

AD-A260 849



2

NAVAL POSTGRADUATE SCHOOL

Monterey, California



DTIC
ELECTE
FEB 23 1993
S c D

THESIS

A DIGITAL HARDWARE
TEST SYSTEM ANALYSIS
WITH
TEST VECTOR TRANSLATION
by

James T. Loeblein

December, 1992

Thesis Advisor:

Chin-Hwa Lee

Approved for public release; distribution is unlimited

93-03648



93

10/2/98

10 10

REPORT DOCUMENTATION PAGE												
1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED			1b. RESTRICTIVE MARKINGS									
2a. SECURITY CLASSIFICATION AUTHORITY			3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.									
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE												
4. PERFORMING ORGANIZATION REPORT NUMBER(S)			5. MONITORING ORGANIZATION REPORT NUMBER(S)									
6a. NAME OF PERFORMING ORGANIZATION Naval Postgraduate School		6b. OFFICE SYMBOL (If applicable) 55	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School									
6c. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000			7b. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000									
8a. NAME OF FUNDING/SPONSORING ORGANIZATION		8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER									
8c. ADDRESS (City, State, and ZIP Code)			10. SOURCE OF FUNDING NUMBERS									
			<table border="1"> <tr> <td>Program Element No</td> <td>Project No</td> <td>Task No</td> <td>Work Unit Accession Number</td> </tr> <tr> <td></td> <td></td> <td></td> <td></td> </tr> </table>		Program Element No	Project No	Task No	Work Unit Accession Number				
Program Element No	Project No	Task No	Work Unit Accession Number									
11. TITLE (Include Security Classification) A DIGITAL HARDWARE TEST SYSTEM ANALYSIS WITH TEST VECTOR TRANSLATION												
12. PERSONAL AUTHOR(S) James T. Loeblein												
13a. TYPE OF REPORT Master's Thesis		13b. TIME COVERED From To	14. DATE OF REPORT (year, month, day) December 1992	15. PAGE COUNT 109								
16. SUPPLEMENTARY NOTATION The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.												
17. COSATI CODES			18. SUBJECT TERMS (continue on reverse if necessary and identify by block number)									
FIELD	GROUP	SUBGROUP	Digital testing, simulation, lex, yacc, language translation, test vector									
19. ABSTRACT (continue on reverse if necessary and identify by block number)												
Digital logic testing occurs in two different test environments, digital simulation and actual hardware testing. A computer aided design (CAD) tool applies a set of stimulus/response test vector patterns to check the functionality of a digital circuit design. Once manufactured, the chip with this design is tested by a hardware tester system (i.e. automatic test equipment (ATE)). The ATE performs many tests in addition to the functionality test. However, the stimulus/response test vector formats used in these two environments are different and, therefore, incompatible in present form. This thesis is aimed at two major objectives. First, a system study will be performed on the GenRad-125 VLSI Hardware Tester System, including its usage, test capabilities and limitations. Secondly, this thesis addresses the problem of test vector format incompatibility between the two testing environments. Special UNIX tools, Lex & Yacc, are used to create a software translator which changes the CAD simulation file into the GenRad-125 Hardware Test System format.												
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS REPORT ☐ DTIC USERS			21. ABSTRACT SECURITY CLASSIFICATION Unclassified									
22a. NAME OF RESPONSIBLE INDIVIDUAL Chin-Hwa Lee			22b. TELEPHONE (Include Area code) (408) 646-2190	22c. OFFICE SYMBOL EC/Le								

Approved for public release; distribution is unlimited.

A Digital Hardware
Test System Analysis
With
Test Vector Translation

by

James T. Loeblein
Lieutenant, United States Navy
B.S., United States Naval Academy

Submitted in partial fulfillment
of the requirements for the degree of

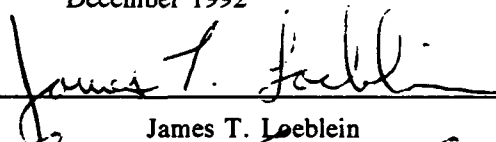
MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

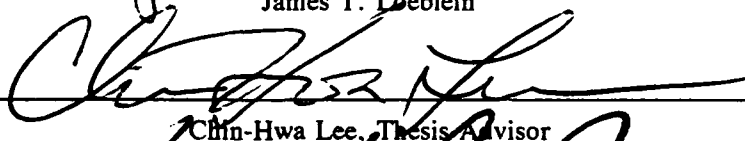
NAVAL POSTGRADUATE SCHOOL

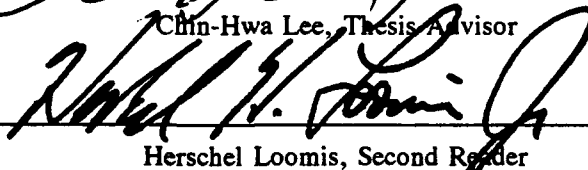
December 1992

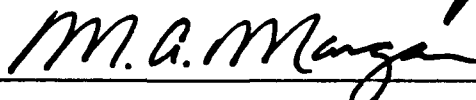
Author:


James T. Loeblein

Approved by:


Chin-Hwa Lee, Thesis Advisor


Herschel Loomis, Second Reader



Michael A. Morgan, Chairman
Department of Electrical and Computer Engineering

ABSTRACT

Digital logic testing occurs in two different test environments, digital simulation and actual hardware testing. A computer aided design (CAD) tool applies a set of stimulus/response test vector patterns to check the functionality of a digital circuit design. Once manufactured, the chip with this design is tested by a hardware tester system (i.e. automatic test equipment (ATE)). The ATE performs many tests in addition to the functionality test. However, the stimulus/response test vector formats used in these two environments are different and, therefore, incompatible in present form.

This thesis is aimed at two major objectives. First, a system study will be performed on the GenRad-125 VLSI Hardware Tester System, including its usage, test capabilities and limitations. Secondly, this thesis addresses the problem of test vector format incompatibility between the two testing environments. Special UNIX tools, Lex & Yacc, are used to create a software translator which changes the CAD simulation file into the GenRad-125 Hardware Test System format.

DTIC QUALITY INSPECTED 3

Accession For	
NTIS CRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution /	
Availability Codes	
Dist	Avail and/or Special
A-1	

TABLE OF CONTENTS

I.	INTRODUCTION	1
A.	DESIGN FOR TESTABILITY BACKGROUND	1
B.	DESIGN TESTING PROCESS	2
C.	THESIS OBJECTIVES	3
II.	HARDWARE TEST ENVIRONMENT	4
A.	SYSTEM OVERVIEW (GENRAD-125)	4
1.	General	5
2.	Rating Characteristics	5
3.	Basic System Structure	6
a.	Main Assemblies	7
b.	Peripheral I/O Devices	8
4.	System Software Description	9
a.	Operating System	10
b.	Utilities	10
B.	TEST PROGRAMMING AND EXECUTION METHODOLOGY	12
1.	Data Input Phase	13
a.	Test Pattern (.tpp) File	14
(1)	PINDEFS Section.	14
(2)	ADAPTOR Section	17
(3)	Test Pattern MODULE	17
b.	Parameter Specification File	20

(1) Programming Menu Screens.	20
(2) ASCII Text File Format	23
2. ASCII To Binary Translation Phase	24
a. Test Pattern Processor (TPP) Compiler .	24
b. ASCII to Par (ATP) Compiler	25
3. Test Execution Phase	26
a. Test Execution Menu Screens	27
(1) Debugging Control	29
(2) Execution Control	29
(3) Analysis Control	29
(4) Output Control	30
(5) Datalogging Control	30
b. Results Display Menu Screens	31
c. Input/Output Menu Screens	32
(1) File System Control	32
(2) Screens Configuration	33
4. Test Results Phase	33
a. Pass/Fail Results	34
b. Actual Measurement Results	34
c. Special Functions	35
(1) "Shmoo" Plots	36
(2) "Learn" Function	36
C. TESTING CAPABILITIES SUMMARY	37
1. Functional Tests	37
2. Power Supply Tests	38
3. DC Parametric Tests	38

a.	Input DC Parametric Tests	38
(1)	Iil Test	38
(2)	Iih Test	39
(3)	Vik Test	39
b.	Output DC Parametric Tests	39
(1)	Iol Test	39
(2)	Ioh Test	39
(3)	Vol Test	39
(4)	Voh Test	39
(5)	Iozl Test	39
(6)	Iozh Test	39
(7)	Ios Test	40
4.	AC Functional Tests	40
5.	Contact Tests	40
III.	CAD SIMULATION ENVIRONMENT	41
A.	SIMULATION OVERVIEW (MENTOR GRAPHICS)	41
1.	Schematic Capture	42
2.	Test Simulation	43
B.	SIMULATION OUTPUT FILE	45
1.	Structure	46
a.	Time Values	46
b.	Pin Labels	46
c.	Pin Values	46
2.	Design Example (74S181 ALU)	48
a.	Circuit Description	48

b. Input Stimulus	48
c. Output Simulation File	49
IV. SOFTWARE TRANSLATION METHODOLOGY	53
A. DISCUSSION	53
B. INTERPRETERS AND COMPILERS	56
C. UNIX TOOLS OVERVIEW	58
D. LEXICAL ANALYZER GENERATOR (LEX)	58
1. Background	58
2. Lex Specification Format	59
a. Rules Section	60
b. Definition Section	63
c. User Routine Section	64
3. Usage	64
E. YET ANOTHER COMPILER COMPILER (YACC)	65
1. Background	65
2. Yacc Specification Format	66
a. Declarations Section	68
b. Grammar Rules Section	70
(1) Symbol	70
(2) Definition	71
(3) Action	72
c. C Programs Section	73
3. Usage	73
4. Flow Control Summary	75

V.	TRANSLATOR DESIGN RESULTS	77
A.	OVERVIEW	77
B.	PROGRAM STRUCTURE	78
	1. Main Program (vector_map.c)	78
	2. Lex Routine (vector_map.l)	78
C.	PROGRAM USAGE	83
	1. Input File	83
	2. Command Line Entry	84
D.	RESULTS	84
VI.	CONCLUSIONS	88
A.	SUMMARY OF RESEARCH	88
B.	RECOMMENDATIONS FOR FURTHER RESEARCH	89
	LIST OF REFERENCES	93
	INITIAL DISTRIBUTION LIST	94

LIST OF TABLES

Table I	GR-125 RATING CHARACTERISTICS	6
Table II	GR-125 VALID TEST PATTERN ELEMENTS	18
Table III	GR-125 PROGRAMMING MENU SCREENS (CATEGORIES)	21
Table IV	DEVICE DESCRIPTIONS CATEGORY	22
Table V	ANALOG DATA SETS CATEGORY	23
Table VI	TIMING DATA SETS CATEGORY	24
Table VII	VECTOR TRUTH TABLE CATEGORY	25
Table VIII	TEST OPERATIONS CATEGORY	26
Table IX	TPP COMPILER (COMMAND LINE ENTRY)	26
Table X	ATP TRANSLATOR (COMMAND LINE ENTRY)	26
Table XI	TEST EXECUTION MENU SCREENS	28
Table XII	GR-125 RESULT DISPLAY MENU SCREENS	31
Table XIII	GR-125 INPUT/OUTPUT MENU SCREENS	32
Table XIV	PASS/FAIL MODE (REQUIRED SCREEN ENTRIES)	34
Table XV	ACTUAL MEASUREMENT (REQUIRED SCREEN ENTRIES)	35
Table XVI	"SHMOO" PLOT (REQUIRED SCREEN ENTRIES)	38
Table XVII	QUICKSIM SIGNAL VALUES	47
Table XVIII	QUICKSIM WRITE LIST ENTRY	47
Table XIX	74S181 PIN DESIGNATIONS [from Ref. 8]	50
Table XX	LEX REGULAR EXPRESSION EXAMPLE	63
Table XXI	LEX REGULAR EXPRESSION OPERATORS [from Ref. 9]	64

Table XXII	YACC DECLARATION ENTRY	69
Table XXIII	YACC DECLARATION SECTION KEYWORDS [from Ref.	
	9]	69
Table XXIV	YACC GRAMMAR RULE ELEMENTS	72
Table XXV	"vector_map.1" COMPILATION STEPS	80
Table XXVI	"vector_map" COMPILATION STEPS	83
Table XXVII	"vector_map" COMMAND LINE ENTRY	84
Table XXVIII	INCLUDE STATEMENT FOR GR-125 .tpp FILE .	85

LIST OF FIGURES

Figure 1	GR-125 Overall Structure Layout [from Ref.3]	7
Figure 2	GR-125 Main Assembly Structure [from Ref. 3]	8
Figure 3	UniPlus+ Operating System [from Ref. 3] . .	10
Figure 4	UniPlus+ Utilities [from Ref. 3]	11
Figure 5	GR-125 Testing Procedure Phases	13
Figure 6	GR-125 Testing Methodology	15
Figure 7	74S04 Hex Inverter .tpp File	16
Figure 8	GR-125 Tester Channel Assignment[from Ref.5]	17
Figure 9	GR-125 Sample Test Vector Pattern	19
Figure 10	GR-125 "Shmoo" Plot Example [from Ref. 4] .	36
Figure 11	CAD Design/Simulation Process	42
Figure 12	QuickSim Input/Output Files [from Ref.7] . .	44
Figure 13	QuickSim List Window Display [from Ref. 7] .	45
Figure 14	74S181 ALU Connection Diagram	49
Figure 15	74S181.misl Stimulus File (partial)	51
Figure 16	74S181.list Sim_output File (partial) . . .	52
Figure 17	Test Vector Translation Procedure	55
Figure 18	Compiler Processing Stages	57
Figure 19	UNIX Toolkit Hierarchy	59
Figure 20	Full Lex Specification Format	60
Figure 21	Lex Specification Rule	61
Figure 22	Lex Usage Steps	66
Figure 23	Parsing Description [from Ref. 9]	67

Figure 24	Full Yacc Specification Format	68
Figure 25	Yacc Grammar Rule Format	70
Figure 26	Yacc Usage Steps	73
Figure 27	Lex And Yacc Usage Summary [from Ref. 9] . .	74
Figure 28	Lex And Yacc Flow Control [from Ref. 9] . .	75
Figure 29	"vector_map.c"	79
Figure 30	"vector_map.l" Code	81
Figure 31	"74S181.v_out" File	87
Figure 32	Translation Summary Without WAVES	91
Figure 33	Translation Summary With WAVES	92

ACKNOWLEDGEMENTS

A special note of thanks goes to the Naval Maritime Intelligence Center whose sponsorship made the GenRad-125 Digital Tester available for this research. Additionally, without Mrs. Janet Hooper's material and administrative support surrounding this GenRad-125 Tester this thesis could not have been possible.

I also wish to recognize John Sweeney, John Groat, and Damon Baker from the Nuclear Effects Directorate at the White Sands Missile Range. Their patience and knowledge provided tremendous insight into the operation and capabilities of the GenRad-125 Tester.

The guidance, direction, and constant encouragement provided by Dr. Chin-Hwa Lee, my thesis advisor, was invaluable to the completion of this thesis. Furthermore, Dr. Herschel Loomis' constructive review greatly enhanced its readability.

Finally, I wish to thank my wife Carol for her love and support during the entire thesis process. She helped keep a smile on my face.

I. INTRODUCTION

A. DESIGN FOR TESTABILITY BACKGROUND

Electronic circuit testing has become an extremely crucial step in SSI/LSI/VLSI digital circuit design and manufacturing. In the past, digital component testing was considered at best a "post-design" activity [Ref. 1]. Digital testing seemed to occur last in the R&D, design, prototype, and production sequence. However, manufacturing industries of today are discovering that the high costs associated with testing amount up to 60% of the total production costs [Ref. 2:p. v]. Furthermore, recent increases in digital design complexity give rise to a situation where a circuit designer can produce a digital circuit which is virtually un-testable completely. Therefore, the only way to reduce this cost is to incorporate test activities into the design process, hence, creating a "testable design" [Ref. 2].

In order to pursue a testable design it is necessary to define the term "circuit testability".

A circuit is testable if a set of test patterns can be generated, evaluated and applied in such a way as to satisfy pre-defined levels of performance, defined in terms of fault-detection, fault-location and test-application criteria, within a pre-defined cost budget and time scale [Ref. 2:p. ix].

B. DESIGN TESTING PROCESS

Modern digital circuit testing occurs within two design environments, simulation tests and actual hardware testing. A Computer Aided Design (CAD) tool with an interactive logic simulator tests the functionality of a digital circuit design. This CAD logic simulator allows a specific design test cycle: stimulus application, simulation, results analysis, and design modification. This thesis will utilize the Mentor Graphics Quicksim CAD tool for an actual design conducted within the computer simulation environment.

Actual hardware testing using Automatic Test Equipments (ATE), such as the GenRad GR125 VLSI Tester, compose the second test environment. Once a digital chip is manufactured, a series of testing is performed. In addition to logic functionality, modern ATEs also perform D.C. Parametric, A.C. Parametric, Functional and Power Supply tests. This thesis will analyze the capability of the GenRad GR125 VLSI Test System and examine the testing cycle within the integrated hardware testing environment.

As described above, a chip design will be tested in both environments. Testing for functionality allows the digital chip designer to determine if his design responds correctly to a given input stimuli (i.e. does the chip logic function work as expected). To test this aspect of the design, a set of stimuli and expected response patterns are applied to the chip's input and output pins. This set of input stimuli and

expected response patterns are known as test vector patterns. However, both the Mentor Graphics Quicksim, and the ATE, GenRad GR125 VLSI Test System are stand-alone systems. The test vector pattern syntaxes used in each environment are not compatible. As a result, presently, test vector patterns for two separate formats must be generated.

C. THESIS OBJECTIVES

This thesis has two major objectives achieved within the hardware and software design and test environments. First, a thorough study was performed on the GenRad GR125 VLSI hardware test system, which reveals its usage, test capabilities and limitations. Secondly, this thesis provides a solution to the problem of test vector pattern incompatibility between the simulation and tester environments. Special UNIX tools, Lex and Yacc, are used to create a software translation program to bridge this incompatibility gap. This translation program provides an interface between the test vector patterns generated from the Mentor Graphics Quicksim simulator and the required format for the GenRad-125 VLSI hardware tester system.

II. HARDWARE TEST ENVIRONMENT

Automatic Test Equipment (ATE) provides the capability to thoroughly test a digital logic chip in a power on situation. There are many modern ATE's similar in functionality. However, this thesis will be focused on one specific ATE: the GenRad GR125 VLSI Test System (GR-125). In order to reveal the complete hardware test system, three major areas will be discussed. The first area provides a comprehensive overview of the GR-125 focusing on its characteristics, main component layout and system software implementation. Secondly, the overall programming and execution methodology for component testing will be analyzed. This methodology will be described in four major phases (Data Input, Translation, Execution and Results). The discussion concerning the software interface to the GR-125 will lead to discussion in chapters III and IV of this thesis. Finally, the third major area of discussion identifies some special testing capabilities of the GR-125.

A. SYSTEM OVERVIEW (GENRAD-125)

The GenRad GR-125 tester provides a broad range of digital logic testing capability. However, in order to effectively utilize this capability a basic understanding of GR-125 characteristics, system structure and system software is required. Therefore, the purpose of this system overview is

to provide a logical, comprehensive and user-friendly documentation for the GR-125 tester operation. The approach taken here will focus on the user's perspective instead of technical manual details.

1. General

The GR-125 is classified as a low voltage digital logic tester. Although originally designed for high quantity output production testing, the GR-125 provides an excellent research testing platform for diagnostic analysis of individual chips. As the name implies, the GenRad GR125 VLSI test system has the flexibility to accommodate a wide range of chip component complexity. The entire spectrum of complexity from Small Scale Integration (SSI) to Very Large Scale Integration (VLSI) are accommodated by the GR-125. The complexity of the digital component under test is limited only by its maximum number of pins.

2. Rating Characteristics

The GR-125 has the capability to test any digital device up to a maximum of 64 pins. As discussed above, these pins consist of low voltage only (|0-8| volts). Of the total pin count, half the pins can function as drive elements and half the pins can function as sense elements. Drive pins are used to put a desired digital stimulus signal on a pin. Sense pins, however, use comparators to compare the actual pin condition signal with the expected pin condition values. The

timing signals for chip testing are generated by a 12.5 Mhz clock. Memory capacity of the GR-125 limits each test pin to 64 Kbytes of test vectors. Table I provides a summary of these general characteristics for the GR-125.

Table I GR-125 RATING CHARACTERISTICS

... 64 pins low voltage
 (0-8 volts)

... 12.5 Mhz clock speed

... 64 Kbytes of test vector memory
 per pin

... 32 Drive pins

... 32 Sense pins

3. Basic System Structure

The GR-125 test system consists of two subsystems: main assemblies and peripheral Input/Output (I/O) devices. Refer to Figure 1 for an overall structure layout.

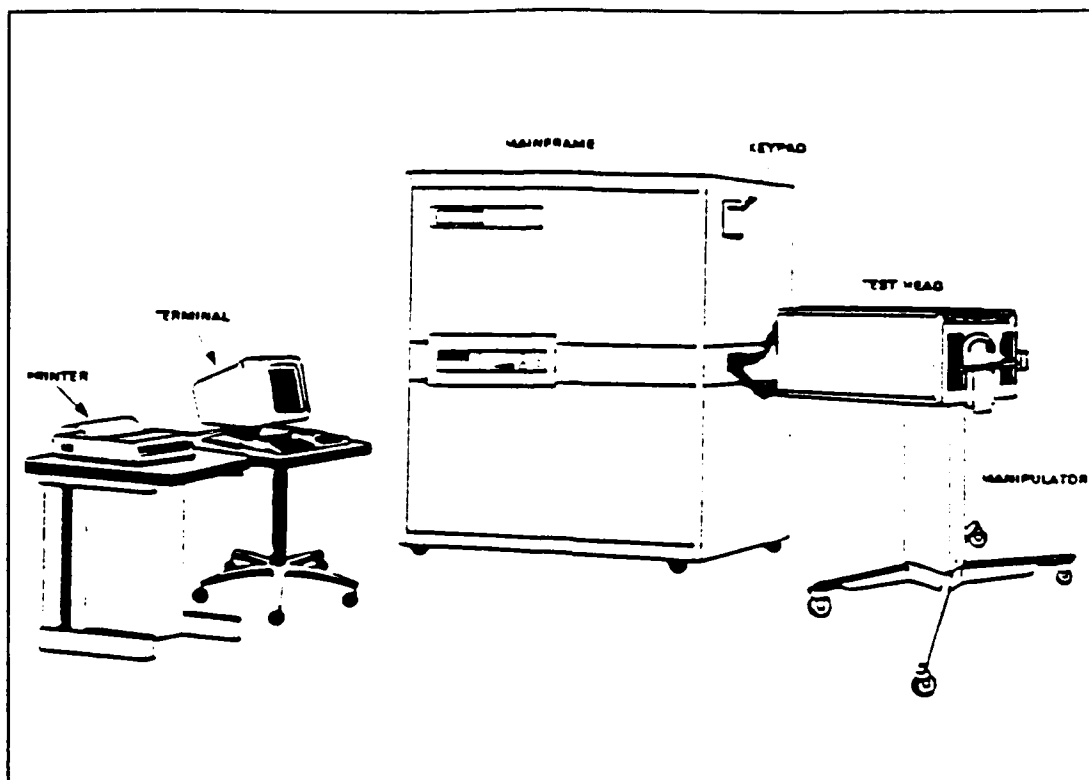


Figure 1 GR-125 Overall Structure Layout [from Ref. 3]

a. Main Assemblies

There are two main subsystems which make up the GR-125 test system: Mainframe and Test Head. The Mainframe functions as the central control unit which houses the power supply, computer, and test signal generator of the tester. The Test Head houses various interface adaptor boards which connect to the Device Under Test (DUT) via plug-in connectors. Circuits located within the Test Head provide an interface for the test signals between the main frame and DUT. Three subassemblies are contained in the Mainframe: the System Power Supply Bay, CPU Card Cage, and Test Electronics Card Cage. Refer to figure 2.

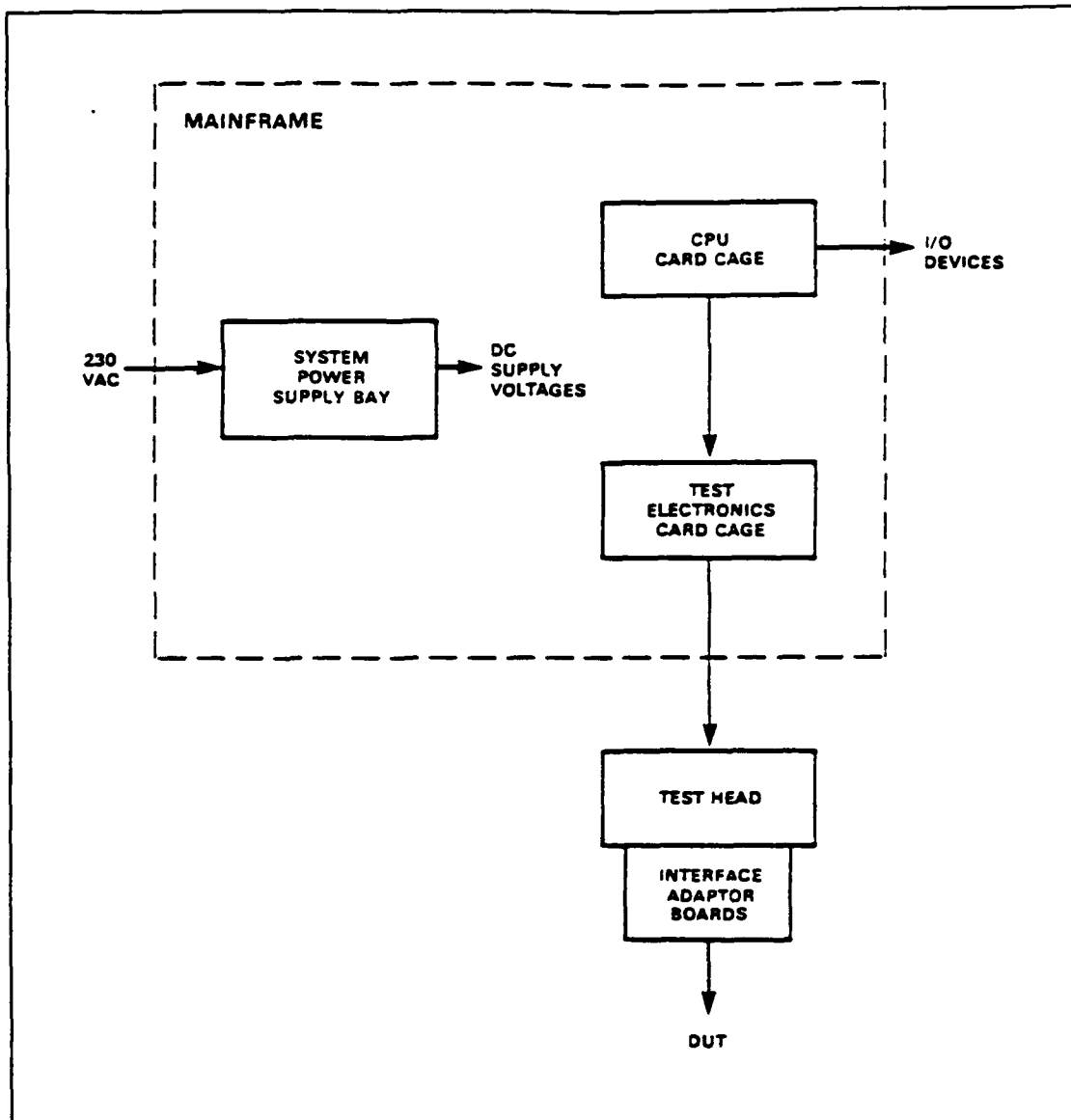


Figure 2 GR-125 Main Assembly Structure [from Ref. 3]

b. Peripheral I/O Devices

Various peripheral I/O devices are also incorporated within the GR-125 test system. The VT-220 Video Display Terminal presents menu formatted user screens to set up detailed test requirements, to observe test results and to

interact with the CPU via a UNIX based software. In its present set up conditions, the Printer works in a screen dump mode allowing the Print Screen command only. Although restrictive, this set up is adequate for obtaining a hard copy test result. The Keypad (refer to figure 1) was designed to be used when performing routine testing under high production, high volume test conditions. Because of the low volume and research orientation of this thesis, the operation of the keypad will not be addressed. On the front of the Mainframe is a control panel which holds the main power switches. A magnetic disk and tape unit is located next to these switches. This disk and tape I/O device provides an electronic copy capability for program and data storage as well as system backups. Because an Ethernet card is not available, the present GR-125 hardware tester is a stand alone system.

4. System Software Description

The heartbeat of the GR-125's operation is its system software. The GR-125 test system utilizes UNIX and custom software packages to form the backbone operating system for the GR-125 tester. Because of its wide acceptance in industrial and engineering applications, UNIX software provides an interactive and general purpose operating system. The version of UNIX software installed in the GR-125 is the UniPlus+ software (v2.0), which runs on the Motorola 68000 CPU

chip. This UniPlus+ software resides in a 15 Mbyte space on the 85 Mbyte hard disk.

a. Operating System

The UniPlus+ consists of both an operating system and its utilities. The custom software works as a link between the user and the GR-125 test system controlling several I/O functions. Refer to figure 3. This custom software permits the extensive use of interactive user menu screens. These screens, which will be covered later in detail, allow an operator to set up the GR-125 to test a device as well as developing new test programs (on-line or off-line).

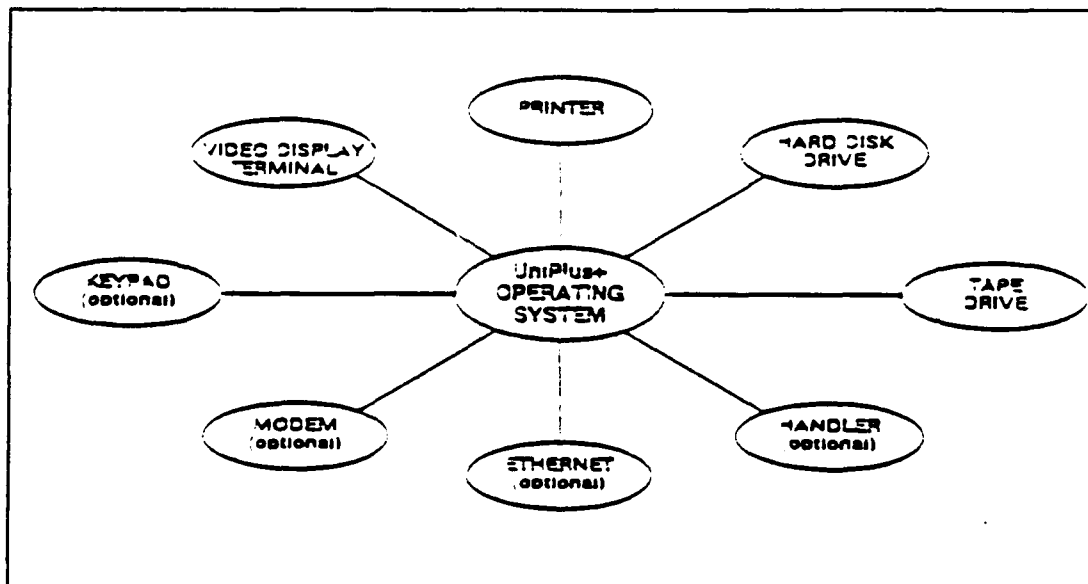


Figure 3 UniPlus+ Operating System [from Ref. 3]

b. Utilities

In addition to its operating system, the UNIX software package also contains many help utilities. See

Figure 4. These utilities enable a user to develop new test programs and make modifications to existing programs. The file system consists of both ASCII text and binary data files stored on the hard disk. The main purpose of the file system is to act as a storage mechanism for test programs and system configuration data associated with particular test programs.

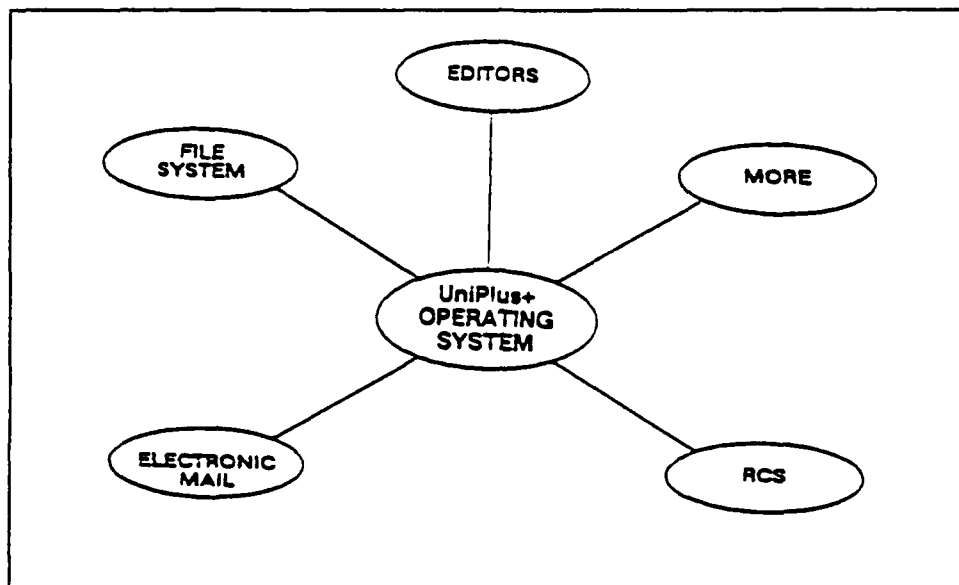


Figure 4 UniPlus+ Utilities [from Ref. 3]

The UNIX software package also supports three different editors: ACE, vi, and uEMACS. The More command allows a text file to be read one screen at a time. As a stand alone system, this particular GR-125 setup does not support Electronic Mail. The RCS (Revision Control System) makes up the last of the major UNIX utilities supported by the GR-125 test system. The RCS manages software libraries. It stores and retrieves multiple revisions of program and test files. Furthermore, the RCS maintains a complete history of

changes between test program versions so that one can easily find the changes made between different versions. For more details on RCS refer to Ref. B.

B. TEST PROGRAMMING AND EXECUTION METHODOLOGY

Test programming and execution methodology describes the overall GR-125 testing procedure from data input to observable test results. This section will present the four main phases of the GR-125 procedure: Data Input, Translation, Execution, Results. Refer to Figure 5. The first phase of the testing procedure includes a software interface to the GR-125 tester as illustrated in Figure 6. Test pattern information and parameter specification data are input during this phase. Next, during the translation process, ASCII formatted data in the software interface is translated to binary code for use by the GR-125 tester. Once the required binary files are obtained, the GR-125 can support a variety of test executions. By manipulating different support screens within the test execution phase, the user can produce several desired output formats. The Results phase produces several categories of results depending on the option chosen during the Execution phase. In the following discussion, the purpose is to describe this testing methodology in a manner which provides the most benefit to the GR-125 user.

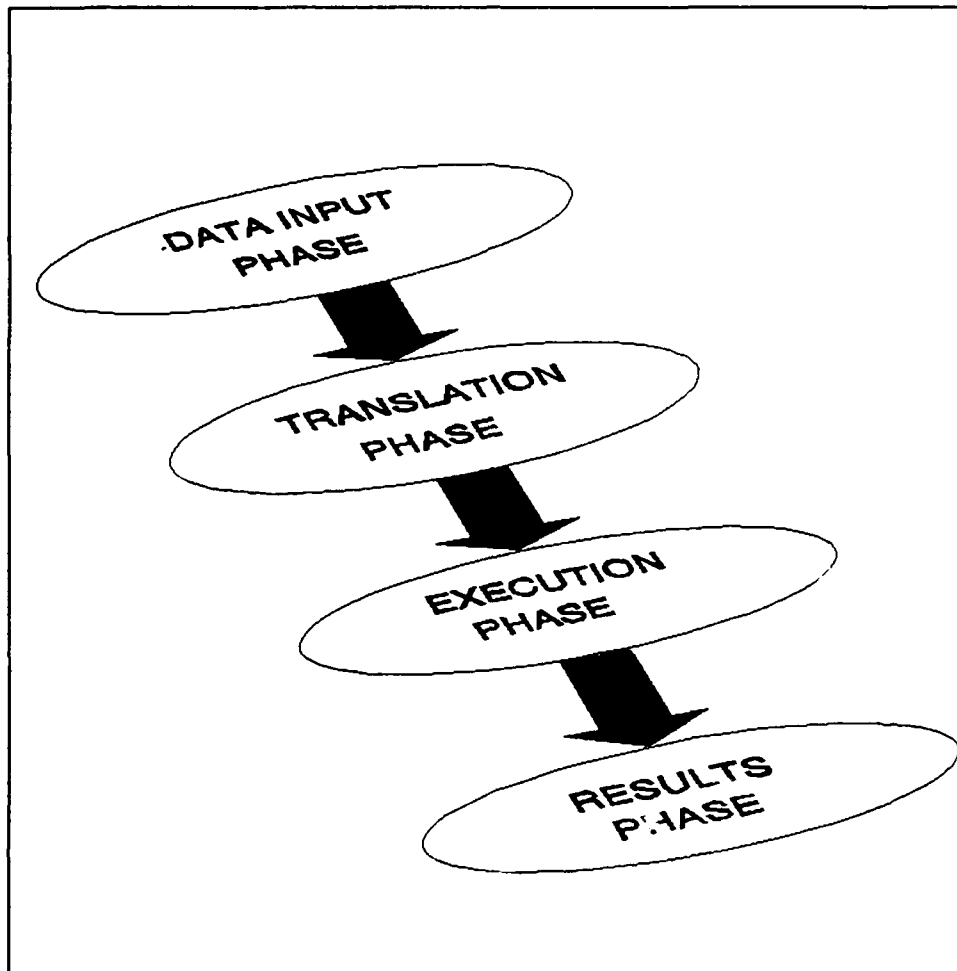


Figure 5 GR-125 Testing Procedure Phases

1. Data Input Phase

Two software files are used to enter the Device Under Test (DUT) specification to the GR-125 tester. The test pattern processor (.tpp) file defines the actual pin mapping and test vector configuration of the GR-125. Secondly, the parameter specification file contains the bulk of the programming information necessary to perform a successful

test. Data is placed into this file through the use of custom Programming Screens. These screens prompt the user for a wide range of component parameters from pin current levels to timing specifications.

a. Test Pattern (.tpp) File

The first step in the development of a GR-125 test program is the generation of a .tpp file. This file contains the tester pins to device pins mapping information. Additionally, the actual test vector patterns which are applied to the DUT are also incorporated into this file. It is important to note that the .tpp file is an ASCII text file created by any generic text editor. Figure 7 illustrates a basic .tpp file for a 7404 Hex Invertor.

The .tpp file is made up of three sections: PINDEFS, ADAPTOR, and MODULE. The PINDEFS and MODULE sections are mandatory. The ADAPTOR section contains the only optional entries. [Ref. 5:p. 1-5]

(1) *PINDEFS Section.* The PINDEFS section is composed of columns of information to inform the GR-125 tester of the physical (adaptor) connections between the GR-125's tester channels and the DUT. This section contains the tester pin mapping information. Figure 8 illustrates a typical pin assignment for a 74F374 Octal D-Type Flipflop. In this example, the 74F374's pin #1 (output enable line "/oe") is connected to the GR-125 tester channel #3. Note, however,

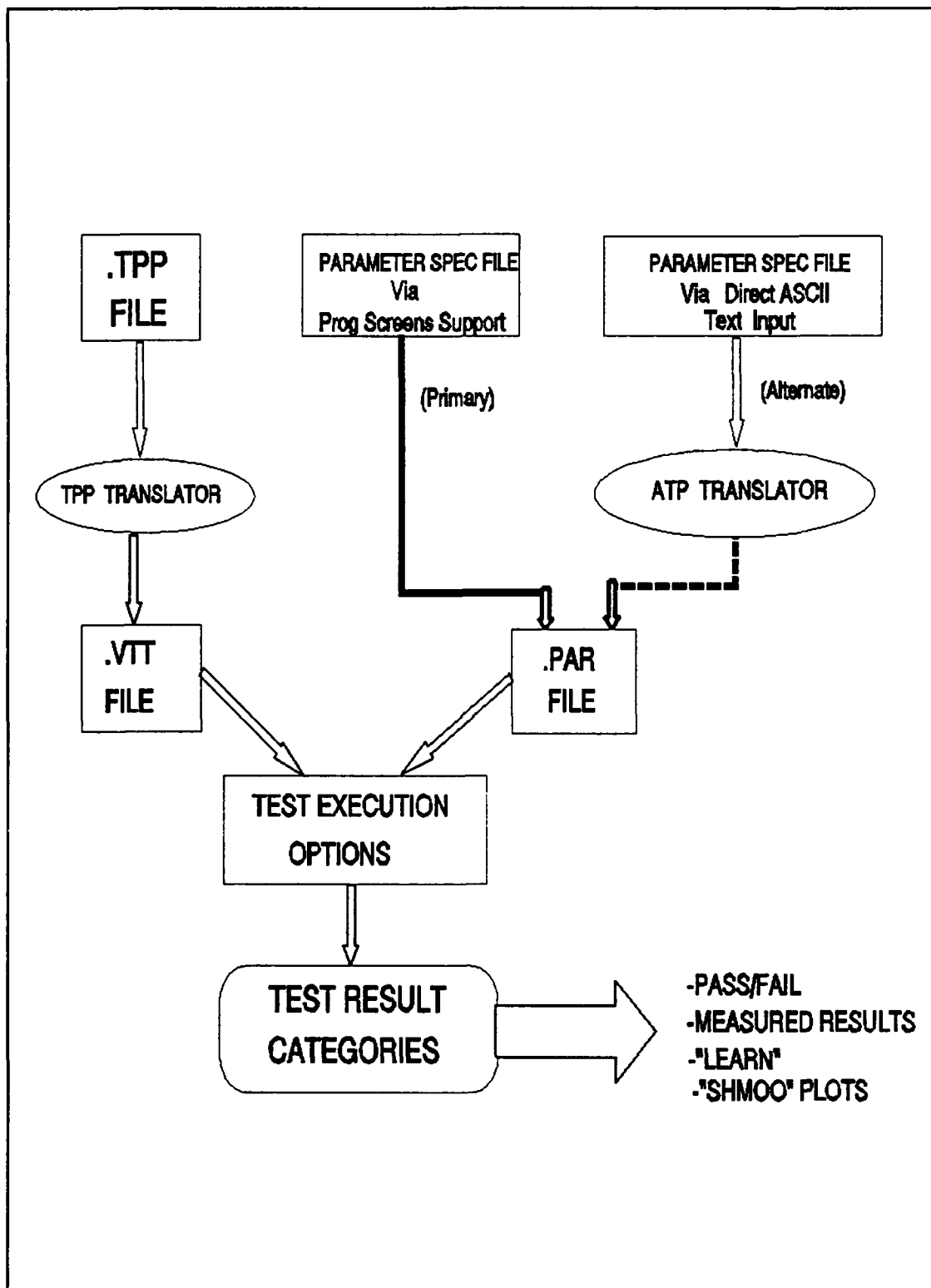


Figure 6 GR-125 Testing Methodology

```

{7404 HEX INVERTERS}

ADAPTOR UNIDAB/24
        FABID = 1;
        FABREV = 0;
        DABID = 0;
        DABREV = 0;

END;

PINDEFS
{
    pinname      mode      column      dutpin      mapping      testerpin  }
    in1a         1         1         1         >         6;
    in2a         2         3         3         >         8;
    in3a         3         5         5         >        10;
    in4a         4         9         9         >        14;
    in5a         5        11        11         >        16;
    in6a         6        13        13         >        18;

    out1y        7         2         2         >         7;
    out2y        8         4         4         >         9;
    out3y        9         6         6         >        11;
    out4y       10        8         8         >        13;
    out5y       11       10       10         >        15;
    out6y       12       12       12         >        17;

    Ground       GND       7;
    Vcc          PWR      14;

END;

MODULE 7404:
PATTERN

{Functional & Parametric Test Patterns}

{VIX Test}
/...../ T01
/000000 ...../ T01 STOP

{ICCH Test}
/000000 HHHHHH/ T01
/000000 ...../ T01 STOP

{ICCL Test}
/111111 LLLLLL/ T01
/111111 ...../ T01 STOP

{Simple Functional Test}
/000000 HHHHHH/ T01
/010101 HLHLHL/ T01
/101010 LHLHLH/ T01
/111111 LLLLLL/ T01
/100000 LHHHHH/ T01
/010000 HLHHHH/ T01
...
/111110 LLLLLH/ T01 STOP

END;

```

Figure 7 74S04 Hex Inverter .tpp File

that the 74F374's power and ground pins are not connected to GR-125 tester channels. Ground and power pins (pins #10 and #20 on the device socket) are already internally wired to the adaptor. [Ref. 5:p. 4-5]

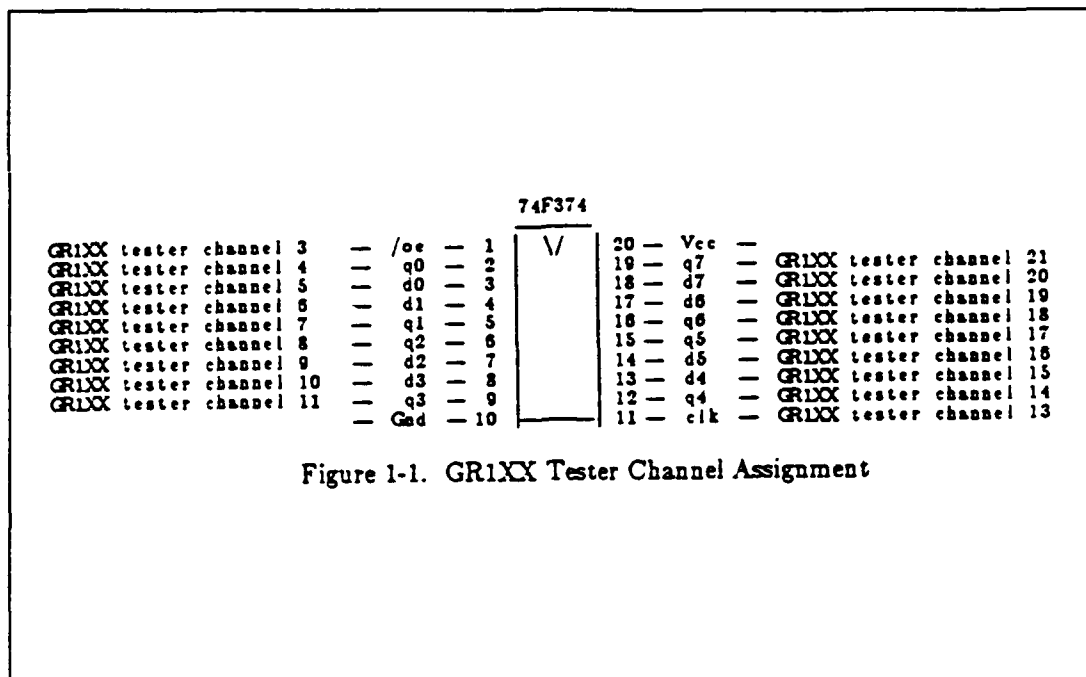


Figure 1-1. GR1XX Tester Channel Assignment

Figure 8 GR-125 Tester Channel Assignment [from Ref. 5]

(2) *ADAPTOR Section.* The ADAPTOR section is the only optional section within the .tpp file. This section allows the GR-125 user to specify the Device Adaptor Board (DAB) and Tester Adaptor Board (TAB) to be used with the test program. [Ref. 5:p. 4-12]

(3) *Test Pattern MODULE.* This section of the .tpp file contains the actual test vector pattern elements. These elements define whether a pin functions as a driver for stimulus or a comparator (monitor) for response. A test

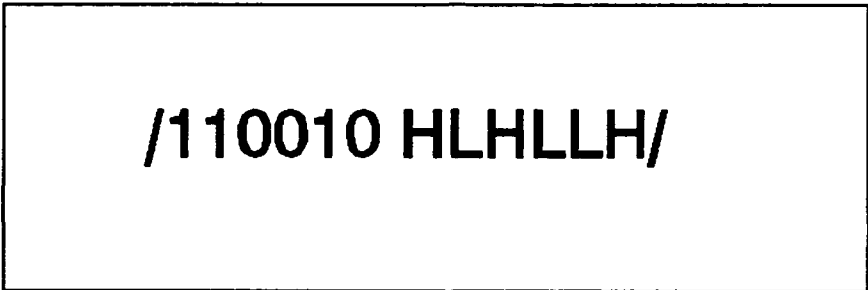
pattern MODULE section is mandatory. The drive and compare state information (used by the tester to sequence the DUT through the functional table) is placed here using the values specified in the manufacture's data book. Note that these pattern elements are then stored in memory behind each tester pin. Valid test pattern elements are listed in Table II. "Drive" indicates the stimulus applied to the input pins by the GR-125. "Monitor" indicates the sensing condition of the output pins.

Table II GR-125 VALID TEST PATTERN ELEMENTS

Pattern Vector Element	Explanation
0	Drive low, neglect response
1	Drive high, neglect response
X (or .)	Driver off, neglect response
L	Driver off, monitor low
H	Driver off, monitor high
T	Driver off, monitor tri-state
N	Drive low, monitor low
Y	Drive high, monitor high
Z	Drive low, monitor high
U	Drive high, monitor low
K	Klunk (CLOCKMODE only)
C	Clock (CLOCKMODE only)
i	1 in Alternate format (CLOCKMODE only)
o	0 in Alternate format (CLOCKMODE only)
?	Repeat previous state
- (dash)	Hold pin state

(a) Format. Three field entries are required to make a valid test pattern: MODULE, PATTERN, END. The MODULE input provides for a name field associated with this

section of the .tpp file. The PATTERN entry stores the actual test vector patterns. A test vector pattern has a pin control field which contains a pattern vector element for each device pin. Proper syntax requires test vector patterns to be buffered by slashes "/". Refer to Figure 9. Note, the number of elements must be equal to the number of columns listed in the PINDEFS section of the .tpp file. Embedded spacing is ignored by this syntax. Ref 5 discusses four optional fields which can also be added to these pattern vector elements. [Ref. 5:p. 4-14]



/110010 HLHLLH/

Figure 9 GR-125 Sample Test Vector Pattern

(b) Include statement. A special use of an INCLUDE statement permits the user to input a large test vector pattern code without having to retype each pattern element into the .tpp ASCII text file. The INCLUDE statement actually helps to modularize test vector patterns. This statement will be used to input a test vector file translated from a CAD simulation file. Chapter IV of this thesis will concentrate on this translation process.

b. Parameter Specification File

The parameter specification file is also composed of data taken from a chip manufacturer's data handbook. This data is then used to test the DC parametrics and AC timing of a DUT. Data input into this file can occur via two methods. The first method utilizes custom Programming Screens which prompt the user for data input. The second method for inputting data into the parameter specification file does not use menu driven screens support. This method uses a simple ASCII text file format generated by the user from any text editor. Note, however, that the input data is in the same format for both methods.

(1) *Programming Menu Screens.* Programming Menu Screens are provided by the GR-125 which allow the user to enter data for a particular device of interest. These screens work in coordination with the test pattern vectors listed in the .tpp file. The Programming Menu Screens are subdivided into five categories. Refer to Table III. This section will discuss the primary purpose of each of these five categories. Furthermore, every screen contained in these major categories will be listed for easy reference. Refer to Ref 5 for a detailed description of each individual screen.

**Table III GR-125 PROGRAMMING MENU SCREENS
(CATEGORIES)**

Category	Function Key
DEVICE DESCRIPTIONS	"F6"
ANALOG DATA SETS	"F7"
TIMING DATA SETS	"F8"
VECTOR TRUTH TABLE	"F10"
TEST OPERATIONS	"F9"

(a) Device Descriptions. The Device Descriptions screens make up the bulk of the DUT parameter specification file. The screens listed here describe the device and test plan, including such parameters as package size, device technology, pin types, pin names, pin condition sets, pin test sets, etc. Table IV lists the screens located in this category. [Ref. 5:p. 2-4]

(b) Analog Data Sets. The Analog Data Sets screens category defines the force and measurement levels used in functional and parametric testing. Supply voltage and current limits are also specified in these screens. Finally, all of the drive and compare pin levels for the functional tests are input during this set of screens. Table V lists the screens located in the category. [Ref. 5:p. 2-23]

(c) Timing Data Sets. The Timing Data Sets screens category provides a wide range of timing information. Period and edge times defining the test program vectors are

Table IV **DEVICE DESCRIPTIONS CATEGORY**

DEVICE DESCRIPTIONS [F6]



- **Device Describe**
 - **Device Adaptor Pin Mapping**
 - **Pin Condition Set**
 - **Pin Test Set**
-

entered here. The resolution of the timing edge is determined by the largest period used for the test. Additionally, this category of screens provides a mechanism for assigning the formats and edge selections used for each pin condition set. Finally, several screens in this category produce a pictorial and/or tabular representation of the timing relationships of various input waveforms. Table VI lists the screens found in this category. [Ref. 5:p. 2-33]

(d) Vector Truth Table. The Vector Truth Table screens are used to edit data located in the test vectors memory. Changes can be temporary or made permanent by editing the source file. This set of screens is used in conjunction with a special GR-125 diagnostic tool, "Learn". The function of this special output is covered later in this chapter. Table VII lists the screens found in this category. [Ref. 5:p. 2-47]

Table V ANALOG DATA SETS CATEGORY

ANALOG DATA SETS [F7]



- **Default Pin Levels**
 - **Pin Levels Set**
 - **Power Supply Levels Set**
 - **Load Relay Set**
-

(e) Test Operations. The Test Operations screens provide the user with the capability to describe how a particular test is to be performed. Each type of test represents a particular AC or DC parameter that the GR-125 test system can measure. In summary, the screens in this category allow the user to modify various parameters associated with the test categories defined by the GR-125. These test categories will be addressed in the Test Execution phase discussion later in this chapter. Table VIII lists the screens found in this category. [Ref. 5:p. 2-53]

(2) *ASCII Text File Format*. An alternate method of inputting information into the parameter specification file is through a direct ASCII text file as shown in Figure 6. This method may prove to be a more streamlined-line approach by bypassing the Programming Screens menus; however, the required data input is the same for both processes. Appendix A of

Table VI TIMING DATA SETS CATEGORY

TIMING DATA SETS [F8]



- Timing Array Set
 - Edge & Format Set
 - Timing Waveforms
 - Timing Data Table
 - Timing Data Sets(Combined)
-

(Ref. 5) presents detailed information on the syntax required for this method of data input.

2. ASCII To Binary Translation Phase

Recall that the two major files produced during the Data Input Phase of the GR-125 test procedure were in an ASCII text format. However, the GR-125 requires that these ASCII input files be translated into machine-coded binary files for proper tester implementation. This translation process is accomplished through the use of two separate compilers within the GR-125 test system.

a. Test Pattern Processor (TPP) Compiler

The TPP compiler translates the .tpp ASCII text file into a machine-coded binary Vector Truth Table (.vtt) file. The .vtt file is actually used to perform the machine-coded functions specified in the .tpp file. The user must compile the .tpp file prior to testing. Table IX illustrates

the command line entry required to compile the .tpp file.

[Ref. 5]

b. ASCII to Par (ATP) Compiler

The ATP compiler translates the parameter specification file (ASCII text file format) into a machine-coded binary parameter specification (.par) file. Table X illustrates the command line entry required to compile the ASCII formatted parameter specification input file. Note, however, that the parameter specification input file produced through the Programming Menu Screens does not require a separate compilation process. Refer to Figure 6. Once data is entered into an individual Programming Menu Screen, the GR-125 internally compiles it into a binary .par file.

Table VII VECTOR TRUTH TABLE CATEGORY

VECTOR TRUTH TABLE [F10]



- ☐ Truth Table Edit
- ☐ Truth Table Column Mapping

Table VIII TEST OPERATIONS CATEGORY

TEST OPERATIONS [F9]



- Test Operation
- Vector Set
- Bin Mapping & Control
- Binning Sequence

Table IX TPP COMPILER (COMMAND LINE ENTRY)

prompt> **tpp filename_one.tpp**

Table X ATP TRANSLATOR (COMMAND LINE ENTRY)

prompt> **atp filename_two.atp**

3. Test Execution Phase

Test execution in the GR-125 is designed and initiated by the system user through the "system test menu screens". These screens are actually a series of menus similar to the

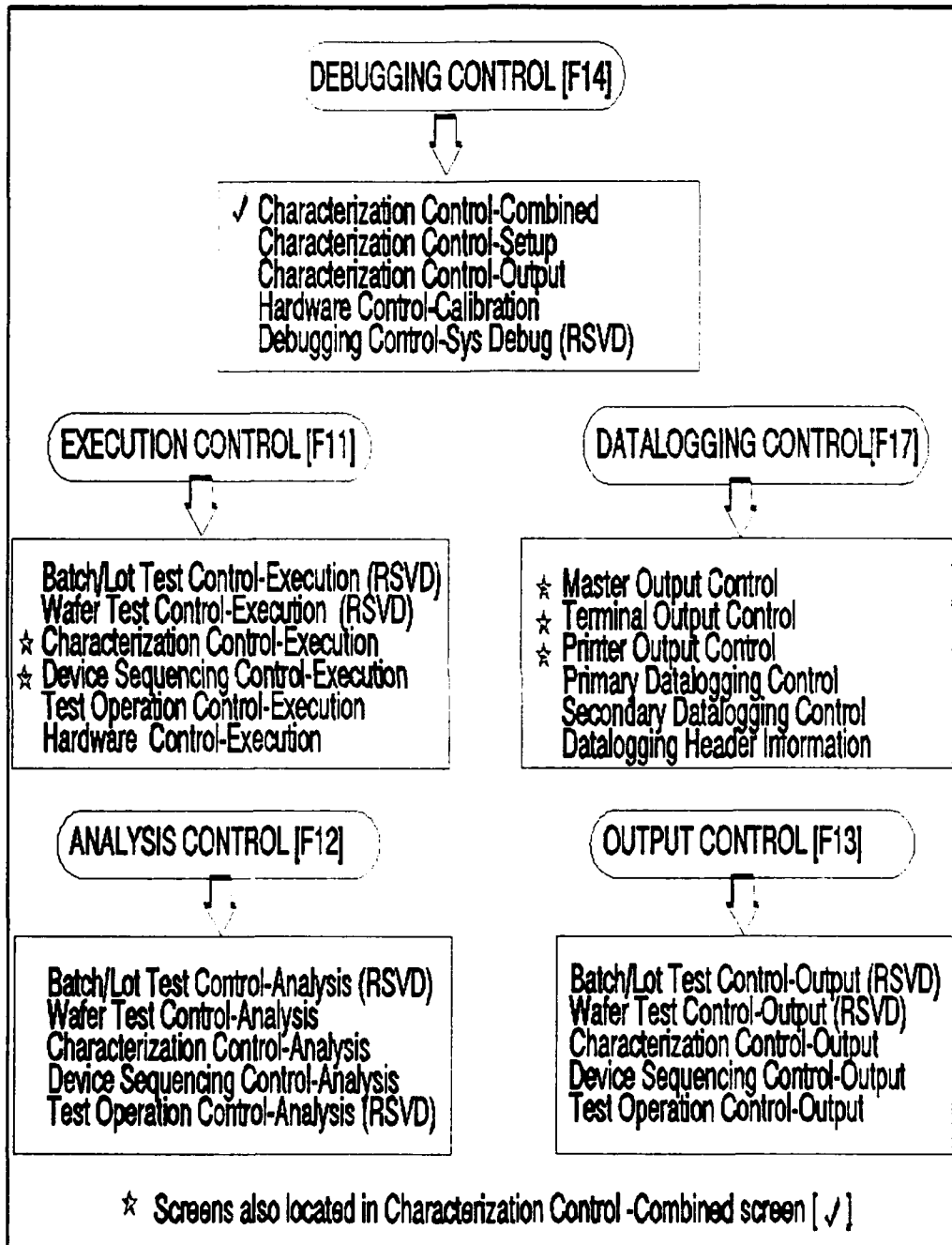
Programming Menu Screens used to create the .par file. These system test menu screens provide a convenient method for configuring the GR-125 tester for a test operation. This configuration determines which part of a test program will be executed as well as establishing which type of format the output produces. Furthermore, this series of menus guides the user through a decision-making and data selection process. Modifications to these screens can be temporary for current testing or made permanent by overwriting the program or creating a copy.

The system test menu screens described above can be organized into three broad categories: Test Execution, Results Display, and Input/Output. The following paragraphs will outline the actual screens in each of these categories. Due to the emphasis on the overall testing procedure, this section will not discuss every detail of each individual screen. However, specific references to the applicable technical manuals will be made.

a. Test Execution Menu Screens

These screens provide the majority of control over the Test Execution Phase. Manipulation of these test execution menu screens actually determines the output format of test results. The usage of these screens can be defined in four categories. Refer to Table XI. Certain individual screens denoted by "(RSVD)" are for GENRAD usage only.

Table XI TEST EXECUTION MENU SCREENS



(1) *Debugging Control.* The screens in this category are used to set up the GR-125 tester to perform a one dimensional (1D) or two dimensional (2D) plot of various current, voltage or timing parameters. These engineering characterization (i.e. "shmoo") plots will be discussed in the Test Results Phase portion of this chapter. Additionally, the hardware control screen allows the user to vary the times for system calibration. Note, the GR-125 takes approximately 4 to 10 minutes for a full system calibration. [Ref. 4:p. 2-22]

(2) *Execution Control.* The execution control screens provide two major functions in the test execution procedure. First, the decision to perform a single test or multiple tests is made within this set of screens. Secondly, the specific type of test results desired for each test is annotated. Specifically, these results options include pass/fail, full tabulated results, special plots, etc. These test result options will also be covered in detail in section 4 of this chapter. [Ref. 4:p. 2-38]

(3) *Analysis Control.* The analysis control set of screens were designed to keep track of statistical failure rate data during high volume production testing. As a result, this particular subcategory of test screens are not required for individual component testing. [Ref. 4:p. 2-48]

(4) *Output Control*. All of the output control screens are reserved for use by GenRad, Inc. except for the specific Characterization Control-Output screen. However, this screen is an identical duplicate screen listed in the Debugging Control category of system test screens. Therefore, the screens in this output control subcategory are presently not required for individual component testing. [Ref. 4:p. 2-55]

(5) *Datalogging Control*. The screens located in this category perform three major functions. First, the user must determine the distribution of data desired during testing. Secondly, the selection of the Terminal Output Control screen establishes the format of the test data outputted to the system terminal. Finally, the Datalogging Output screen is used for the accumulation of test result data for a statistical evaluation of long term trends. [Ref. 4:p. 2-63]

In summary, the Test Execution Menu Screens only require the user to manipulate the Debugging Control, Execution Control and Datalogging Control categories of screens. This condition will exist until a software modification is installed to the GR-125 test system. Additionally, note that the asterisks "*" in Table XI annotate five separate screens which are conveniently collocated in one

combined screen. This utility saves the user a great deal of time during the screen editing process.

b. Results Display Menu Screens

These result display screens contain the results of the most recent test performed. These screens provide the user with a real-time display within the test format desired. Table XII illustrates the screens available in this category.
[Ref. 5:p. 2-112]

Table XII GR-125 RESULT DISPLAY MENU SCREENS

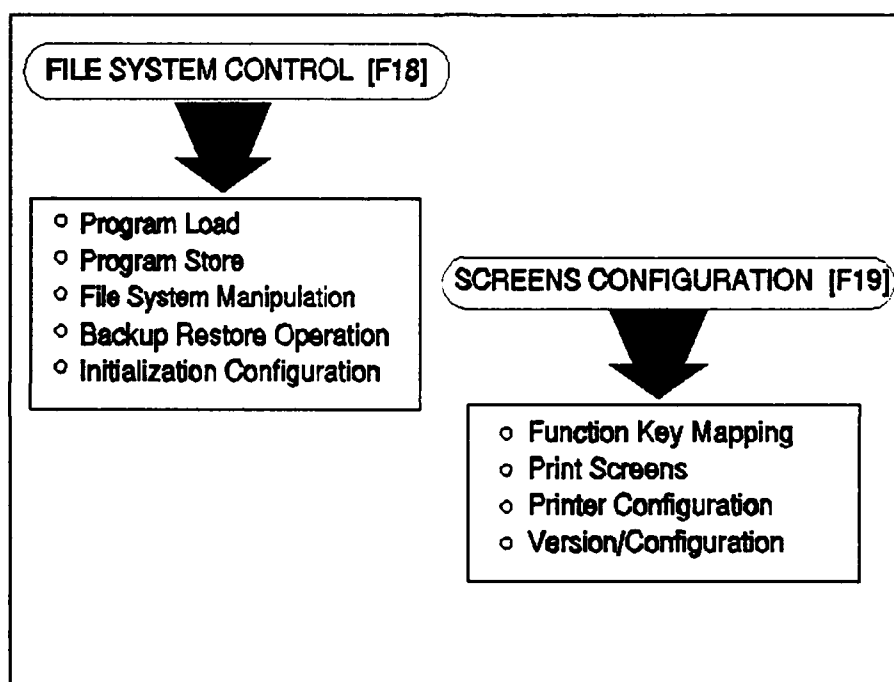
RESULTS DISPLAY [F20]

- | | |
|--------------------------|----------|
| o Batch/Lot Test | -Results |
| o Wafer Test | -Results |
| o Characterization(Plot) | -Results |
| o Characterization(Prod) | -Results |
| o Device Sequencing | -Results |
| o Test Operation | |
| o Vector History RAM | |
-

c. Input/Output Menu Screens

The Input/Output (I/O) Menu Screens provide the user with file manipulation and screen configuration control. These screens help to stream-line the testing process by giving I/O control to the user. Table XIII shows the organization of these screens.

Table XIII GR-125 INPUT/OUTPUT MENU SCREENS



(1) *File System Control.* The File System Control screens provide a broad range of user functions. The Program Load screen provides the initial starting point for the testing process. This particular screen allows the user to load a specific test file for editing or test execution.

Additionally, this set of screens provides a means for storing modifications made to an existing test file. Finally, this set of screens allows the user to manipulate other UNIX files while still within the test operation "mtest" mode. [Ref. 4:p. 2-10] The "mtest" mode is the application level above UNIX for GR-125 operation.

(2) *Screens Configuration.* The Function Key Mappings screen located in this set of screens allows the user to define SHIFT Function keys on the VT220 keyboard. Once a key is defined, the user can call a screen of interest directly. In essence, this capability offers the convenience of avoiding menu prompts and therefore, saving setup time. [Ref. 4:p. 2-89]

4. Test Results Phase

The Test Results Phase allows the user to consider which type of output result he desires from a fully edited test program. The GR-125 produces an output in one of three categories of test results. The three categories consist of a pass/fail output, actual measured values, and a pair of special diagnostic functions. This section will discuss each of these categories. Particular emphasis is given to how to edit the previously discussed system test screens in order to achieve a desired test result.

a. Pass/Fail Results

The pass/fail mode of testing provides a Go/No-Go test result. This pass/fail mode requires just two test screens to be edited in the Test Execution Phase. Recall that the Characterization Control-Combined screen actually contains five separate screens. Table XIV provides a summary of entries required to obtain a pass/fail result.

Table XIV PASS/FAIL MODE (REQUIRED SCREEN ENTRIES)

Function Key	Screen Name	Required Entry
F11	Test Operation Control-Execution	[Pass/Fail Mode]
F14	Characterization Control-Combined	
	Characterization Control-Execution	[Product Testing]
	Device Sequencing Ctrl-Execution	[Execute Binning Sequence]
	Master Output Control	[Console; Test Op level]
	Terminal Output Control	[Display Test System stats]
	Printer Output Control	[Print Test System stats]

b. Actual Measurement Results

In addition to a Go/No-Go test, the GR-125 can produce the actual measurements obtained during the testing process. Actual floating point values can be obtained for tests which involve AC and DC parametric measurements. Exceptions to this rule arise for tests which require only a pass/fail result such as a simple functional test. Section C

of this chapter will discuss the various types of tests the GR-125 is capable of doing. As with the pass/fail result, only two test screens require editing. Table XV provides a summary of the entries required to obtain an actual measured result.

Table XV ACTUAL MEASUREMENT (REQUIRED SCREEN ENTRIES)

Function key	Screen Name	Required Entry
F11	Test Operation Control-Execution	[Full Results Mode]
F14	Characterization Control-Combined	[Product Testing]
	Characterization Control-Execution	[Perform Only Specified Test
	Device Sequencing Ctrl-Execution	-list specific test # here]
	Master Output Control	[Console; Test Op level]
	Terminal Output Control	[Display Summarized Result]
	Printer Output Control	[Print Summarized Results]

c. Special Functions

The GR-125 test system provides two special diagnostic functions in the Test Results Phase of the Test Programming and Execution Methodology. These special functions give the user some reverse engineering capability. By using these two special functions, an engineer can plot various engineering characteristics data as well as determine a chip's functionality to a certain extent.

(1) "Shmoo" Plots. Test execution screens can be edited to produce a detailed 1D or 2D engineering characterization (i.e. "shmoo") plot. A wide variety of values can be plotted on a set of labeled axes. These values include current levels, voltage levels and timing data. An example of a 2D "shmoo" plot is given in Figure 10. Note, the "shmoo" plot can only be generated by executing a simple functional test. To obtain a "shmoo" plot, a total of four test execution screens must be edited. Table XVI summarizes the entries required to obtain a "shmoo" plot. [Ref. 4:p. 2-22]

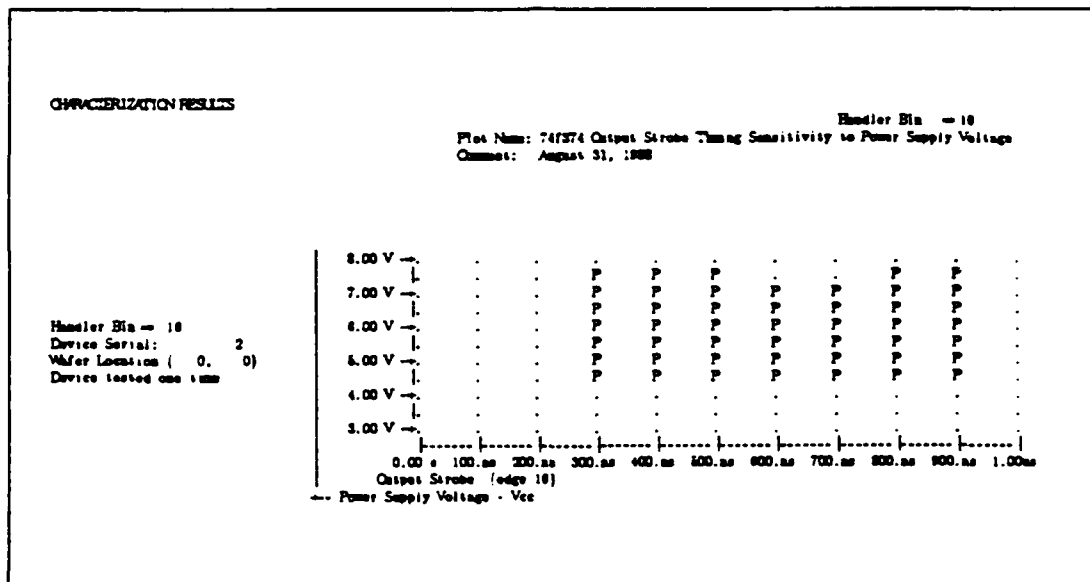


Figure 10 GR-125 "Shmoo" Plot Example [from Ref. 4]

(2) "Learn" Function. The "Learn" function also produces a reverse engineering capability. This function allows a user to determine the functionality of a specific chip. In essence, given the input test vector patterns, the

GR-125 will produce the output response pattern. Therefore, the chip function can be determined through a comparison of the input stimulus and the expected output response of the test vector patterns. The first step in executing the "Learn" function is to retrieve the Truth Table Edit screen (VT220 function key "F10"). Next, replace all the output test vector elements in the simple functional test to a "L", "H", or "X". Next, retrieve the Test Operation screen (VT220 function key "F9") for the simple functional test operation. With the curser placed on the word "Simple", toggle the space bar to get the desired "Learn" function. Once this editing is complete, execute the testing sequence. No special test screen editing is required. Upon completion of the test, return to the Truth Table Edit screen and record the new output vector element values. [Ref. 5:p. 2-64]

C. TESTING CAPABILITIES SUMMARY

The GR-125 hardware test system performs many different types of tests. This section will briefly describe each of the major test areas.

1. Functional Tests

A functional test uses test vector patterns to cycle a DUT through its truth table sequence. After applying an input stimulus pattern, the GR-125 compares the DUT's output pattern with its expected output pattern. A successful

Table XVI "SHMOO" PLOT (REQUIRED SCREEN ENTRIES)

Function key	Screen Name	Required Entry
F11	Test Operation Control-Execution	[Full Results Mode]
F14	Characterization Control-Combined Characterization Control-Execution Device Sequencing Ctrl-Execution	[1 or 2 Dimensional] [Perform Only Specified Test -list function test # here]
	Master Output Control	[Console; Char. level]
	Terminal Output Control	[Display Summarized Result]
	Printer Output Control	[Print Summarized Results]
F14	Characterization Control-Output	[table "X" and/or "Y" axis]
F14	Characterization Control-Setup	[select "Timing Array" or "Power Supply" or "Pin Levels"]

functional test requires a successful comparison of these output patterns. [Ref. 3:p. 1-44]

2. Power Supply Tests

The power supply test measures the current drawn from a selected power supply when the DUT is operating in either a static or dynamic state. [Ref. 3:p. 1-53]

3. DC Parametric Tests

Various tests for dc parametrics determine dc electrical characteristics by current and voltage measurements. These test can be classified as input or output dc parametric tests. [Ref. 3:p. 1-57]

a. Input DC Parametric Tests

(1) *Iil Test.* Leakage current is measured at a DUT input pin while forcing a logic low voltage.

(2) *Iih Test.* Leakage current is measured at a DUT input pin while forcing a logic high voltage.

(3) *Vik Test.* Voltage is measured at a DUT input pin while forcing a current.

b. Output DC Parametric Tests

(1) *Iol Test.* This test measures the DUT drive current at an output pin set low while forcing a logic low voltage.

(2) *Ioh Test.* This test measures the DUT drive current at an output pin set high while forcing a logic high voltage.

(3) *Vol Test.* This test measures the voltage at a DUT output pin while forcing the specified current with the device in the low state.

(4) *Voh Test.* This test measures the voltage at a DUT output pin while forcing the specified current with the device in the high state.

(5) *Iozl Test.* This test measures the current at a DUT output pin while the pin is in the tri-state condition and while forcing a logic low voltage.

(6) *Iozh Test.* This test measures the current at a DUT output pin while the pin is in the tri-state condition and while forcing a logic high voltage.

(7) *Ios Test*. This test measures the current at a DUT output pin while the pin is in the logic high state and while forcing a zero voltage.

4. AC Functional Tests

AC functional tests perform certain ac measurements on the DUT. These measurements include setup time, propagation delay, pulse width, hold time, and transition time. [Ref. 3:p.1-66]

5. Contact Tests

Contact tests can be classified as continuity tests. These tests check that a non-shortcd and non-open path exists from a given tester pin through the DUT to ground or a power supply connection. During testing, all of the ground pins on the DUT are connected to ground, and all of the power supplies to the DUT are set at 0 volts. [Ref. 3:p. 1-66]

III. CAD SIMULATION ENVIRONMENT

The CAD simulation environment provides the first major testing platform in the digital design process. Once a schematic design is obtained, a series of simulations determine the functional characteristics of the design. This process is repeated until a desired functionality is achieved. Figure 11 illustrates this process. When a successful simulation is obtained, a graphical plot and simulation output file are produced. The output file contains all of the stimulus and response information observed in the graphical plot. Additionally, this simulation output file is produced in an ASCII file format. Chapter IV of this thesis will discuss how to change the test vector information found in this simulation output file into the .tpp file format required by the GR-125 tester.

A. SIMULATION OVERVIEW (MENTOR GRAPHICS)

Because of availability, this thesis utilized the Mentor Graphics IDEA Series Version 7.0 CAD package for analysis. CAD simulation incorporates two basic design steps:

- Schematic Capture
- Test Simulation

A short discussion of these two design steps will illustrate the framework of the design process. The objective of this simulation overview is to provide a brief background to simulation within the CAD environment.

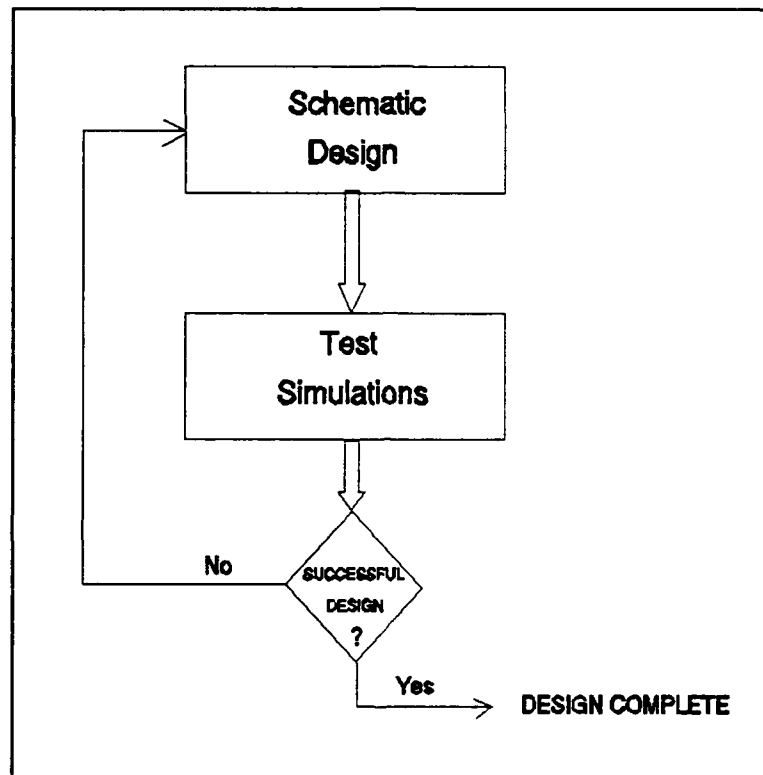


Figure 11 CAD Design/Simulation Process

1. Schematic Capture

The first step in the design simulation process is to generate a schematic diagram of the desired digital component. This process is known as "schematic capture." Schematic capture is accomplished on an IDEA Series workstation through the use of a network editor and a symbol editor. The network

editor (NETED) allows a designer to create hierarchical (multi-level) designs using a top-down approach. The symbol editor (SYMED) works with the NETED. SYMED allows the designer to draw and edit component symbols that can be placed on NETED schematic sheets. These symbols can represent basic design elements such as logic gates, transistors and off-the-shelf integrated circuit (IC) components. Together, these two schematic editors provide the resources for producing a testable design. [Ref. 6:p. 1-1]

2. Test Simulation

Once a designer has completed the schematic editing phase, he enters the actual testing procedure - simulation. As discussed previously, CAD simulation enables a designer to check the functionality of a component design. By defining input stimuli and observing the output responses, the designer's simulation is identical to the simple functional test portion of the GR-125 hardware tester. Quicksim is the simulation program used to perform the actual simulation. This CAD software tool is also included in the Mentor Graphics IDEA Series (v 7.0) package.

Quicksim is an interactive logic simulator that allows a designer to verify the functionality of the designs produced with SYMED and NETED, the schematic capture tools. Quicksim is a 12-state, timing-wheel simulator that can simulate MOS, TTL, and ECL logic. With Quicksim one can apply stimulus to

the design, run the simulation, analyze the results, and then modify the design based on those results. Stimulus is defined as the input stimuli and the expected output results data. Basically, a stimulus consists of a set of test vectors as discussed in chapter II. [Ref. 7:p. 1-1]

Quicksim accepts and produces a variety of stimulus data. This data includes various graphical and text file formats. Figure 12 illustrates the variety of Quicksim's input and output data. The input is the stimulus as defined above. The output contains the actual results of the simulation. Next, one specific output file, the List Window File will be discussed.

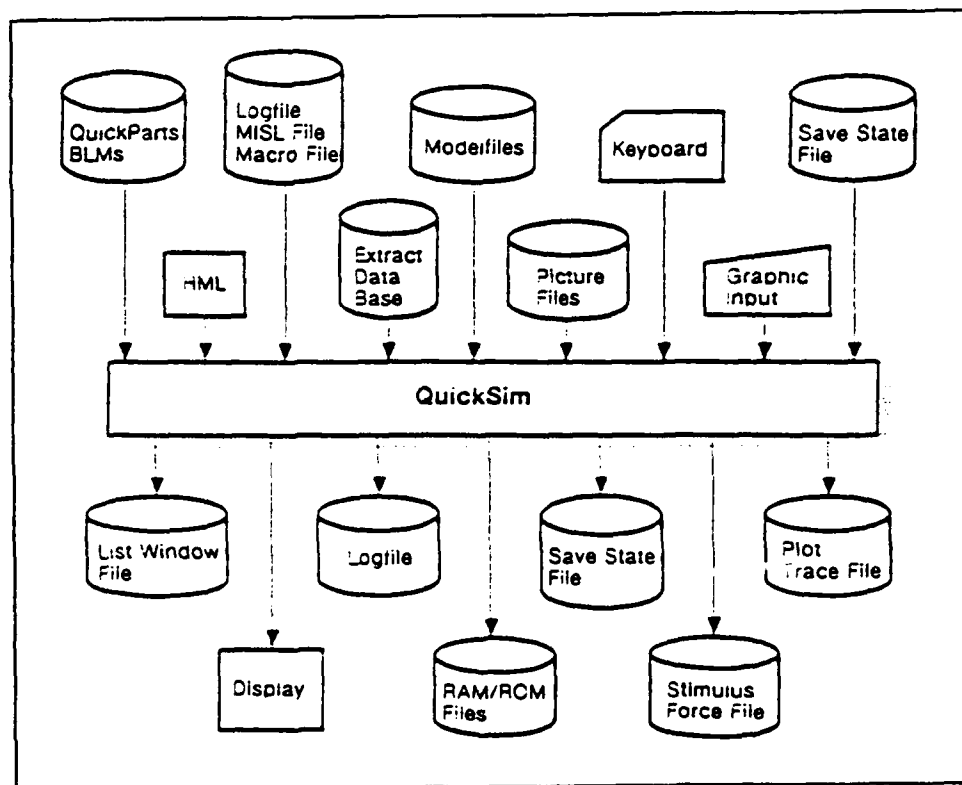


Figure 12 Quicksim Input/Output Files [from Ref.7]

B. SIMULATION OUTPUT FILE

The simulation output file, List Window, contains all of the stimulus and response data from a simulation session executed within the Quicksim environment. The command "List" is used to create the List Window file. Figure 13 shows a typical List Window file. Notice that this file contains three sections of information: time, pin labels, and pin values. The following paragraphs will briefly discuss the structure of these specific sections.

```
# USER: joeb.design.eng.D777
# DESIGN: /user/joeb/sim/work_area/control.ckt/design.ere1
# REV: 10
# VERSION: LOGIC SIMULATION SERVER V7.0_0.16 Monday, January 30, 1989
                                                    3:30:29 pm (PST)
# DATE: Friday, February 10, 1989    2:13:03 pm (PST)
# SCALE USER TIME: 1.000000    NS
# TIME STEP: 0.100000
# TRANSPORT SWITCH: Inertial delays
# SPIKE MODEL: X_Immediate
# TIMING MODEL: Typical Timing Model
    0.0 0 0 1 1 1 X X X X X
    12.7 0 0 1 1 1 X X 0 0 X
    12.4 0 0 1 1 1 X 1 0 0 X
    13.9 0 0 1 1 1 X 1 0 0 1
    50.0 0 0 1 1 1 X 1 0 0 1
   100.0 0 1 1 1 1 X 1 0 0 1
   106.9 0 1 1 1 1 0 1 0 0 1
   116.9 0 1 1 1 1 0 0 0 0 1
   150.0 0 1 1 1 1 0 0 0 0 1
   200.0 0 0 1 1 1 0 0 0 0 1
   212.4 0 0 1 1 1 0 1 0 0 1
   250.0 0 0 1 1 1 0 1 0 0 1
   300.0 0 0 1 0 1 0 1 0 0 1

TIME  ~clock  ~c      ~x1      ~x4
      ~clear  ~d      ~x2
      ~b      ~out    ~x3
```

Figure 13 QuickSim List Window Display [from Ref. 7]

1. Structure

a. Time Values

Specific time values occupy the first column of the List Window file. These times are actually user scaled units. A designer can scale these units to any desired value. For clarity, the user time unit is scaled to nanoseconds throughout this thesis. Note that a new time value is generated at every instance an input or output pin changes state. This condition allows a designer to observe how long a component takes to reach a desired output state. This time is defined as delay time.

b. Pin Labels

The pin labels section of the List Window file appears at the very end of the file. These labels actually break the List Window into separate columns. Each column is reserved for a specific input or output pin value. The first column, reserved for the time values, provides the one exception to this rule. By convention, the input pin values occur prior to the output pin values.

c. Pin Values

The pin values section consists of the various columns of single digit numbers located directly above the pin labels. These pin values can contain one of three separate signal levels, ("0", "1", "X"). Table 17 describes each of these signal levels.

Table XVII QUICKSIM SIGNAL VALUES

SIGNAL DESIGNATION	SIGNAL LEVEL
0	LOW
1	HIGH
X	UNKNOWN

In summary, once a successful simulation is complete, a designer can obtain an ASCII file containing all of the List Window data information. This new ASCII simulation output file is generated by invoking the Quicksim command summarized in Table XVIII. The `sim_output` file now contains all of the stimulus and response test vector data in an ASCII format. Chapter IV of this thesis will show how to translate the ASCII data from the `sim_output` file into the `.tpp` ASCII file format required by the GR-125 tester.

Table XVIII QUICKSIM WRITE LIST ENTRY

prompt> *WRite Llist sim_output*

2. Design Example (74S181 ALU)

An example can better illustrate a typical simulation output from the Quicksim environment. The 74S181 Arithmetic Logic Unit (ALU) provides an excellent design example for analysis. In order to illustrate the CAD simulation data discussed previously, this design example will be introduced in three areas:

- Circuit Description
- Input Stimulus
- Output Simulation File

a. Circuit Description

The 74S181 ALU performs binary arithmetic or logic operations on two 4-bit words. Figure 14 illustrates the connection diagram. Additionally, Table 19 describes the pin designations. These arithmetic operations are selected by the four function select lines (S0,S1,S2,S3), and it includes addition, subtraction, decrement and straight transfer. The internal carries must be enabled by applying a low level voltage to the carry_in (Cn). A full carry look-ahead scheme is available for fast carry generation by means of two cascaded outputs (P,G). [Ref. 8:p. 5-100]

b. Input Stimulus

The input stimulus to the 74S181 is applied through a .misl file (Refer to Figure 12). For this particular

example, the input pin values are forced to change every 10 nanoseconds. Figure 15 shows a portion of the 74S181 .misl file.

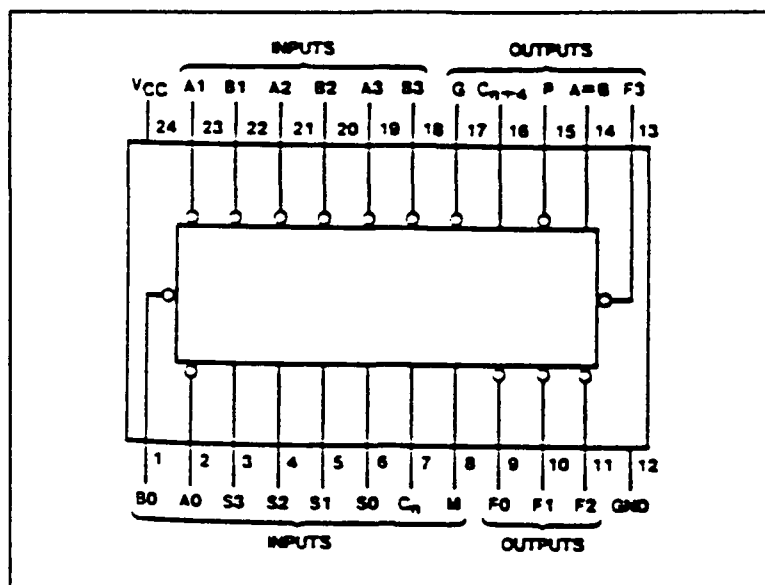


Figure 14 74S181 ALU Connection Diagram

c. Output Simulation File

After the stimulus data is entered and the List Window screen is set up within the Quicksim environment, the simulation is started. The "Write List" command is executed at the end of the simulation. The successful completion of each of these steps produces an ASCII formatted simulation output file. Figure 16 shows a portion of the simulation output file obtained for the 74S181 design example.

Table XIX 74S181 PIN DESIGNATIONS [from Ref. 8]

Designation	Pin Nos.	Function
A3, A2, A1, A0	19, 21, 23, 2	Word A Inputs
B3, B2, B1, B0	18, 20, 22, 1	Word B Inputs
S3, S2, S1, S0	3, 4, 5, 3	Function-Select Inputs
C_n	7	Inv. Carry Input
M	9	Mode Control Input
F3, F2, F1, F0	13, 11, 10, 9	Function Outputs
A = B	14	Comparator Output
P	15	Carry Propagate Output
C_{n+4}	16	Inv. Carry Output
G	17	Carry Generate Output
VCC	24	Supply Voltage
GND	12	Ground


```

CIRCUIT 74S181_test;

timedef period = 1ps;

INPUT s3 s2 s1 s0 m cin a0 a1 a2 a3 b0 b1 b2 b3;

OUTPUT p g ab cout f0 f1 f2 f3;

/* check out arithmetic functions */
s3=LO; s2=HI; s1=HI; s0=LO;
m=LO; cin=LO;
a0=HI; a1=LO; a2=LO; a3=LO;
b0=LO; b1=HI; b2=LO; b3=LO $

s3=LO at 10ns; s2=HI at 10ns; s1=HI at 10ns; s0=LO at 10ns;
m=LO at 10ns; cin=LO at 10ns;
a0=HI at 10ns; a1=HI at 10ns; a2=LO at 10ns; a3=LO at 10ns;
b0=LO at 10ns; b1=LO at 10ns; b2=LO at 10ns; b3=HI at 10ns $

s3=LO at 20ns; s2=HI at 20ns; s1=HI at 20ns; s0=LO at 20ns;
m=LO at 20ns; cin=LO at 20ns;
a0=HI at 20ns; a1=LO at 20ns; a2=HI at 20ns; a3=LO at 20ns;
b0=LO at 20ns; b1=HI at 20ns; b2=HI at 20ns; b3=LO at 20ns $

s3=LO at 30ns; s2=HI at 30ns; s1=HI at 30ns; s0=LO at 30ns;
m=LO at 30ns; cin=LO at 30ns;
a0=HI at 30ns; a1=HI at 30ns; a2=HI at 30ns; a3=LO at 30ns;
b0=LO at 30ns; b1=LO at 30ns; b2=LO at 30ns; b3=HI at 30ns $

s3=LO at 40ns; s2=HI at 40ns; s1=HI at 40ns; s0=LO at 40ns;
m=LO at 40ns; cin=LO at 40ns;
a0=HI at 40ns; a1=LO at 40ns; a2=LO at 40ns; a3=HI at 40ns;
b0=LO at 40ns; b1=HI at 40ns; b2=LO at 40ns; b3=HI at 40ns $

s3=LO at 50ns; s2=HI at 50ns; s1=HI at 50ns; s0=LO at 50ns;
m=LO at 50ns; cin=LO at 50ns;
a0=HI at 50ns; a1=HI at 50ns; a2=LO at 50ns; a3=HI at 50ns;
b0=LO at 50ns; b1=LO at 50ns; b2=HI at 50ns; b3=HI at 50ns $

s3=LO at 60ns; s2=HI at 60ns; s1=HI at 60ns; s0=LO at 60ns;
m=LO at 60ns; cin=LO at 60ns;
a0=HI at 60ns; a1=LO at 60ns; a2=HI at 60ns; a3=HI at 60ns;
b0=LO at 60ns; b1=HI at 60ns; b2=HI at 60ns; b3=HI at 60ns $

s3=LO at 70ns; s2=HI at 70ns; s1=HI at 70ns; s0=LO at 70ns;
m=LO at 70ns; cin=LO at 70ns;
a0=HI at 70ns; a1=LO at 70ns; a2=LO at 70ns; a3=LO at 70ns;
b0=HI at 70ns; b1=HI at 70ns; b2=LO at 70ns; b3=LO at 70ns $

      .
      .
      .

```

Figure 15 74S181.misl Stimulus File (partial)

0.0	0	1	1	0	0	0	1	0	0	0	0	1	0	0	X	X	X	X	X	X	X	X	X
4.0	0	1	1	0	0	0	1	0	0	0	0	1	0	0	1	X	X	X	X	X	X	X	X
5.0	0	1	1	0	0	0	1	0	0	0	0	1	0	0	1	0	X	X	X	1	1	X	X
6.0	0	1	1	0	0	0	1	0	0	0	0	1	0	0	1	0	X	1	1	1	1	1	1
7.0	0	1	1	0	0	0	1	0	0	0	0	1	0	0	1	0	1	1	1	1	1	1	1
10.0	0	1	1	0	0	0	1	1	0	0	0	1	0	0	1	1	0	1	1	1	1	1	1
14.0	0	1	1	0	0	0	1	1	0	0	0	0	0	0	1	1	1	1	1	0	1	1	1
15.0	0	1	1	0	0	0	1	1	0	0	0	0	0	0	1	1	0	0	0	1	1	0	1
16.0	0	1	1	0	0	0	1	1	0	0	0	0	0	0	1	1	0	0	1	1	1	0	1
20.0	0	1	1	0	0	0	1	0	1	0	0	1	1	0	1	0	0	0	1	1	1	0	1
24.0	0	1	1	0	0	0	1	0	1	0	0	1	1	0	1	0	0	0	1	1	0	1	1
25.0	0	1	1	0	0	0	1	0	1	0	0	1	1	0	1	0	0	0	1	1	1	0	0
26.0	0	1	1	0	0	0	1	0	1	0	0	1	1	0	1	0	0	0	1	1	1	1	1
27.0	0	1	1	0	0	0	1	0	1	0	0	1	1	0	1	0	1	1	1	1	1	1	1
30.0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	1	1	0	1	1	1	1	1	1
34.0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	1	1	1	1	1	1	0	1	1
35.0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	1	1	0	0	0	1	1	1	1
36.0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	1	1	0	1	1	1	1	1	1
40.0	0	1	1	0	0	0	1	0	0	1	0	1	0	1	1	1	0	1	1	1	1	1	1
44.0	0	1	1	0	0	0	1	0	0	1	0	1	0	1	1	1	1	1	1	0	0	0	0
45.0	0	1	1	0	0	0	1	0	0	1	0	1	0	1	1	0	0	0	0	1	1	0	0
46.0	0	1	1	0	0	0	1	0	0	1	0	1	0	1	1	0	0	0	1	1	1	1	1
47.0	0	1	1	0	0	0	1	0	0	1	0	1	0	1	1	0	1	1	1	1	1	1	1
50.0	0	1	1	0	0	0	1	0	0	1	0	0	0	1	1	0	1	1	1	1	1	1	1
54.0	0	1	1	0	0	0	1	0	0	1	0	0	0	1	1	1	1	1	1	0	1	1	1
55.0	0	1	1	0	0	0	1	1	0	1	0	0	0	1	1	0	0	0	0	1	1	1	0
56.0	0	1	1	0	0	0	1	1	0	1	0	0	0	1	1	0	0	0	1	1	1	1	1
57.0	0	1	1	0	0	0	1	1	0	1	0	0	0	1	1	0	0	1	1	1	1	1	1
60.0	0	1	1	0	0	0	1	0	1	1	0	0	1	1	1	0	1	1	1	1	1	1	1
64.0	0	1	1	0	0	0	1	0	1	1	0	0	1	1	1	1	1	1	1	0	0	1	1
...																							
590.0	1	0	0	1	0	1	0	0	0	0	0	0	1	0	1	1	0	0	0	0	1	1	1
593.0	1	0	0	1	0	1	1	0	0	0	0	0	1	0	0	0	0	0	0	0	1	1	1
594.0	1	0	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1
595.0	1	0	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0
600.0	1	0	0	1	0	1	1	0	0	1	1	1	0	0	0	0	0	0	1	0	1	0	0
603.0	1	0	0	1	0	1	1	0	0	1	1	1	0	0	1	0	0	0	1	0	0	0	0
604.0	1	0	0	1	0	1	1	0	0	1	1	1	0	0	1	0	0	0	1	0	0	0	0
605.0	1	0	0	1	0	1	1	0	0	1	1	1	0	0	1	0	0	0	1	0	1	0	1
610.0	1	0	0	1	0	1	1	1	0	1	0	1	0	1	1	0	0	0	1	0	1	0	1
614.0	1	0	0	1	0	1	1	1	0	1	0	1	0	1	1	0	0	0	0	0	1	0	0
615.0	1	0	0	1	0	1	1	1	0	1	0	1	0	1	1	0	0	0	0	0	1	0	0
620.0	1	0	0	1	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
624.0	1	0	0	1	0	1	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	1	0
625.0	1	0	0	1	0	1	0	0	0	1	0	0	0	0	1	0	0	1	0	1	1	0	0
630.0	1	0	0	1	0	1	1	0	0	1	0	0	0	1	1	0	0	1	0	1	1	0	0
634.0	1	0	0	1	0	1	1	0	0	1	0	0	0	1	1	0	0	1	0	0	0	1	1
635.0	1	0	0	1	0	1	1	0	0	1	0	0	0	1	1	0	0	1	0	0	1	1	1
TIME	cs3	cs1	cm	ca0	ca2	cb0	cb2	cp	cad	cf0	cf2												
	cs2	cs0	ccin	ca1	ca3	cb1	cb3	cg	ccout	cf1	cf3												

Figure 16 74S181.list Sim_output File (partial)

IV. SOFTWARE TRANSLATION METHODOLOGY

A. DISCUSSION

This thesis has addressed two separate digital testing environments. As discussed in chapter III, the GR-125 hardware tester enables a designer to perform many different types of tests including a functional test. Additionally, chapter IV described how the QuickSim CAD simulator offers a functional test capability within the simulation test environment. A close comparison of the functional test requirements within each of these environments reveals an interesting similarity: the stimulus/response data required for each test environment contains the same general information. The only difference lies in its structural format.

As discussed in chapters II and III, the stimulus required for both test environments is composed of test vector elements. Although these test vector elements contain essentially the same stimulus information, their input format is quite different between the GR-125 and the QuickSim test environments. Recall that the GR-125's test vector stimulus is located in the ASCII formatted .tpp file (refer to Figure 7). In contrast, test vector stimulus for the QuickSim simulator originates in a .misl file (Figure 15). After

simulation these test vector stimulus elements and their response patterns are recorded in the list window .list file (Figure 16).

An enormous amount of time and effort is required to generate a set of stimulus test vector patterns. These patterns can easily exceed thousands of lines of data elements. Furthermore, manually copying these test patterns into two formats can lead to many inadvertent editing errors. Accordingly, finding a way to make these two test environments compatible with each other is extremely advantageous. As a result, developing a software translation program will effectively link the digital simulation environment with the GR-125 hardware tester environment. This process will translate the test vector patterns generated by the QuickSim simulator into an acceptable format for the GR-125 .tpp file. The desired translation process is illustrated in Figure 17.

The software translation procedure described above reads an input file, performs various editing, and produces a desired output file. This process is actually performing the function of a mini compiler or interpreter. This chapter will discuss how various software tools can be used to build such an appropriate translator. Finally, chapter V will present the actual translator results.

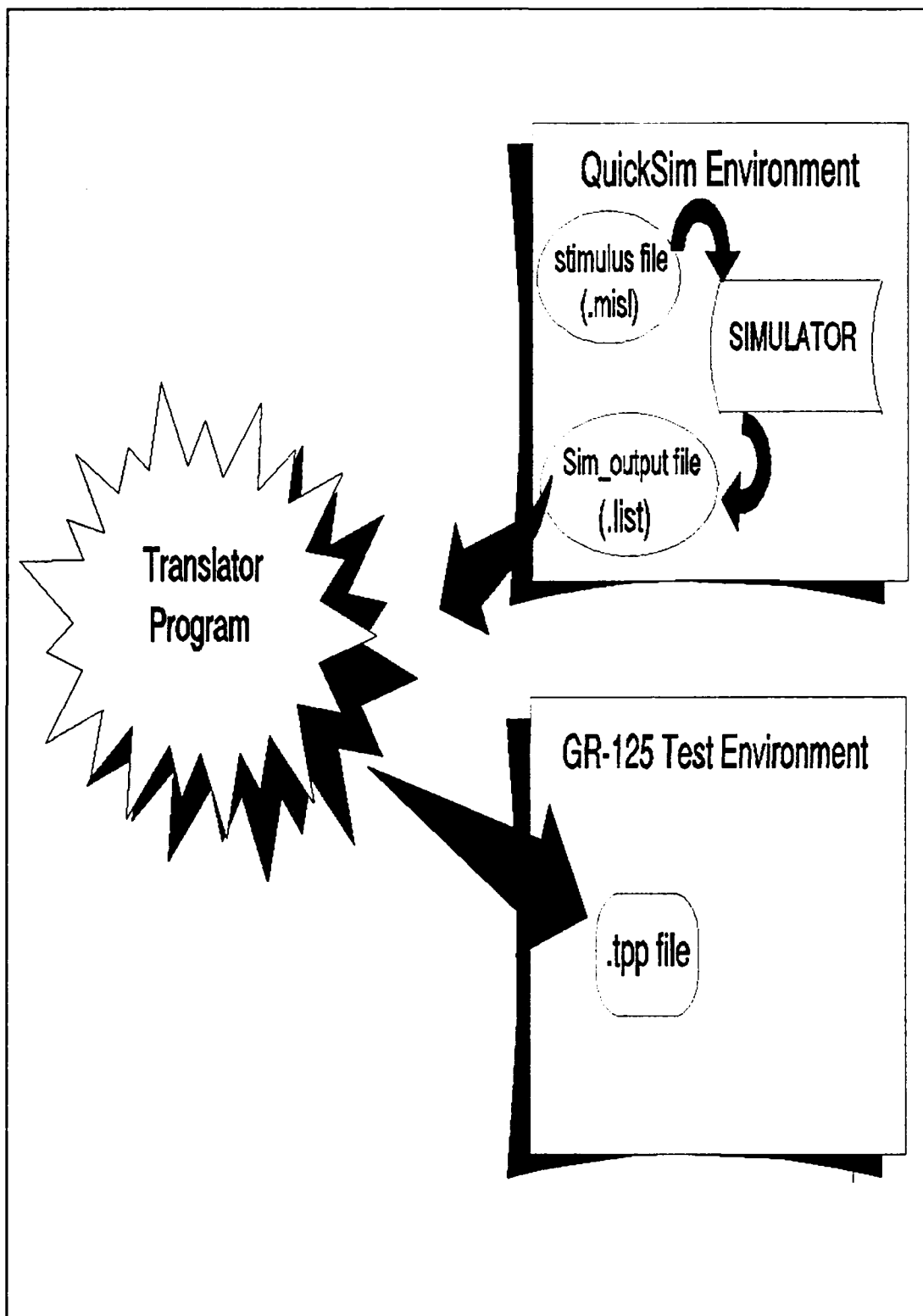


Figure 17 Test Vector Translation Procedure

B. INTERPRETERS AND COMPILERS

As discussed above, a compiler and/or interpreter are the heart of any software language translation. A compiler inputs a program and converts it into a set of instructions that can be performed by the computer. The input for a compiler typically spans multiple lines. In comparison, an interpreter acts immediately on the user's typed input, one line at a time. Compilers and interpreters are very similar in how they process input and generate output; therefore, this thesis will use the term compiler to mean both interpreter and compiler. The input to a compiler is a character stream. Alternately, the output of a compiler is an action or series of actions, possibly as simple as printing an output identical to the input.

The compiler performs its function in three separate stages:

- lexical analysis
- parsing
- actions

The first stage, lexical analysis, scans the input stream and converts various sequences of characters into groups known as tokens. Tokens are groups of characters predefined by the compiler writer. In the second stage, a parser reads these newly created tokens and assembles them into language constructs. The constructs of a language actually describe

how expressions, identifiers, and keywords can be combined to form statements. For example, the "if-then" statement in Ada is a language construct. Finally, in the third stage of a compiler, actions were taken once a token is matched. Every stage is important. The completion of one stage provides the input for the next stage. However, in less complex applications, the action stage can immediately follow the lexical analysis stage. Figure 18 summarizes these stages. A programmer could write a custom analyzer or parser in any computer language. However, there exists some special C based UNIX tools which offer superior flexibility and capability in compiler design.[Ref.9]

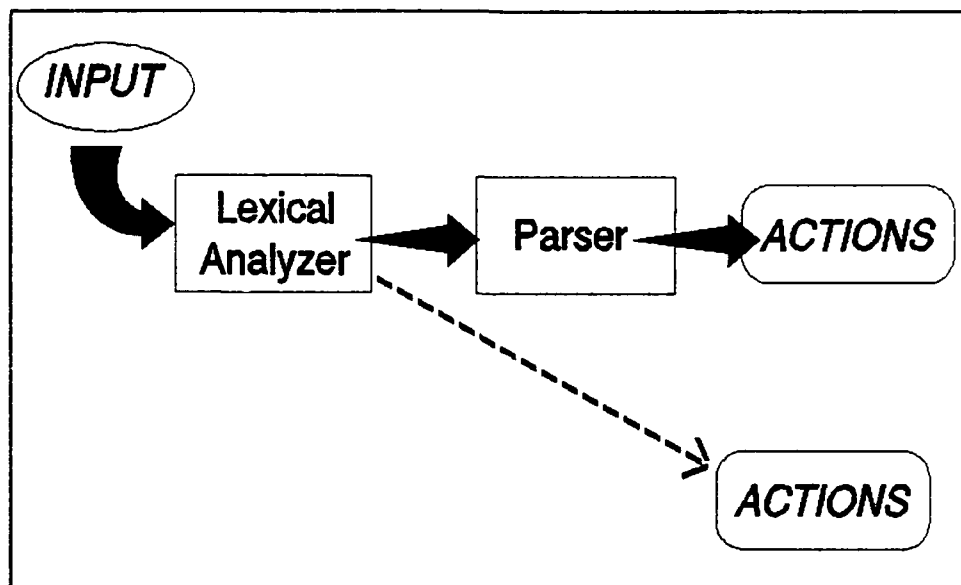


Figure 18 Compiler Processing Stages

C. UNIX TOOLS OVERVIEW

Special UNIX tools exist which makes compiler design rather simple and straight forward. This chapter will analyze two specific UNIX utilities which can be used to design a translation program:

- Lex (Lexical Analyzer Generator)
- Yacc (Yet Another Compiler Compiler)

Lex and yacc are specifically designed for writing compilers. These tools create C routines that analyze and interpret an input stream of characters to produce a desired output product. Both of these utilities were developed at Bell Laboratories in the 1970's. Additionally, lex and yacc have been standard UNIX utilities since Version 7. Figure 19 provides a graphical comparison of the power of various tools in the UNIX programming toolkit. Note that lex and yacc are powerful but still provide a programmer with tools not so complex as C itself. [Ref. 9:p. xiv]

D. LEXICAL ANALYZER GENERATOR (LEX)

1. Background

Lex performs the lexical analysis function of a compiler. Specifically, lex reads an input file containing regular expressions for pattern matching and generates a C routine that performs lexical analysis. As discussed previously, this routine will read a stream of characters and

match predefined sequences as tokens. These input streams are byte streams in UNIX. Lex, therefore, breaks these byte streams up into tokens. Once these tokens are assembled, lex can choose between two options:

- (1) pass the tokens to yacc for future action
- (2) perform immediate action based on a token match

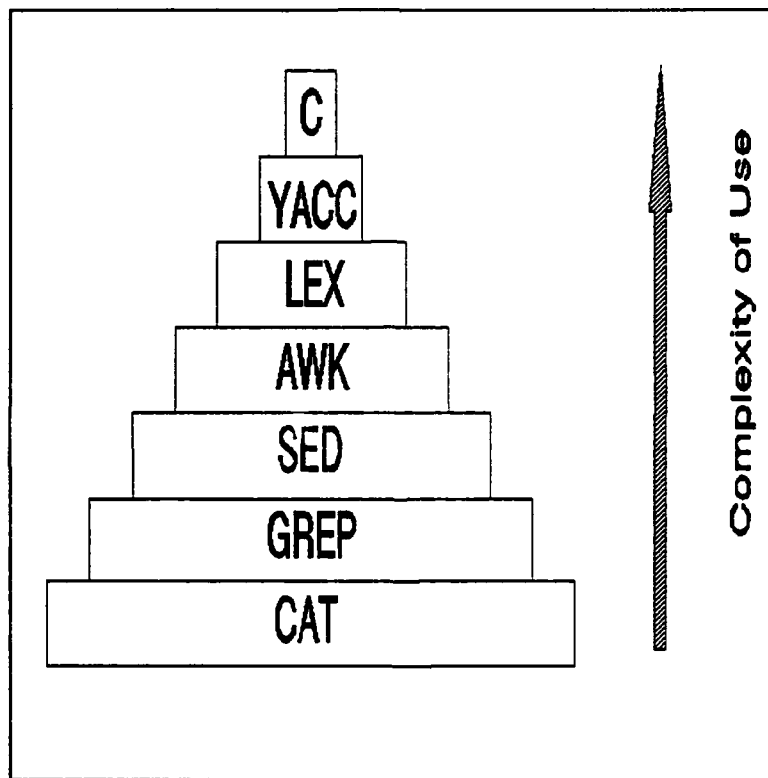


Figure 19 UNIX Toolkit Hierarchy

2. Lex Specification Format

The structure of a lex program is known as a lex specification file. Figure 20 delineates the three sections which form a full or complete lex specification. The first

and last sections are optional entries. Consequently, a lex specification can actually be composed of only the rules section. Although each section will be addressed, only the rules section will be covered in detail. By convention, the lex specification is created in a file using a ".l" suffix.

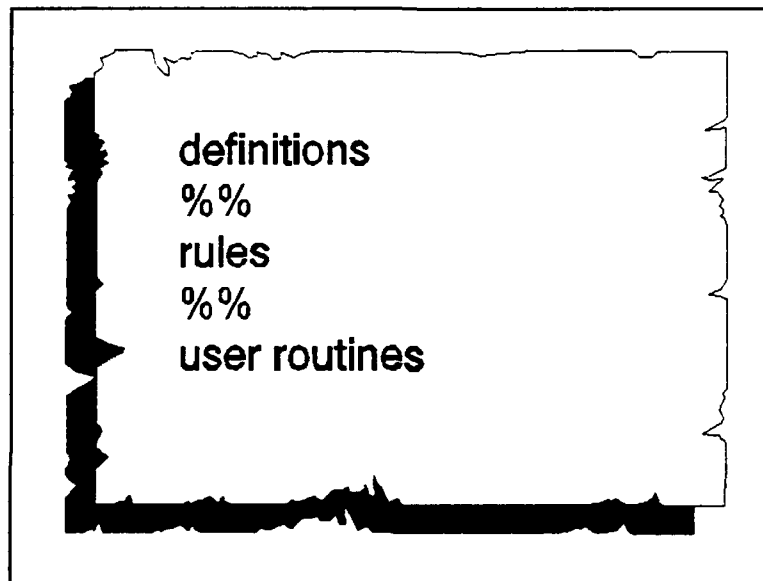


Figure 20 Full Lex Specification Format

a. Rules Section

The main section of a lex specification is composed of a set of rules. Two percentage signs "%%" are a required symbol to indicate the start of this section. Each rule contains a regular expression that is matched against an input stream. Once this **match** is made a specified **action** is taken. These pattern matching rules are expressed in UNIX regular expression syntax. Figure 21 illustrates a simple lex

specification with a single rule. In this rule "Navy" is a regular expression in which each character is interpreted literally. The action is composed of the C library function "printf". Basically, this lex specification states that if the token "Navy" is recognized in the input stream, then "Beat Army" is printed. Note, however, that if the input does not match any of the regular expressions explicitly defined in the rules, a default action is executed. This default action will copy the input to the output with no modifications made. Therefore, a lex specification with no specified rules will completely copy or echo the input to the output. Consequently, if a programmer wants to restrict the output, explicit rules must be written to match the input and then discard it.



```
%%  
Navy    printf("BEAT ARMY")
```

Figure 21 Lex Specification Rule

A lex specification can actually be thought of as an input scanner which scans the input stream and executes a set of actions. This is the concept which will be implemented to develop the translator program in chapter V. The key to an effective input scanner is properly defining the regular expressions in the rules section. Analyzing a specific regular expression with specifically defined expression

operators will help to explain its usage. Table XX provides a simple example of a regular expression representing real numbers. These real numbers consist only of digits and decimal points. It is advantageous to break this regular expression into two parts for analysis. Looking at the second part first reveals:

`[0-9]+`

The brackets `[]` enclose a set of exclusive choices. A consecutive range of digits or letters within brackets can be abbreviated by the use of a hyphen. This particular expression matches any single digit from 0 to 9. A plus `+` symbol means one or more of the preceding. Therefore, this part of the expression matches `"2"`, `"223"`, or any sequence of digits. Now, a look at the first part of this expression reveals:

`(([0-9]*\.)*)`

The asterisk `"*"` means zero or more of the preceding. Parentheses `"()"` are used to group an expression so that it can be modified as a single unit. As a result, the asterisk following the expression in parentheses makes the entire expression optional. Additionally, the asterisk following the `"[0-9]"` makes the digits preceding the decimal point optional as well. The dot `"."` normally is used to match any character except a newline `"\n"`. However, in this example, a backslash `"\"` is used to make the dot be taken literally. Therefore, this part of the expression matches a decimal point preceded

by any sequence of digits. Table XXI provides a listing of the regular expression operators used in lex. For a more detailed discussion of the syntax required for regular expressions, refer to chapter 6 of Ref. 9.

Table XX LEX REGULAR EXPRESSION EXAMPLE

Numbers desired to match:

223

2.2

2

22.32

Regular expression:

`([0-9]*\.)*[0-9]+`

b. Definition Section

The definition section of a lex specification is optional. However, this section does allow a programmer to define simple macros for use in the rules section discussed above. For example, the regular expression expressed in Table XX could be defined in the definitions section as follows:

`real_num ([0-9]*\.)*[0-9]+`

Therefore, the term "real_num" followed by an appropriate action would constitute a valid rule without having to rewrite the full expression.

c. User Routine Section

The user routine section is also an optional section in the lex specification. This section can contain any valid C coded routines. Frequently, however, this section will have no code since the necessary routine will be provided by the lex library. This lex library is discussed in the next section on usage.

Table XXI LEX REGULAR EXPRESSION OPERATORS [from Ref. 9]

Character	Meaning
.	Matches any single character (except newline).
\$	Matches the end of the line as trailing context.
^	Matches beginning of line, except inside [] when it means "complement".
[]	Matches any of the specified characters.
-	Inside [], if it is not the first or last character, means "the range of".
?	The previous regular expression is optional (e.g., 10?9 is 109 or 19).
*	Any number of repetitions, including zero.
+	Any positive number of repetitions, but not zero.
	Allows alternation between two expressions (e.g., 10 11 matches 10 or 11).
()	Allows grouping of expressions.
/	Matches an expression if followed by the next expressions (e.g., 10/11 matches 1011).
{ }	Allows repetitions or substitutes a definition.
<>	Defines a start condition.

3. Usage

There are three steps required to run lex. Figure 22 describes each of these steps. It is important to note that the lex.yy.c file, created in step 2, is not a complete

program. It contains a lexical analysis routine called "yylex". Consequently , there are two ways to call yylex:

- Supply a hand-coded main routine that calls yylex()
- Integrate the lexical analyzer with a yacc-generated parser

The second method of calling yylex() will be addressed in the next section on yacc. The actual translator program, which is developed in chapter V, will utilize a separate main routine to call yylex(). Finally, the program compilation in step 3 requires the "-ll" option. This compiler option is required. By invoking this "-ll" option lex.yy.c is linked with the UNIX standard library "libl.a".

E. YET ANOTHER COMPILER COMPILER (YACC)

1. Background

Recall that the second stage of a compiler process involves a parsing routine. Refer back to Figure 18. As mentioned earlier, the parser reads the tokens created by the lexical analyzer and assembles them into language constructs. These constructs will then be used to describe how expressions, identifiers, and keywords combine to form statements. Yacc performs the duty of a parser. Basically, yacc reads a specification file that codifies the grammar of a language and generates a parsing routine. This parsing routine will then group the tokens produced from a lexical

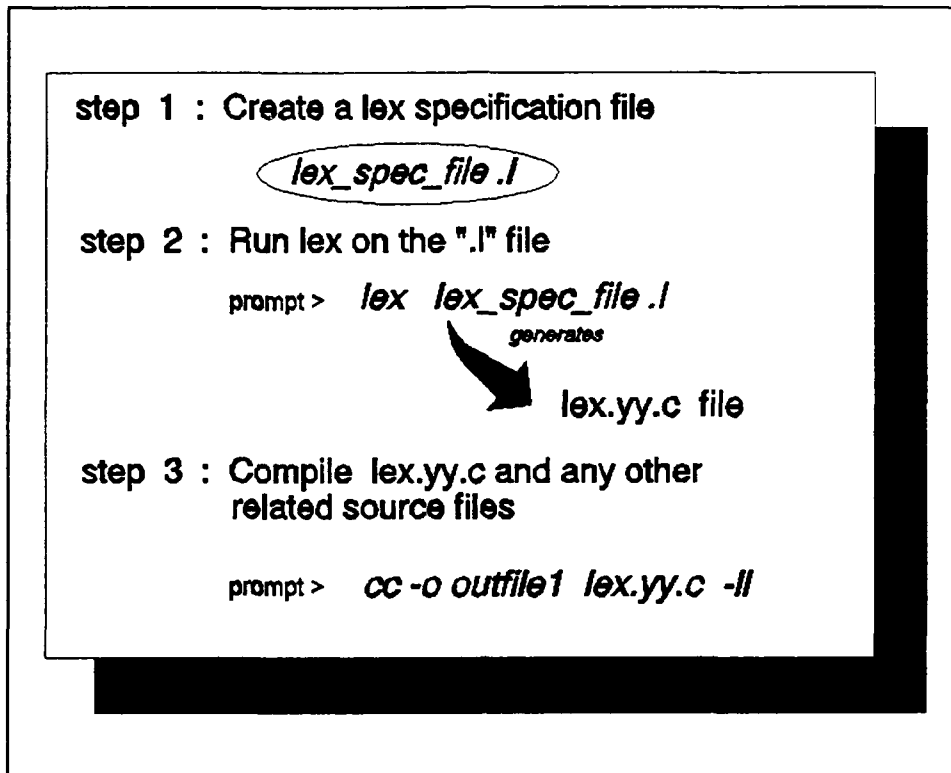
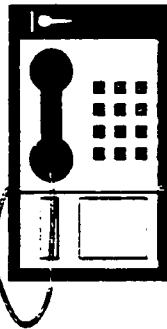


Figure 22 Lex Usage Steps

analyzer into meaningful sequences and take action as specified in the action routines. Figure 23 taken from Ref. 9 describes the basic function of a parser. In summary, it is important to recognize the fact that a parser like yacc must have an associated lexical analyzer to provide it with tokens. Yacc will not function as a stand alone routine like lex.

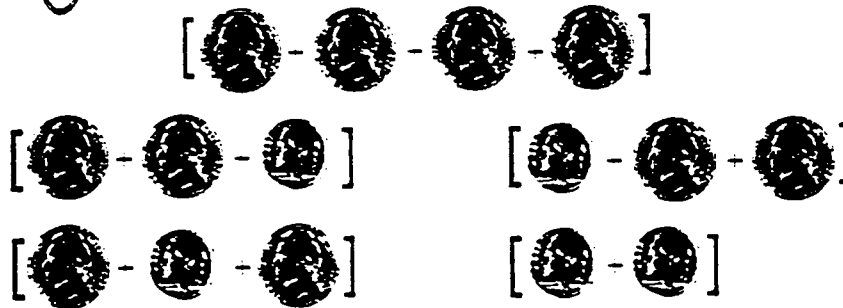
2. Yacc Specification Format

The yacc specification format closely parallels the lex specification format. Figure 24 illustrates the three sections which form a full yacc specification. The declarations section and the grammar rules section are both



The *lingua franca* of a Pay Phone

To understand what a parser does, let's describe it by analogy to a pay telephone. To place a call, it costs 20 cents, and that 20 cents can be paid using nickels and dimes. Each coin represents one token. The syntax of our language must state what combinations of tokens make up 20 cents. The following rules describe these combinations:



For example, if the first coin is a nickel and the second coin is a dime, we do not yet have a valid combination, and to produce one, we need a third coin that is a nickel. Each of these lines can be considered rules for producing a valid combination totaling exactly 20 cents. The "machine" is able to apply these rules by "ruling out" the ones that are no longer valid. For instance, if the first coin is a dime, we know that only the last two rules can be applied. If the next coin is a nickel, then only the fourth rule is left to be applied on remaining input. Parsing, then, is the ability to recognize certain sequences of tokens.

The above set of rules have the same action associated with them, which might be "connect caller." We could write rules to recognize other tokens and to specify different actions. For instance, we might have a rule for pennies and slugs, dropping the token into the coin return slot. Similarly, we could have a rule:



and specify an action that returns the nickel and makes the connection. (This, of course, is not a *real* pay phone.) The set of rules constitute a *grammar*. In other words, a *grammar* describes the combinations of tokens that produce meaningful results.

Figure 23 Parsing Description [from Ref. 9]

required for a complete yacc specification. By convention, a yacc specification file uses a ".y" suffix.

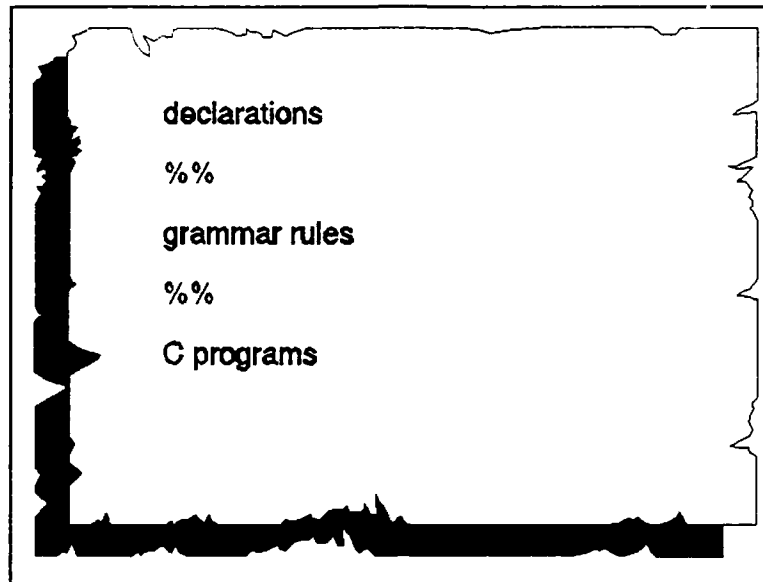


Figure 24 Full Yacc Specification Format

a. Declarations Section

The declarations section establishes the framework throughout the parser. The tokens and operators, which originated from the lexical analyzer, are defined here. The actual form of the token is declared as well as any other global variables that will be used. These token definitions describe all the possible tokens that the lexical analyzer will return to the parser. Recall, yacc was developed to help translate one software language into another. Any generic language will have text, comments, commands, numbers, etc. Therefore, tokens are used to define these different language elements. Table XXII shows a typical declaration in a yacc

specification. As discussed above, the declaration section also defines the operators used in the parser. Table XXIII lists several keywords and their associated meanings which can be used in the declaration section. Refer to chapter 7 of Ref. 9 for additional information.

Table XXII YACC DECLARATION ENTRY

```
% token < val > NUMBER
% token < text > COMMENT
% token < cmd > COMMAND
% token < text > TEXT
```

**Table XXIII YACC DECLARATION SECTION
KEYWORDS [from Ref. 9]**

%token	Declare the names of tokens.
%left	Define left-associative operators.
%right	Define right-associative operators.
%nonassoc	Define operators that may not associate with themselves.
%type	Declare the type of nonterminals.
%union	Declare multiple data types for semantic values.
%start	Declare the start symbol. Default is first in rules section.
%prec	Assign precedence to a rule.

b. Grammar Rules Section

The grammar rules section of the parser is where all of the action in yacc takes place. As in lex, there are production rules followed by action statements. However, the rules section in yacc is quite a bit more complicated than lex. A complete grammar rule in this section is composed of three elements:

- symbol
- definition
- action

Figure 25 shows the format of a yacc grammar rule.

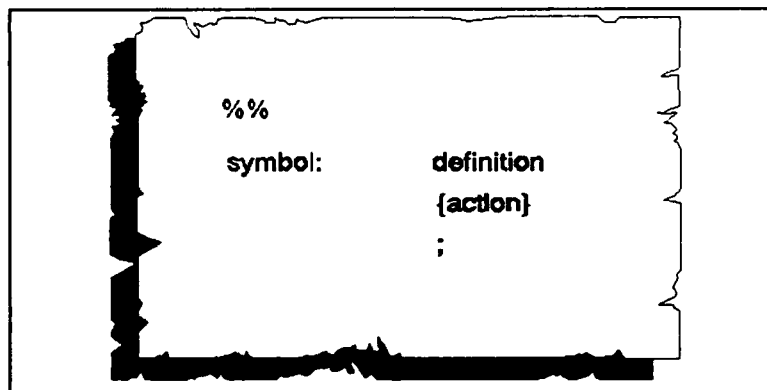


Figure 25 Yacc Grammar Rule Format

(1) *Symbol*. There are two types of symbols used in yacc: "terminal" and "nonterminal". A terminal symbol is an actual token or literal character that is recognized by the lexical analyzer. Conversely, a nonterminal is strictly

defined as a non-token. By convention, the names of nonterminal symbols are written in lower case letters while the names of terminal symbols are capitalized. These two symbols should not be confused with the symbol location in the left hand side of the grammar rule. Only a nonterminal symbol is allowed in this symbol location. Alternatively, the right hand side of the grammar rule, the definition location, can be made up of both terminal and nonterminal symbols.

(2) *Definition.* The definition portion of a yacc grammar rule consists of zero or more symbols made up of terminal and nonterminal symbols. The syntax of this section is essentially a hierarchical structure which uses a top-down structure relating various terminal and nonterminal symbols. Table XXIV provides an example of how these various symbols interrelate. Recall that the capitalized words are tokens (i.e. terminal symbols). The first rule in this example states that a nonterminal symbol, "list", is made up of either an object or of a list and an object. Note the use of recursive definitions. The pipe "|" symbol is used as a union operator. Finally, the last rule in Table XXIV specifies that the nonterminal symbol number is either a NUMBER, a NUMBER with a plus "+" in front, a NUMBER with a minus "-" in front, or two numbers separated by a decimal point ".".

The construction of this grammar clearly shows a bottom-up process. Each grouping is included in larger

groupings until there is a single top-level grouping that includes all other groupings. This top level language construct is referred to as the "start" symbol. In the sample of Table XXIV "list" is the start symbol. When the start symbol is recognized and there is no more input, then the yacc parser knows it has seen a complete program. [Ref. 9:p. 12]

(3) *Action*. The action within a yacc grammar rule consists of one or more C language statements similar to a lex action statement. These actions are executed each time a corresponding rule is matched. Actions usually manipulate the values of tokens.

Table XXIV YACC GRAMMAR RULE ELEMENTS

list	←	object list object
object	←	string number
string	←	TEXT COMMENT COMMAND
number	←	NUMBER '+' NUMBER '-' NUMBER NUMBER '.' NUMBER

c. C Programs Section

The C programs section of a yacc specification is composed of C coded routines. This section performs the identical function of the user routine section found in the lex specification.

3. Usage

There are five steps to creating a yacc parser. Figure 26 describes each of these steps.

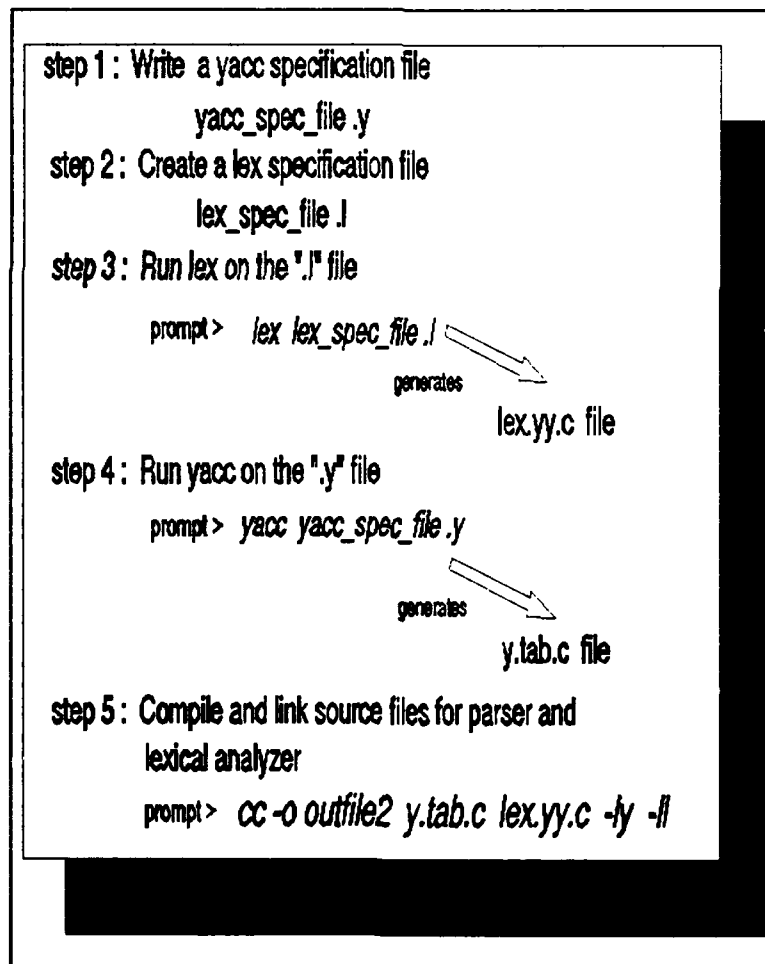


Figure 26 Yacc Usage Steps

The y.tab.c file is not a stand alone routine similar to the lex.yy.c file. Step 5 of Figure 26 requires the "-ly" option in addition to the "-ll" option. The order is important between the lex and yacc library extensions. The overall use of lex and yacc are summarized in Figure 27.

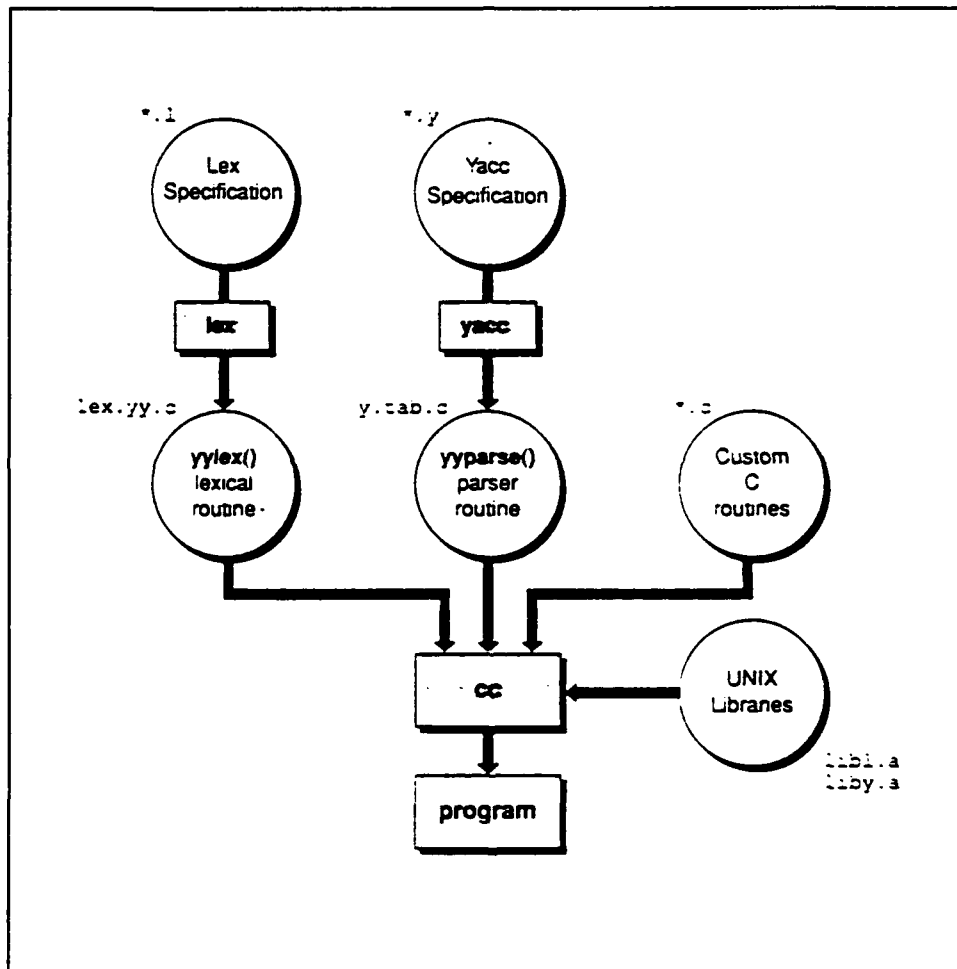


Figure 27 Lex And Yacc Usage Summary [from Ref. 9]

4. Flow Control Summary

Lex and yacc have each been analyzed separately. However, the lexical routine created by lex, yylex, and the parsing routine created by yacc, yyparse, work together. Figure 28 illustrates the flow of control in lexical and parsing routines.

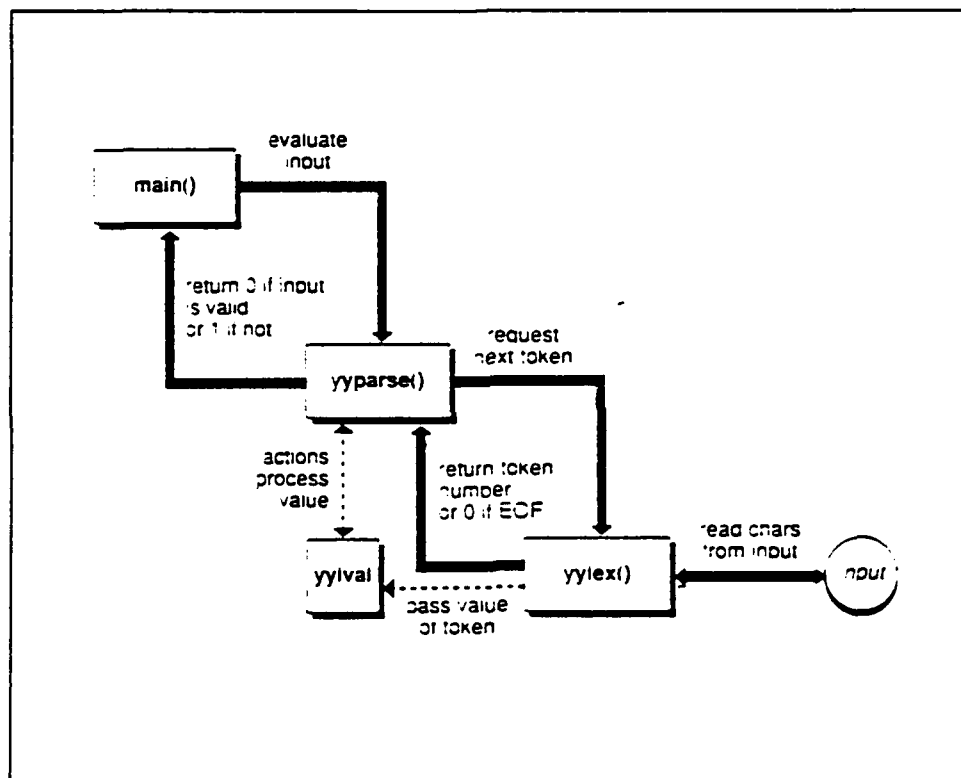


Figure 28 Lex And Yacc Flow Control [from Ref. 9]

The main program invokes yyparse to evaluate whether the input is valid or not. Next, yyparse invokes the yylex routine each time it needs a token. The lexical routine reads the input stream and returns a token number to the parser for each token

it matches. The token number lets yacc know which token has been received. The token number corresponds to the ASCII value of each ASCII character (0 to 256). Thus special user-defined tokens begin at 257. These special user-defined tokens are defined in the definitions section of a lex specification. Additionally, the lexical routine can also pass the value of the token using the external variable "yylval". Once the lexical routine has exhausted the input, it returns a "0" to the parser. If the parser has recognized the start rule, then the parser returns a "0" which means that the input is valid. If the parser receives a token number or a sequence of tokens that it does not recognize or if the lexical routine returns 0 (end of file) when the start symbol has not been recognized, then the parser returns 1, reporting a syntax error. [Ref. 9:p. 14]

This chapter has described the software translation procedure by comparing it to a compiler process. The special UNIX tools, lex and yacc, provide ideal resources for building such a language translation program. Although lex and yacc work well together, lex is extremely powerful on its own. Lex's ability to both scan an input and take actions based on that input data makes lex an effective "stand alone" compiler system. As a result of this capability, lex alone will be used in chapter V to build an actual translator to modify a CAD simulator output file into the compatible format required by the GR-125 tester system.

V. TRANSLATOR DESIGN RESULTS

Chapter IV discussed the benefit of developing a software translation tool to link the digital design environment with the GR-125 hardware tester environment. Additionally, a software translation methodology was presented incorporating special UNIX tools such as lex and yacc. This chapter will present the results of an actual translation process which provides a solution to the test vector incompatibility problem between these two test environments. After a brief overview, the structure, usage, and results of this translator program will be presented.

A. OVERVIEW

The overall objective of this translator program is to translate the test vector patterns generated by the QuickSim simulator into the structural format required by the GR-125 .tpp file. The List Window file produced by the QuickSim simulator will provide the input for this translator. The output file produced from the translator will then be included in the GR-125 .tpp file (refer back to Figure 17). Because of the extreme capability and flexibility of the lexical analyzer generator (lex), this special UNIX tool alone will provide the backbone for this translator design. Furthermore, in order to maintain continuity for discussion, this translator program

will perform manipulations on the 74S181 ALU simulation files developed back in chapter III. The name of this translator program is "vector_map".

B. PROGRAM STRUCTURE

The basic components of the translator program, vector_map, consist of a main program and a lex routine. The main program, vector_map.c, is a C based program which calls the lex routine, vector_map.l, to perform the token matching and corresponding execution functions. The lex routine is the heart of the vector_map translator. This section will discuss the composition of each of these two routines.

1. Main Program (vector_map.c)

The entire vector_map.c program is presented in Figure 29. This main program performs two major functions:

- provides a location for inputting the number of input pins
- calls the lex routine

The correct number of input pins are required by vector_map.l in order to function properly. As discussed in chapter IV, vector_map.c uses the function yylex() to call the lex routine.

2. Lex Routine (vector_map.l)

As stated above, the lex routine does most of the work in the vector_map translator. Figure 16 presents a typical input file scanned by the vector_map.l routine. In order to

produce a test vector stimulus file compatible with the GR-125 .tpp file format, three major alterations must be performed. Table XXV delineates these translation requirements. The vector_map.l routine ,provided in Figure 30, scans different stimulus elements by keeping track of the number of spaces encountered. These spaces are defined as a "field_count" in this specific lex routine.

```

/***** vector_map.c *****/

/* Usage:  vector_map num
   ....where "num" is the number of input pins from the
   ....input file ... for our test case, the input file
   ...."74S181.list" has 14 input pins
*/

int   input_num;

main(argc, argv)
int argc;
char *argv[];

{
    if(argc > 1)
        input_num = atoi(argv[1]);
    else
        input_num = 1;
    yylex();
}

```

Figure 29 "vector_map.c"

As mentioned in chapter IV, every character a lex routine encounters on input will be copied directly to the output unless explicitly defined as a token with a

corresponding action statement. Consequently, the first requirement, listed in Table XXV, is satisfied by matching unwanted characters and/or character strings and deleting these matched tokens with a semicolon ";". This action is accomplished in the last four lines of the `vector_map.l` routine.

Table XXV `"vector_map.l"` COMPILATION STEPS

1. Remove unwanted Characters
 2. Produce input vector pattern immediately followed by the final state output pattern
 3. Change output pattern elements "0" and "1" to "L" and "H" respectively
-

Lex creates an external variable named `"yytext"` that contains the string of characters that are matched by the regular expression defining the token. Therefore, changing the value of this variable provides a solution to the third translation requirement of Table XXV. Because `yytext` is a string type variable, the C function `"atoi"` is used to convert it to an integer value prior to comparing it to the integer

```

.....
**** vector_map.1 ****
****
.....
/* LEX SPECIFICATION */
.....
* "definitions section" *
.....
pin  ["^"] [a-zA-Z]* [a-zA-Z0-9]*
    extern int input_num;
    int field_count = 1;
    int flag = 0;
    int ignore = 1;
    int i, j;
    int column_check = input_num + 3;
    int new_input[30];
    int old_input[30];

    [ ]+ ++field_count;

\n  (
    if (ignore == 0) {
        printf("/n");
        ECHO;
    }
    ignore = 1;
    field_count = 1;
    flag = 1;
)

(0-1)+ {
    if (field_count < column_check) {
        if (flag == 0) {
            old_input[field_count] = atoi(yytext);
            new_input[field_count] = atoi(yytext);
        }
        if (flag == 1)
            new_input[field_count] = atoi(yytext);
    }
    if (field_count == column_check) {
        for (i=3; i<column_check; i++) {
            if (new_input[i] == old_input[i]) {
                ignore = 1;
            }
            else {
                ignore = 0;
                printf("/n");
                for (j=3; j<column_check; j++) {
                    printf("%d", old_input[j]);
                    old_input[j] = new_input[j];
                }
            }
            if (atoi(yytext) == 0) {
                printf(" L");
                break;
            }
            else {
                printf(" H");
                break;
            }
        }
    }
    else /*if (field_count > column_check)*/ {
        if (ignore == 0) {
            if (atoi(yytext) == 0) {
                printf(" L");
            }
            else {
                printf(" H");
            }
        }
        else {
            ;
        }
    }
}

(pin) ;
(x)+ ;
TIME ;
{"^ \n}+ ;

```

Figure 30 "vector_map.1" Code

values 0 or 1. The following code shows a portion of the `vector_map.l` routine which satisfies this translation requirement to change the output vector elements to a "H" or "L":

```
if (atoi(yytext) == 0) {  
    printf (" L");  
    break;  
}
```

The second translation requirement listed in Table XXV poses the most challenging programming algorithm. A close observation of the `74S181.list` file in Figure 16 reveals several output vector states for every set of input vector stimuli. These output vector elements change state until the end of a delay time is encountered. The delay times for the 74S181 ALU chip in the simulator environment lasts approximately 6-7 nanoseconds. Although this intermittent state change information is interesting, it would confuse the GR-125 tester. The test vector elements placed in the GR-125 .tpp file require a set of input stimulus elements followed by expected output result elements. If every test vector in the QuickSim .list file were put into the GR-125, the functional test would always produce a failed result. Accordingly, only the initial input stimulus elements followed by the final state output elements are chosen to form a valid test vector.

By using various "arrays" of input elements and "for" loops, this lex routine provides a look ahead capability to determine which set of input and output elements to record for an accurate test vector.

C. PROGRAM USAGE

The main program and lex routine must be compiled and linked to form a usable translator program. Table XXVI reviews these required steps. This section will address the usage of the newly developed translator program, vector_map.

Table XXVI "vector_map" COMPILATION STEPS

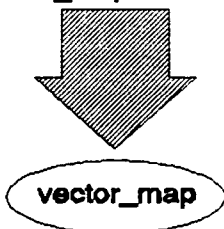
step 1 : Write lex specification → vector_map.l

step 2 : Run lex:

prompt > *lex vector_map.l*

step 3 : Compile and link with main program (vector_map.c)

prompt > *cc -o vector_map vector_map.c lex.yy.c -ll*



1. Input File

As stated previously, the input file used by the vector_map translator is composed of the List Window file produced from the QuickSim simulator. This .list file must

have the input elements listed before the output elements. This requirement is easy to obtain since the programmer can order the list window in any desired way within the QuickSim environment. Secondly, the actual number of input pins must be known prior to invoking the translator program. The next section will discuss the placement of this input number.

2. Command Line Entry

The command line entry required to obtain a valid test_vector_out file is illustrated in Table XXVII. Notice that this entry uses the UNIX tools cat and pipe "|" to funnel the input file through the translator to produce the valid output file. Additionally, the number immediately following vector_map is the required location for inputting the number of input pins. In this example, the number 14 represents the 14 input pins in the 74S181 ALU chip.

Table XXVII "vector_map" COMMAND LINE ENTRY

prompt > cat input_file.list | vector_map 14 > test_vector_out

D. RESULTS

The output file created from the command line entry described above, contains input and output pin states in proper test vector pattern format. This .tpp file format was

discussed back in Figure 9 of chapter II. Figure 31 illustrates the output file produced by invoking the vector_map translation program on the 74S181.list simulation file developed in chapter III.

The final action required to fully link the CAD simulator and hardware tester environments is to include this newly created output file into the GR-125 .tpp file. Chapter II referenced the use of the "INCLUDE" statement in the PATTERN section of the .tpp file. Only one line is added to the existing .tpp file to incorporate this newly created output file. The additional line of code is placed in the PATTERN section of the GR-125 .tpp file. Table XXVIII illustrates the proper line of code required to incorporate this 74S181.v_out file into a GR-125 .tpp file.

Table XXVIII INCLUDE STATEMENT FOR GR-125 .tpp
FILE

INCLUDE "74S181.v_out"

The vector_map translator program, developed in this chapter, has produced an extremely useful tool for hardware testing using the GR-125. Highly accurate test vector data, produced in the computer simulation environment, can now be directly placed into the GR-125 .tpp file without the need for

time consuming test vector edits or rewrites. Additionally, the chance for errors occurring within these test vectors also decreases significantly. As a result of these characteristics, this translator program has successfully solved the incompatibility problem between the digital design and hardware test environments.

```

/01100010000100 HLLLLLLL/
/01100011000001 HLLLLLLL/
/01100010100110 HLLLLLLL/
/01100011100001 HLLLLLLL/
/01100010010101 HLLLLLLL/
/01100011010011 HLLLLLLL/
/01100010110111 HLLLLLLL/
/01100010001100 LLLLLLLL/
/01100010101110 LLLLLLLL/
/01100010011101 LLLLLLLL/
/01100010111111 LLLLLLLL/
/01100010000010 HLLLLLLL/
/01100011001110 LLLLLLLL/
/01100011101011 HLLLLLLL/
/01100010001010 LLLLLLLL/
/01100011001001 HLLLLLLL/
/01100110000100 HLLLLLLL/
/01100111000001 HLLLLLLL/
/01100110100110 HLLLLLLL/
/01100111100001 HLLLLLLL/
/01100110010101 HLLLLLLL/
/01100111010011 HLLLLLLL/
/01100110110111 HLLLLLLL/
/01100110001100 LLLLLLLL/
/01100110101110 LLLLLLLL/
/01100110011101 LLLLLLLL/
/01100110111111 LLLLLLLL/
/01100110000010 HLLLLLLL/
/01100111001110 LLLLLLLL/
/01100111101011 HLLLLLLL/
/01100110001010 LLLLLLLL/
/01100111001001 HLLLLLLL/
/10010010000100 LLLLLLLL/
/10010011000001 LLLLLLLL/
/10010010100110 HLLLLLLL/
/10010011100001 LLLLLLLL/
/10010010010101 HLLLLLLL/
/10010011010011 HLLLLLLL/

```

```

• •
• •
• •

```

Figure 31 "74S181.v_out" File

VI. CONCLUSIONS

A. SUMMARY OF RESEARCH

This thesis analyzed the digital testing process within two separate test environments. Chapter II focused on the hardware test environment by performing a detailed system description of the GENRAD (GR-125) Hardware Tester System. Therefore, chapter II accomplished the first major thesis objective. This description provided a detailed overview of the methodology required to successfully program and execute a GR-125 digital logic test. Additionally, the specific testing capabilities of the GR-125 were also evaluated. Next, chapter III described the digital test process inside the computer simulation (i.e. software) environment. The Mentor Graphics IDEA Series (v 7.0) QuickSim simulator provided the platform to analyze this digital design and test process within this environment. The emphasis of the discussion is centered on the test vector stimulation/response information format produced in the QuickSim simulator output file. This output file contains all of the basic information required by the test pattern portion of the GR-125 .tpp input file. However, the structure formats of the two environments are quite different and ,therefore, incompatible.

Chapter IV provided a general methodology of using special UNIX tools, lex and yacc, to produce a successful computer language translation. This discussion gave the necessary background information required to solve the specific test vector format incompatibility problem between the QuickSim simulator output file and the GR-125 .tpp input file. Chapter V provided the actual solution to the second major thesis objective. This chapter presents the actual code used to translate the test vector stimulus/response information in the simulator output file into the format required by the GR-125 .tpp input file. Therefore, a successful link between the software and hardware test environments has been accomplished.

B. RECOMMENDATIONS FOR FURTHER RESEARCH

This thesis discussed two extremely powerful programs in the standard UNIX toolkit (lex & yacc). These programs provide an extremely useful mechanism for translating from one computer language to another. Although lex by itself provided an adequate capability to build the successful translator program, vector_map, developed in chapter V, a host of increasingly more difficult translator applications are possible with lex and yacc.

The test vector translation program developed in this thesis successfully linked a computer simulator output with a hardware tester input. However, both the computer simulator (QuickSim) and the hardware tester (GR-125) are stand alone

systems. This situation requires a new translation program to be developed whenever a different simulator or ATE is used. Figure 32 illustrates this dilemma. Fortunately, a new IEEE standard is forcing more standardization. This standard is known as WAVES (Waveform And Vector Exchange Specification). Basically, this standard will force all of the CAD simulation programs to produce a standard test vector format. Once this standard is fully implemented industry wide, only one simulator file format will be produced. As a result, the number of different translation programs required decreases drastically. Figure 33 illustrates this scenario. Now, all of the different hardware testers would use a common input test vector file format. Conforming with WAVES development of more generic test vector translation applications would be extremely advantageous.

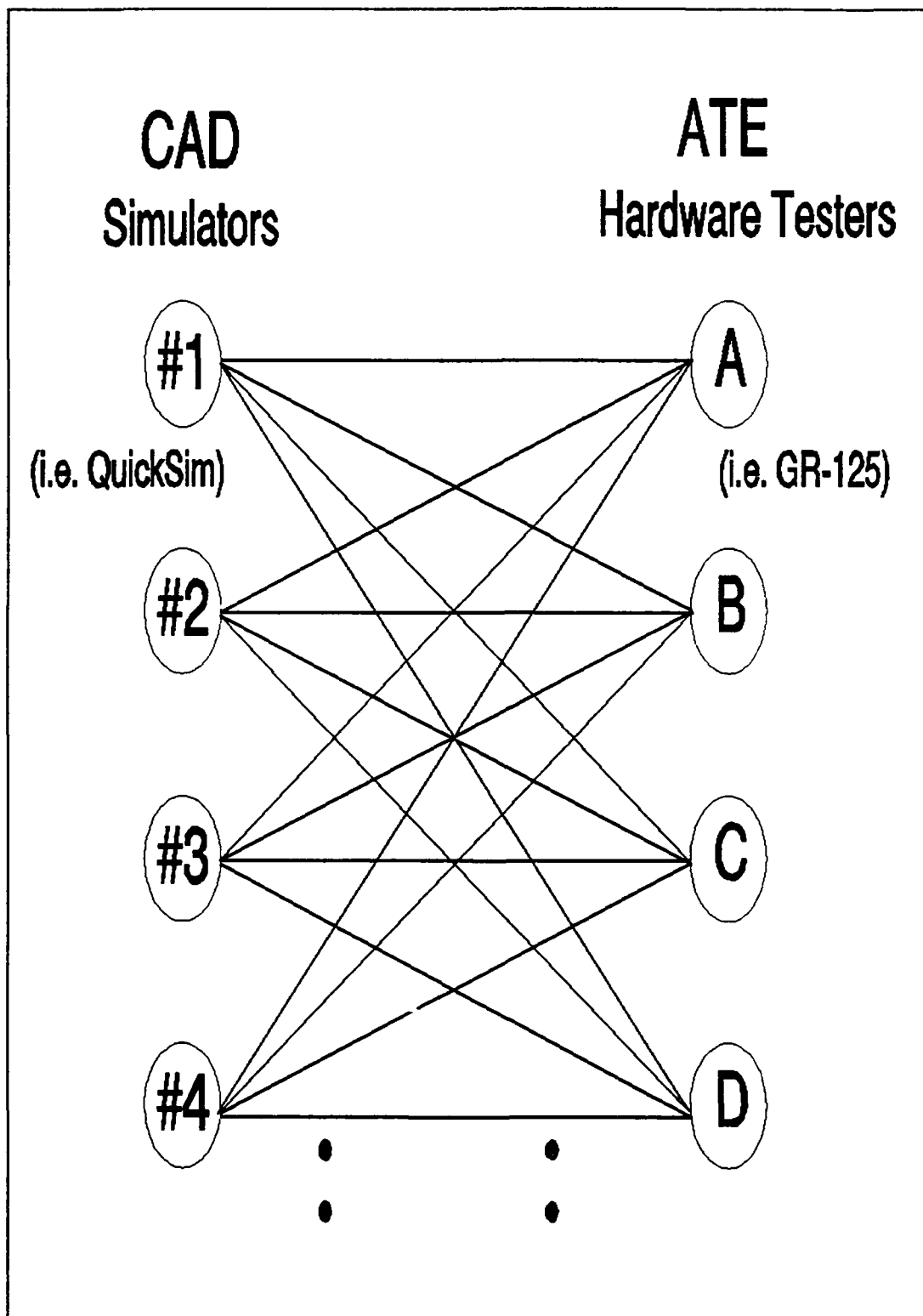


Figure 32 Translation Summary Without WAVES

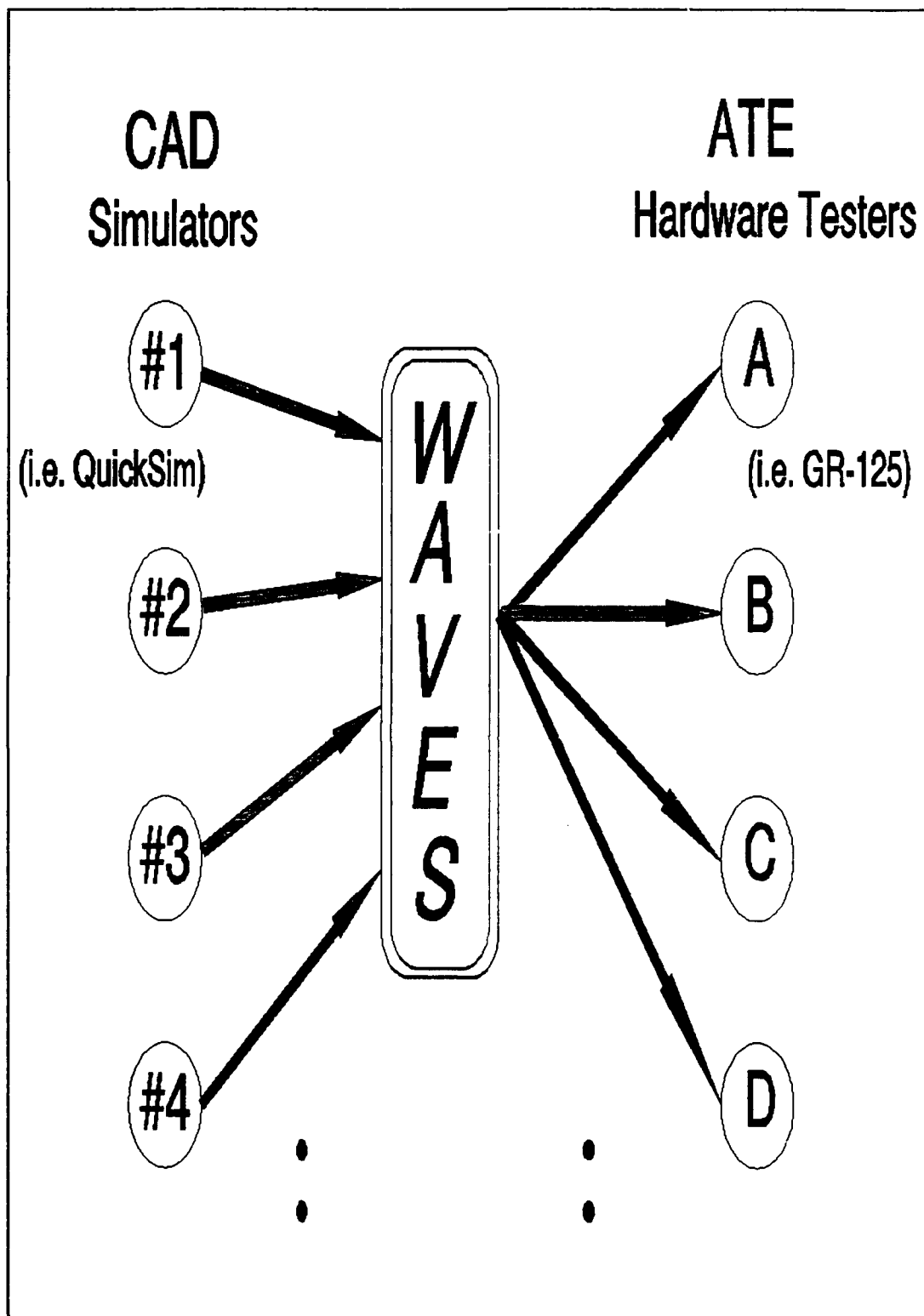


Figure 33 Translation Summary With WAVES

LIST OF REFERENCES

1. Fujiwara, Hideo, *Logic Testing and Design for Testability*, MIT Press, 1985.
2. Bennetts, R. G., *Design of testable logic circuits*, Addison-Wesley Publishers Limited, 1984.
3. *GR125 Manager's Guide*, v. 2.3, GenRad, Inc., Concord, Massachusetts, 1990.
4. *GR125 Operator's Guide*, v. 2.3, GenRad, Inc., Concord, Massachusetts, 1990.
5. *GR125 Programmer's Guide*, v. 2.3, GenRad, Inc., Concord, Massachusetts, 1990.
6. *IDEA Series Schematic Capture User's Manual*, v. 7.0, Mentor Graphics, Corp., Beaverton, Oregon, 1989.
7. *QuickSim User's Manual*, v. 7.0, Mentor Graphics, Corp., Beaverton, Oregon, 1989.
8. *1984 LOGIC DATA BOOK*, vol II, National Semiconductor Corp., Santa Clara, California, 1984.
9. Mason, T., and Brown, D., *lex & yacc*, O'Reilly and Associates, Inc., 1990.

INITIAL DISTRIBUTION LIST

- | | | |
|----|--|---|
| 1. | Defense Technical Information Center
Cameron Station
Alexandria, VA 22304-6145 | 2 |
| 2. | Library, Code 52
Naval Postgraduate School
Monterey, CA 93943-5002 | 2 |
| 3. | Chairman Code EC
Department of Electrical and Computer Engineering
Naval Postgraduate School
Monterey, CA 93943-5000 | 1 |
| 4. | Professor Chin-Hwa Lee, Code EC/Le
Department of Electrical and Computer Engineering
Naval Postgraduate School
Monterey, CA 93943-5000 | 4 |
| 5. | Professor Herschel Loomis, Code EC/Lm
Department of Electrical and Computer Engineering
Naval Postgraduate School
Monterey, CA 93943-5000 | 1 |
| 6. | Naval Maritime Intelligence Center
ATTN: Mr. Charles Bradley
Building 75
5803 Bayside Road
Chesapeake Beach, MD 20730 | 1 |
| 7. | Naval Maritime Intelligence Center
ATTN: Mrs. Janet Hooper
Building 75
5808 Bayside Road
Chesapeake Beach, MD 20730 | 1 |
| 8. | LT James T. Loeblein
410 Brownrigg Road
Salisbury, NC 28144 | 2 |
| 9. | STEWS NE-A
ATTN: John Sweeney
Building 21225
White Sands Missile Range
White Sands, NM 88002 | 1 |