged if your tests need a lot of debugging, not to mention your chip design. Such is a normal state of affairs, even for the experienced. Good luck!

11 References

[1] Kathleen Jensen and Niklaus Wirth, "PASCAL User Manual and Report", Second Edition, Springer-Verlag, New York, 1975.

[2] Wilson, I. R., and Addyman, A. M., "A Practical Introduction to Pascal", Springer-Verlag, New York, 1978.

[3] "UCSD Pascal Users Manual", Ed. by Shillington, K. A., Ackland, G. M., and Clark, R., Softech Microsystems, San Diego, 1980.

[4] "TERAK/UCSD p-SYSTEM VERSION II.0 OPERATING SYSTEM", document number 60-0111-001A, Terak Corporation, Scottsdale, AZ, 1982.

[5] "UNIX Programmer's Manual", Seventh Edition, Virtual VAX-11 Version, UCB, Berkeley, CA, 1981.

[6] Brian W. Kernighan and Dennis M. Ritchie, "The C Programming

[7] Gary Bishop, "Self Tracker: A Smart Optical Sensor on Silicon", PhD Thesis, Chapel Hill, NC, 1984.