RESEARCH TRIANGLE INSTITUTE

SOFTWARE SYSTEM USER'S MANUAL, REFERENCE MANUAL, AND INSTALLATION GUIDE FOR THE TEST ENGINEER'S ASSISTANT SYSTEM

February 28, 1989

Contract No. DAAL01-86-C-0039

Prepared for:

Department of the Army
Electronics Research and Development Command
Fort Monmouth, New Jersey 07703

Prepared by:

Center for Digital Systems Research
Research Triangle Institute
Research Triangle Park, NC 27709

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	February 28, 1989	Software Users Manual

4. TITLE AND SUBTITLE
Software System User's Manual, Reference Manual, and Installation Guide for the Test Engineer's Assistant System

5. FUNDING NUMBERS

6. AUTHOR(S)
Center for Digital Systems Research
Research Triangle Institute
Research Triangle Park, NC 27709

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
SAME as above

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
ETDL, LABCOM
Circuits & Subsystems Design Branch
Fort Monmouth, NJ 07703-5601

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
DAAL01-86-C-0039

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for Public Release; Distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

The Test Engineer's Assistant (TEA) is a set of software tools and utilities that help a digital hardware designer design testable systems. These software tools and utilites have been integrated into the Architecture Design and Assessment System (ADAS). TEA provides a design automation system for the incorporation of design for testa nd built-in test techniques into VHSIC digital hardware designed using ADAS and the VHSIC Hardware Description Language (VHDL).

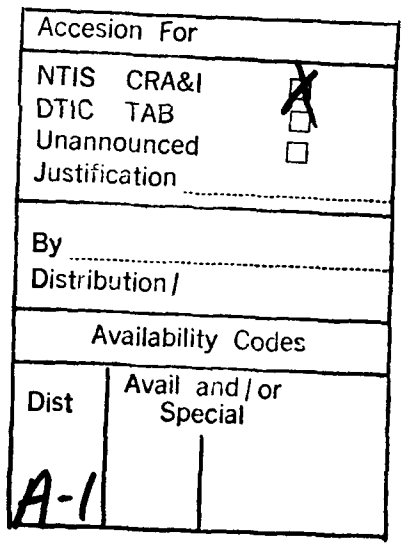
14. SUBJECT TERMS
VHDL, Built in Test, Digital Electronics, ADAS design for test

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLAS	UNCLAS	UNCLAS	

SOFTWARE SYSTEM USER'S MANUAL FOR THE TEST ENGINEER'S ASSISTANT SYSTEM



Software System User's Manual
for the Test Engineer's Assistant System

TABLE OF CONTENTS

LIST OF FIGURES

1. SCOPE

1.1. Identification. This Software System User's Manual (SSUM) establishes the operating and user procedures for the computer software configuration item (CSCI) identified as the *Integration of Very High Speed Integrated Circuit (VHSIC) System Functional Design and Design for Testability* tools, to be known as the Test Engineer's Assistant (TEA) system.

1.2. Purpose. The Architecture Design and Assessment System (ADAS®) provides a graphic schematic capture capability to VHSIC designers. TEA provides a design automation system for the incorporation of design for test (DFT) and built-in test (BIT) techniques into VHSIC digital hardware which is described by using ADAS and VHSIC Hardware Description Language (VHDL). TEA provides the methodology and the tools for evaluating proposed designs for hard-to-test constructs, for recommending alternative constructs, and for identifying hierarchical BIT techniques to increase the overall testability of a hardware system. TEA is designed to aid users at the board, subsystem, and system levels. TEA estimates the costs associated with the BIT techniques to allow the designer to make informed choices about BIT incorporation. A library of reusable BIT modules supports TEA and enables the designer to simulate the testable system in VHDL. ADAS provides the interface to VHDL from the graphic connectivity schematic.

This document should be used in conjunction with the *Reference Manual for the Test Engineer's Assistant System.* The reference manual will help the user prepare for use of the TEA tools; this user's manual will aid in the mechanics of the use of the tools; the reference manual will be useful in the analysis of the outputs of the tools and in identifying the direction that the user should take depending on that output.

The main purpose of this document is to present TEA from the user's point of view.

1.3. Introduction. This SSUM specifies the computer operation procedures for the VAXstation II/GPX (Section 3) and the user specifics for the TEA system (Section 4).

ADAS is a registered trademark of Research Triangle Institute.

2. REFERENCED DOCUMENTS

The following documents are considered useful to the user of TEA:

- *TEA Reference Manual*
- *ADAS User Manual* - Version 2.3 or later
- *TEA Installation Guide - Version PROTO.TYPE/Contract*
- *Software User's Manual for the VHSIC Silicon Compiler System*
- *Software User's Manual for the ADAS/VHDL System*

These documents are available by contacting the Research Triangle Institute (RTI) ADAS marketing coordinator at

Research Triangle Institute
Center for Digital Systems Research
P. O. Box 12194
Research Triangle Park, NC 27709

Phone: (919) 541-7436

- *MicroVMS Workstation User's Guide*, May 1986
- *MicroVMS Workstation Software Installation Guide*, November 1986
- *VAXstation II/GPX Hardware Information Manual*, 1986

These documents are available by writing to

SSG Publications ZK1-3/J35
Digital Equipment Corporation
110 Spit Brook Road
Nashua, NH 03062-2698

3. COMPUTER SYSTEM OPERATION

Separate installation instructions are provided with the TEA distribution tape.

This section will provide pointers to the documentation provided with the VAXstation II/GPX (TEA workstation) by Digital Equipment Corporation rather than try to rewrite this information. Information concerning computer system preparation is found in the hardware manual. Operating procedures are found in the user's guide, and software installation and testing procedures are found in the software installation guide. The following paragraphs will summarize the information found within these three information sources.

The *Digital Equipment Corporation's VAXstation II/GPX Hardware Information Manual* (dated 1986) is divided into six parts:

Part I: Base System Installation — describes how to unpack, install, and test the VAXstation II/GPX.

Part II: Operation — describes the operation of the VAXstation II/GPX.

Part III: Option Installation — describes the hardware options for the VAXstation II/GPX and gives installation information where applicable.

Part IV: Troubleshooting — describes how to isolate a problem and decide what to do next.

Part V: Appendixes — provides VAXstation II/GPX system specifications. Appendix B lists related documents.

Part VI: Glossary — defines computer terms that are italicized at first use in the text as well as other common computer terms.

The *Digital Equipment Corporation's MicroVMS Workstation User's Guide* (dated May 1986) is divided into three chapters and four appendices:

Chapter 1: Getting Started — provides an overview of the workstation functions and introduces some of the terminology used in the guide.

Chapter 2: The Workstation Options Menu — describes the functions of the Workstation Options menu.

Chapter 3: Using Windows — explains window-management tasks such as creating, moving, and resizing windows.

Appendix A: VAXstation Support for Tektronix 4010/4014 Terminals — describes VAXstation support for Tektronix 4010/4014 terminal emulation. TEA users will not specifically need this information.

Appendix B: Additional Features of the VT220 Terminal Emulator — provides additional setup information for VT220 terminal emulation.

Appendix C: Compose Sequences — provides information on compose-character sequences, including tables for both the multinational and the national character sets. TEA users will not specifically need this information.

Appendix D: National Keyboards for VT220 Emulation — illustrates the keyboard character configurations for the various languages supported by the VAXstation. TEA users will not specifically need this information.

The *Digital Equipment Corporation's MicroVMS Workstation Software Installation Guide* (dated November 1986) is divided into three chapters and one appendix:

Chapter 1: Installing on a VAXstation I, VAXstation II, or VAXstation II/GPX — describes how to install workstation software on the VAXstations.

Chapter 2: Installing on a Local Area VAXcluster — describes how to install workstation software on the VAXcluster.

Chapter 3: VAXstation Management and Tuning — provides information on tuning the workstation software.

Appendix A: Files Installed by MicroVMS Workstation Software — lists files installed by the workstation software.

4. INSTRUCTIONS FOR USE — TEA

This section has been labeled as Chapter Z — TEA: The Test Engineer's Assistant and could be added as a chapter in the user's ADAS documentation.

CHAPTER Z — TEA: THE TEST ENGINEER'S ASSISTANT

INTRODUCTION

The Test Engineer's Assistant (TEA) is a set of software tools and utilities that help a digital hardware designer design testable systems. These software tools and utilities have been integrated into the ADAS user interface and have been combined with several existing ADAS functions to provide all that is necessary to accomplish the goal.

ASSUMPTIONS ABOUT USER KNOWLEDGE

TEA makes some assumptions about the user's knowledge. The tool is designed to be used by a system design team with little or no testability experience. Such a team might include system specialists and/or hardware systems designers. Thus, a familiarity with hardware design terminology is assumed.

Although TEA operates on the assumption that the user is not a testability engineer, it does assume that the basic terminology of testability is known. The following list describes some of the basic testability terms with which the user must be familiar. Consult the *TEA Reference Manual* for a complete description of these and other testing terms.

a. ambiguity group (AG),
b. built-in test (BIT),
c. BIT techniques,
d. design for testability (DFT),
e. deterministic and pseudorandom test patterns,
f. mean time to test,
g. test pattern application, and
h. test pattern generation.

The designer must have experience using ADAS to capture a design.

TEA RESTRICTIONS AND LIMITATIONS

- TEA only executes on a non-empty ADAS hardware graph data base.
- Window scaling minimum of 6 locations by 6 locations; maximum of 600 locations by 600 locations.
- Maximum of 100 user-defined macros active at once during a TEA session.
- Several colors are available for graph nodes and arcs. The *ADAS User Manual*, Version 2.3 or later, lists the colors in the ADAS palette.

TEA MENU SYSTEM

The TEA menu system retains the basic format of the ADAS graph editor, EDIGRAF. An additional option, however, has been added to better suit the interactive tools in TEA. The TEA menu system allows one of two types of selections to be made from a menu:

a. single item selection or

b. multiple item selection.

Single Selection Menus

A single select menu is identical to the ADAS EDIGRAF menu selection. One entry selection is allowed per menu. Once the selection has been made, the graph editor automatically cycles to the next level in the menu system or invokes the required tool functions.

Multiple Selection Menus

A multiple select menu allows the user to select multiple entries in a menu. Each selected item in the menu is highlighted to distinguish between selected and unselected items. A multiple select menu is identified by the presence of the **%done** option which is always located at the top of a multiple select menu. The menu remains displayed until the user has selected the **%done** option. Once an item has been selected on the menu, the item may be selected again to unselect that item from the current menu. In some multiple select menus, an **%all** option is available which automatically selects all menu items in the current menu. The **%all** option requires that **%done** be selected to indicate that all highlighted menu items are to be used as input. The user is always reprompted after making a selection (other than **%done**) from a multiple select menu.

Figures 1 and 2 show typical single select and multiple select menus used in the TEA tools.

```
=== menu ===
%help
%self
file1.hwg
file2.hwg
============
Compare: Select the comparison graph:
```

Figure 1. Typical Single Select Menu

```
=== menu ===
%done           board3          board8
%help           board4          board9
%all            board5
board1          board6
board2          board7
============
DFT: Indicate the board(s) of interest:
```

Figure 2. Typical Multiple Select Menu

TEA TOP-LEVEL MENU

The top-level menu contains menu options to access all the TEA tools and utilities as well as the ADAS functions available in TEA. Figure 3 shows the terminal display for the TEA top-level menu.

```
=== menu ===
quit        brt          log_file     hardcopy
save        bit_cost     window       script
edit        placebit     environ      macro
move        compare      stats        reload
dft         ag_name      subgraf      help
============
Command%
```

Figure 3. TEA Top-Level Menu

ADAS Functions

a. **edit** — modify graph, node, arc, bus, and template attributes,
b. **environ** — change the display parameters for the graphics window,
c. **hardcopy** — generate a plot of the graphics display on a hardcopy device,
d. **macro** — a single name representing commonly used command sequences,
e. **move** — relocate an arc, bus junction, node, or block of graph components,
f. **quit** — exit TEA,
g. **reload** — reload the design graph from the data base files,
h. **save** — save data base to disk,
i. **script** — execute a predefined set of menu option commands,
j. **stats** — display individual node, arc and bus statistics,
k. **subgraf** — expand the subgraf associated with a particular node, and
l. **window** — zoom in and out on the current graphics display.

TEA Tools

a. **dft** — Design for Testability Guideline Checker,
b. **brt** — BIT Recommendation Tool,
c. **bit_cost** — BIT Overhead Summary,
d. **placebit** — BIT Placement Recommendation, and
e. **compare** — System Summary.

TEA Utilities

a. **ag_name** — assign ambiguity groups,
b. **log_file** — view and print report files, and
c. **help** — access on-line user support system.

Special Characters

a. **?help** — short description of the menu commands,
b. **.cancel** — skip the command being entered without executing it and return to the top-level menu,
c. **!escape** — exit temporarily to the operating system without leaving TEA, and
d. ***page** — circulate to the next menu page if more items exist.

ADAS FUNCTIONS USED IN TEA

Existing ADAS Functions

TEA utilizes several ADAS graph editor (EDIGRAF) functions in the TEA top-level menu. The menu option names as well as their functions are identical to the ADAS options. The unchanged ADAS functions which are utilized by TEA are:

a. **edit,**
b. **environ,**
c. **hardcopy,**
d. **macro,**
e. **move,**
f. **quit,**
g. **reload,**
h. **script,**
i. **stats,**
j. **subgraf,** and
k. **window.**

A detailed description of these options is provided in Chapter 7 of this document.

Modified ADAS Functions

The **save** ADAS function has been modified to accommodate TEA-specific operations; however, the ADAS option name has been retained.

Save

The **save** function provides a method for the user to write changes made to the "current view" data base files to disk. The TEA **save** function differs from the ADAS function by allowing the user to save a separate copy of the current graph and associated subgraphs under a new filename. By retaining previous versions of the hardware graphs, multiple combinations may be analyzed at a later time. TEA will also allow overwriting of the existing file. The **save** function is invoked by selecting the **save** option from the TEA top-level menu. Figure 4 shows the terminal display after **save** has been selected.

```
=== menu ===
new_copy
overwrite
============
SAVE: Save a copy or overwrite the current graph?
```

Figure 4. save Function Menu

Generating a Copy

A copy of the "current view" hardware graph data base can be made by selecting the **new_copy** option. A **new_copy** is a copy of the "current view" hardware graph stored to a new file on disk. A copy can be made at anytime while at the TEA top-level menu. There are two types of copies that can be made when using the **new_copy** option:

a. **current** — for generating a copy of the currently displayed graph only and

b. **hierarchy** — for generating a copy of the hierarchy from the current graph up to and including the root level graph.

Figure 5 shows the terminal display for the **save new_copy** menu.

```
=== menu ===
current
hierarchy
============
SAVE NEW_COPY: Current graph or hierarchy as well?
SAVE NEW_COPY: Indicate a filename for the copied graph.
```

Figure 5. save new_copy Menu

Filename Selection

If **save new_copy current** has been selected, the user is given the option of indicating the new filename of the copied file. Any valid VMS filename is accepted. If the name of an existing file is used, a message indicating the existance of that file is displayed along with a prompt to enter a unique filename. The user is continually reprompted until a unique filename is given or the user escapes from the menu by typing ".". Figure 5 also shows the prompt for the new filename. At the completion of the copy, program control is returned to the TEA top-level menu.

If **save new_copy hierarchy** has been selected, the names for the new graphs are automatically generated. The new filenames consist of the original filename with a two digit number appended. The two digit number is incremented to generate a unique new filename each time a copy is made. Hardware graphs along with the hardware graph templates are copied. After all graphs in the hierarchy have been copied, program control is returned to the TEA top-level menu.

Overwriting Existing Graph(s)

The **overwrite** option replaces the existing file(s) on disk. There are three options available for overwriting:

a. **current** — for overwriting the currently displayed graph,

b. **hierarchy** — for overwriting the hierarchy from the current graph up to the root level graph, and

c. **full** — for overwriting all graphs and subgraphs connected to the root level graph.

The **current** and **hierarchy** options are identical to the ADAS options, however; the **full** option has been added to TEA to allow copies of complete system graphs for later examination and comparison. Figure 6 shows the terminal display for the **save overwrite** menu.

```
=== menu ===
current
hierarchy
full
============
SAVE OVERWRITE: Overwrite the current graph, hierarchy or all graphs ?
```

Figure 6. save overwrite Menu

TEA TOOLS AND UTILITIES

The following sections describe the operation of TEA. TEA supports the following specific tools:

a. Design for Testablity Guideline Checker (**dft**) — identifies hard-to-test and untestable structures and recommends alternative structures,

b. BIT Recommendation (**brt**) — divides a board into ambiguity groups for fault isolation testing and recommends a class of BIT techniques for each ambiguity group,

c. BIT Overhead Summary (**bit_cost**) — calculates the approximate hardware overhead associated with a particular BIT technique,

d. BIT Placement Recommendation (**placebit**) — generates a new schematic of the board with a specific implementation of the given technique, and

e. System Summary (**compare**) — itemizes the incremental hardware overhead attributed to added testability.

TEA supports the following specific utilities:

a. Ambiguity Group Names (**ag_name**) — identifies user-selected ambiguity groupings,

b. Report Access/Generation Facility (**log_file**) — generates and provides view capability of output reports of the various tools, and

c. On-line User Support System (**help**) — information to support user's understanding of TEA. This function is called **help** on the TEA top-level menu.

The above functions and utilities are detailed in the paragraphs that follow. TEA is an EDIGRAF-like tool in that it uses the ADAS user interface, data base, and graphics. It is invoked by the following command:

```
tea file.hwg [-d graphics_device -s script_file]
```

The graphics device may be specified in an environment variable and may include *null*. It must be initialized for TEA to execute. The script file is optional and works the same as ADAS scripts (see Chapter 4 for further information about scripts and graphics devices). TEA only executes on a non-empty ADAS hardware graph data base.

The user interface is menu-driven, allowing the user to make selections with the terminal keyboard or to pick from the graphics menu or graph itself by using a mouse or data pad. Chapter 4 describes the ADAS user interface in greater detail. The following paragraphs will describe the TEA menu hierarchy as it pertains to individual commands on the top-level menu that are unique to this tool. Figure 7 shows the terminal display for the TEA top-level menu.

```
=== menu ===
quit          brt           log_file      hardcopy
save          bit_cost      window        script
edit          placebit      environ       macro
move          compare       stats         reload
dft           ag_name       subgraf       help
============
Command%
```

Figure 7. TEA Top-Level Menu

Design For Testability Guideline Checker

The Design for Testability Guideline Checker (**dft**) is used to identify hard-to-test and untestable structures in a digital board design. The design topology and functions are compared to predefined guidelines. Any violations are reported to the user. An ADAS hardware graph description is used as input to the **dft** tool. The connectivity of the hardware design as well as specific attributes in the ADAS data base are used to determine if any violations exist. The **dft** tool includes options for graphically identifying several types of structures as well as an option which explains the various standard TEA guidelines. The **dft** tool is invoked from the TEA top-level menu by selecting the **dft** option.

The Design for Testability Guideline Checker has three subfunctions:

a. **analyze** — for isolating hard-to-test and untestable constructs in a design at the board level and recommending alternative structures,

b. **identify** — for graphically identifying structures in a design, and

c. **explain** — for on-line descriptions of predefined guidelines.

Each subfunction may be invoked by selecting the appropriate command from the **dft** Options single select menu. Figure 8 shows the terminal display for the **dft** options menu.

```
=== menu ===
%help
analyze
identify
explain
============
DFT: Select a DFT option to be executed:
```

Figure 8. dft Options Menu

The **analyze** command is the main **dft** function which isolates predefined types of untestable or hard-to-test structures at the board level. In addition, recommendations are given on how to make the design more testable when a violation has been detected. The **analyze** command invokes a menu representing the types of guidelines the user may choose to invoke on a particular board or set of boards. There are two categories of guidelines from which to select:

a. standard TEA guidelines and
b. user-defined guidelines.

Either category may be invoked by selecting the corresponding option from the guideline menu. Figure 9 shows the terminal display for the **dft** guideline menu.

```
=== menu ===
%help
standard
user_rules
============
DFT: Select the type of rules to use:
```

Figure 9. dft Guideline Menu

Standard TEA Guidelines

If the **standard** TEA guidelines option is selected, a list of predefined guidelines appear on the menu. The standard TEA guidelines represent all the guidelines provided as part of the TEA **dft analyze** guideline data base. These guidelines have been divided into two basic groups:

a. aid test pattern generation and
b. aid test pattern application and fault isolation.

The user may select any number of these guidelines to be executed on a specific board. There are no restrictions on invoking test pattern generation or test pattern application guidelines simultaneously on the same board. A typical case would be to invoke

all the test pattern generation guidelines, either individually or in groups, and then proceed to the test pattern application guidelines. Figure 10 shows the terminal display for the **dft** select guildeline menu.

```
=== menu ===
%done          g1_02          g1_08          g1_14
%help          g1_03          g1_09          g1_15
%all           g1_04          g1_10          g1_16
%all_g1        g1_05          g1_11          g2_01
%all_g2        g1_06          g1_12
g1_01          g1_07          g1_13
==========
Enter '*' to view additional menu items
DFT: Select the guideline(s) to be executed:
```

Figure 10. dft Select Guideline Menu

The user has several options for invoking specific guidelines. The most basic option is to select a particular guideline from the menu. Since this is a multiple selection menu, any number of guidelines may be selected. If a guideline has been selected it will be highlighted on the menu display. Any selected guideline may be deselected by selecting the highlighted guideline from the menu again.

To simplify the selection of multiple guidelines, three grouping options have been incorporated into the menu selection. All of the aid-to-test-pattern generation guidelines may be selected with one menu option, **%all_g1**. This single selection will highlight all of the aid-to-test-pattern generation guidelines. All of the aid-to-test-pattern application guidelines may be selected in a similar fashion via the **%all_g2** menu option. The **%all** option indicates that all of the aid-to-test-pattern generation guidelines as well as all of the aid-to-test-pattern application guidelines are to be invoked. Selecting **%all** from the menu will highlight all guidelines in the menu. These grouping options are particularly useful when most guidelines of a particular type are to be invoked. Rather than selecting each guideline individually, a grouping selection may be made followed by a deselection of individual guidelines to produce the desired list.

Since this is a multiple selection menu, the **%done** option must be selected to indicate that the currently highlighted guidelines are the ones to be invoked. The **%done** option also invokes the next menu which isolates the particular subsystem or subsystems to which the guidelines will be invoked.

Subsystem Selection

Since the TEA guidelines operate at the board level, the user is only given the option to chose boards to be evaluated with the selected guidelines. To specify a particular board, the subsystem must first be identified. The subsystem menu contains all the valid subsystems for the current graph hierarchy. When a particular subsystem is selected, all the boards from that subsystem will then be displayed on the next menu. The **%done** option invokes the next menu which is a list of all the boards from the previously selected subsystems. Figure 11 shows the terminal display for the **dft** select subsystem menu.

```
=== menu ===
%done
%help              subsys2
%all               subsys3
subsys1            subsys4
============
DFT: Indicate the subsystem(s) of interest:
```

Figure 11. dft Subsystem Select Menu

Board Selection

Only those boards which reside on the previously selected subsystem(s) are included in the board menu. Any combination of boards may be selected from the menu. A grouping option **%all** has been included to ease the selection of all the boards or a large portion of the boards. Once the **%done** option has been selected, the **dft analyze** tool is automatically invoked. Figure 12 shows the terminal display for the **dft** select board menu.

```
=== menu ===
%done          board3         board8         board13
%help          board4         board9         board14
%all           board5         board10        board15
board1         board6         board11        board16
board2         board7         board12
============
DFT: Indicate the board(s) of interest:
```

Figure 12. dft Select Board Menu

The user-selected boards are analyzed using the user-selected guidelines. The analysis is done in Prolog; therefore, the graph must be converted to a format that can be used by the Prolog guidelines. This process takes a few minutes. Figure 13 shows the terminal display for a typical **dft analyze** execution output. The boards are treated independently.

```
[compiling project:[tea]graph.pl...]
[graph.pl compiled 236.780 sec 78,996 bytes]
[compiling project:[tea]control.pl...]
[control.pl compiled 1.950 sec 632 bytes]
Run analyze rules
running g1_01
```

Figure 13. dft analyze Execution Output

The output lines indicate that the ADAS graph is being converted to Prolog format and is then compiled. If a graph has been compiled once and is not modified, it will not recompile during the current session (e.g., a guideline executed on board1 can be executed on board2 without having to wait for the graph to be recompiled). The subsystem and board selections are also converted to Prolog format and compiled. The currently executing guideline is listed on the output display.

Report Selection

Upon completion of the selected guidelines, the user is prompted for a filename in which to store a report of the results for later examination. A default filename will be provided if the user chooses not to name the file. The default name used in **dft analyze** is dft_01.rpt. The two digit number is included to provide a unique filename if multiple runs are made using the default filename. If the user types `nl:` at the prompt, no output file is created. If the user selects an existing filename, a new version of the file is created. Program control is then returned to the TEA top-level menu. Figure 14 shows the terminal display for the **dft analyze** report prompt.

```
Enter <cr> or YES to accept dft_01.rpt as the default filename
Or enter desired output filename :
```

Figure 14. dft analyze Report Prompt

User-defined Guidelines

The user-defined guidelines represent those guidelines which the user has specifically generated. TEA does not provide any user-defined guidelines. The user-defined guidelines work in the same fashion as the standard ones; however, they are not divided into any specific categories. Standard TEA guidelines and user-defined guidelines can NOT be invoked simultaneously. If both standard TEA and user-defined guidelines are to be invoked, each type must be invoked independently.

TEA looks at the VMS logical **tea_rules** to find a filename where the user-defined guidelines are listed, one per line. If the logical has not been set, the file **tea_rules** is used. Refer to the *TEA Reference Manual* for instructions on adding user guidelines to TEA.

The **dft identify** function allows the user to graphically isolate a particular schematic component (e.g., all memory elements) on the current view display. The selection of **identify** from the **dft** options menu invokes the function by displaying the subsystem selection menu.

Subsystem Selection

The user selects the subsystems of interest from the current menu display. The subsystem menu contains all the subsystems for the current graph hierarchy. The **%done** option invokes the next menu which is a list of all the boards from the previously selected subsystems. Figure 15 shows the terminal display for the **dft identify** select subsystem menu.

```
=== menu ===
%done           subsys2
%help           subsys3
%all            subsys4
subsys1         subsys5
============
DFT: Indicate the subsystem(s) of interest:
```

Figure 15. dft identify Select Subsystem Menu

Board Selection

Only those boards which reside on the previously selected subsystem(s) are included in the board menu. Once the **%done** option has been selected, the **dft identify** function is invoked. The user-selected boards are searched for the requested construct. Figure 16 shows the terminal display for the select board menu.

```
=== menu ===
%done          board3         board8         board13
%help          board4         board9         board14
%all           board5         board10        board15
board1         board6         board11        board16
board2         board7         board12
============
DFT: Indicate the board(s) of interest:
```

Figure 16. dft identify Select Board Menu

Identify Selection

Figure 17 shows the terminal display for the select identify menu.

```
=== menu ===
%help          fanout>5       node           register
attribute      flipflop       one_shot       scan_reg
bit_mod        loop           primaries      test_point
counter        memory         processor
============
DFT: Select an item to be identified:
```

Figure 17. dft identify Select Item Menu

Additional Selections

dft identify contains a set of predefined structures which may be identified in the current graph hierarchy. Some structures require additional information to correctly identify them in an ADAS hardware graph (i.e., loops, attributes).

Loops may be identified by selecting the **loop** option from the identify menu. The user must specify what types of loops are to be identified. There are three types of loop occurrences:

a. loops within a specific board,

b. loops between boards, and

c. loops between subsystems.

Figure 18 shows the terminal display for the **dft identify loop** menu.

```
=== menu ===
%help
on_board
brd_betw
sub_betw
============
DFT: Indicate the extent of the loops:
```

Figure 18. dft identify loop Menu

On-board loops are closed circuits which contain only nodes with the same values for both **Tboard** and **Tsubsystem.**

Loops of nodes between boards (or **brd_betw**) are closed circuits which contain nodes from two or more of the selected boards of each selected subsystem (refer to values of **Tboard** and **Tsubsystem**).

Loops of nodes between subsystems (or **sub_betw**) are closed circuits which contain nodes from two or more of the selected subsystems on selected boards (refer to values of **Tboard** and **Tsubsystem**).

NOTE: The loop routine is in Prolog which introduces a response delay if the graph has not yet been converted to Prolog and compiled.

ADAS attributes with the same value can be identified by selecting the **attribute** option from the **dft identify** menu. The user must specify what attribute is to be identified. The list of attributes is determined by the 'save_status' value for each attribute. If the 'save_status' value is set to 'S' then that attribute will appear in the **identify attribute** menu list. Figures 19 and 20 show the terminal displays for the **dft identify attribute** menu and the **dft identify attribute** value menu respectively.

```
=== menu ===
Tag_name          Tchip_maxfan     Thw_module        Tselftest
Tag_test          Tconfig          Tlogic_family     Tsp_ag_name
Tag_test_time     Tcount_bits      Tlogic_type       Tstates
Tasynch           Tdevice_type     Tnode_spec_type   Tsubsystem
Tbit              Tfault_cov       Tnode_type
Tboard            Tgroup           Tscan
============
Enter '*' to view additional menu items
DFT: Select an attribute to highlight:
```

Figure 19. dft identify attribute Menu

```
=== menu ===
%help
color1
color2
color3
============
DFT: Select one of the values of the attribute:
```

Figure 20. dft identify attribute Values Menu

Any of the other options are invoked immediately after their selection from the identify menu (i.e., no submenus).

Report Selection

Upon completion of the selected identify option, the user is prompted for a filename in which to store a report of the results for later examination. A default filename will be provided if the user chooses not to name the file. Program control is then returned to the TEA top-level menu. Figure 21 shows the terminal display for the **dft identify** report prompt.

```
Enter <cr> or YES to accept dft_01.rpt as the default filename
Or enter desired output filename :
```

Figure 21. dft identify Report Prompt

The **dft explain** function allows the user to see all the on-line information available about a particular guideline, including the purpose of the guideline and any recommended alternative constructions. The selection of **explain** from the **dft** menu invokes the guideline menu. The two types of guidelines are

a. aid to test pattern generation (g1) and
b. aid to test pattern application and fault isolation (g2).

Selecting **g1** invokes the next menu which contains each specific guideline pertaining to test pattern generation. Selecting **g2** invokes a menu which contains a list of the guidelines pertaining to aiding test pattern application. Figure 22 shows the terminal display for the **dft explain** type menu.

```
=== menu ===
%help
g1
g2
============
DFT: Select the appropriate guideline group:
```

Figure 22. dft explain Menu

Guideline Selection

A single guideline may be selected from the guideline menu. The guideline name is used to access the On-line User Support System (also accessible from **%help**) data base for the appropriate information. Figure 23 shows the terminal display for the **dft explain** guideline menu.

```
=== menu ===
%help           g1_05           g1_10           g1_15
g1_01           g1_06           g1_11           g1_16
g1_02           g1_07           g1_12
g1_03           g1_08           g1_13
g1_04           g1_09           g1_14
============
DFT: Select the specific guideline to be explained:
```

Figure 23. dft explain Guideline Menu

Report Selection

Prior to displaying the information, the user is prompted whether or not to store the explanation to a file for later examination. If the user selects **yes**, then the information is written to a file (the filename is chosen as shown in Figure 25). If the user selects **no**, then the information is only displayed on the terminal. Figure 24 shows the terminal display for the **dft explain** filename prompt.

```
=== menu ===
%help
yes
no
============
DFT: Store the explanation to a file?
```

Figure 24. dft explain Filename Prompt

Figure 25 shows the terminal display for the **dft identify** report prompt, if **yes** was selected from the previous menu.

```
Enter <cr> or YES to accept dft_01.help as the default filename
Or enter desired output filename :
```

Figure 25. dft identify Report Prompt

BIT Recommendation Tool

The BIT Recommendation Tool, **brt**, aids the user in determining whether deterministic or pseudorandom test vector oriented BIT techniques are advised for an ambiguity group. **Brt** determines the test method for each node from heuristics, divides a board into ambiguity groups, and then makes its general BIT technique recommendations.

Subsystem Selection

The **brt** function operates at the board level, so it is necessary that the user provide the name of the specific board on which the BIT recommendation is to be made. Selecting **brt** from the TEA top-level menu invokes the subsystem selection menu. The menu contains only those subsystems which exist in the current hardware graph hierarchy. Only one subsystem may be selected for **brt** and the next menu invoked contains the various boards located in the selected subsystem. Figure 26 shows the terminal display for the **brt** subsystem selection menu.

```
=== menu ===
%help          subsys4
subsys1        subsys5
subsys2        subsys6
subsys3
============
BRT: Indicate the subsystem of interest:
```

Figure 26. brt Subsystem Selection Menu

Board Selection

Only those boards which are contained within the selected subsystem are displayed on the **brt** board selection menu. A single board selection identifies which nodes are to be involved in the BIT recommendation. Selecting a specific board invokes the next prompt which requests the maximum ambiguity group size. Figure 27 shows the terminal display for the **brt** board selection menu.

```
=== menu ===
%help          board4
board1         board5
board2         board6
board3
============
BRT: Indicate the board of interest:
```

Figure 27. brt Board Selection Menu

Ambiguity Group Size Requirement

In order for **brt** to determine an optimal ambiguity grouping for the nodes on a board, the maximum number of nodes in ambiguity group must be known. The ambiguity group size can be any integer value greater than zero. The user is prompted for the group size requirement as shown in Figure 28. A value of zero indicates that a minimum number of ambiguity groups of unrestricted size is to be recommended. Ambiguity group is abbreviated AG in menus and reports.

```
BRT: Enter desired AG size:
```

Figure 28. brt Ambiguity Group Size Limit Prompt

Once the maximum ambiguity group size has been specified, the user is asked whether an altered cost function should be used. If the user responds with **yes**, then certain groups will be rewarded or penalized (altered weighting factors) for testability features or hard-to-test structures. See the *TEA Reference Manual* section 2.7.2 for details. If the user responds with **no**, then all enumerated groups are weighted strictly on connectivity. Figure 29 shows the menu for this selection.

```
=== menu ===
%help
yes
no
============
BRT: Alter cost function to account for testability?
```

Figure 29. brt Cost Function Menu

Report Selection

Upon completion of **brt**, the user is prompted for a filename in which to store the results of the tool. A default is provided if the user chooses not to enter a specific filename. Program control is then returned to the TEA top-level menu. Figure 30 shows the terminal display for the **brt** report prompt.

```
Enter <cr> or YES to accept brt_01.rpt as the default filename
Or enter desired output filename:
```

Figure 30. brt Report Prompt

BIT Overhead Summary

The BIT Overhead Summary function, **bit_cost**, determines the approximate cost of implementing a particular BIT technique. Included in the computation is the number of BIT modules and additional I/O that will be needed to incorporate a specific implementation of a BIT technique. The costs are itemized in an output report file.

Subsystem Selection

The **bit_cost** function operates at the board level so it is necessary that the user provide the name of the specific board where the BIT cost estimate is to be made. Selecting **bit_cost** from the TEA top-level menu invokes the subsystem selection menu. The menu contains only those subsystems which exist in the current hardware graph hierarchy. Only one subsystem may be selected for **bit_cost**. A selection invokes the next menu which contains the various boards located in the selected subsystem. Figure 31 shows the terminal display for the **bit_cost** subsystem selection menu.

```
=== menu ===
%help          subsys4
subsys1        subsys5
subsys2        subsys6
subsys3
============
Bit_cost: Indicate the subsystem of interest:
```

Figure 31. bit_cost Subsystem Selection Menu

Board Selection

Only those boards which are contained within the selected subsystem are displayed on the **bit_cost** board selection menu. A single board selection identifies which nodes are to be considered. Selecting a specific board invokes the next menu which requests the BIT technique(s) to be used in the Overhead Summary. Figure 32 shows the terminal display for the **bit_cost** board selection menu.

```
=== menu ===
%help          board4
board1         board5
board2         board6
board3
============
Bit_cost: Indicate the board of interest:
```

Figure 32. bit_cost Board Selection Menu

BIT Technique Selection

The user must select the BIT technique(s) to be considered. The user is prompted for the technique(s) to be used as shown in Figure 33.

```
=== menu ===
%done          %all           tp_sa          scan_set
%help          det_tp         boundary       gen_sa
============
Bit_cost: Select BIT technique(s):
```

Figure 33. bit_cost Technique Selection Menu

The menu options represent the following BIT techniques:

a. **det_tp** — test point monitoring on a cycle-by-cycle basis, while applying either deterministic or pseudorandom patterns at the board primary inputs. Patterns are assumed to be generated by using the BIT modules such as "Testing-Switch," "BILBO," and other available test pattern generator modules,

b. **tp_sa** — test point monitoring with compression of the test output results, while applying either deterministic or pseudorandom patterns at the board primary inputs. Compression of the test output results is assumed to be performed by using "BILBO" modules as parallel signature analyzers,

c. **boundary** — boundary scan using special registers at the primary inputs and outputs of a board,

d. **scan_set** — board level scan-set technique using scan-set BIT modules that are distributed over the board, and

e. **gen_sa** — test pattern application and response compression technique using "Testing-Switch" modules that are distributed over the board.

Any combination of techniques may be selected. The **%all** option provides quick selection of all techniques. When the desired techniques have been selected and are highlighted on the display, the **%done** option must be selected to proceed to the output report name prompt. The costs for multiple techniques are calculated independently. TEA does not provide automated support for multiple off-line BIT techniques on a single board.

Report Selection

The user is prompted for a filename in which to store the results of the BIT Overhead Summary. A default is provided if the user chooses not to enter a specific filename. The designation of a file invokes the tool and produces an output report. Program control is then returned to the TEA top-level menu. Figure 34 shows the terminal display for the **bit_cost** report prompt. High level information is shown in the ".rpt" file. Detailed information is placed in the ".dwg" file.

```
Enter <cr> or YES to accept bitcost_01.rpt as the default filename
Or enter desired output filename:
Enter <cr> or YES to accept bitcost_01.dwg as the default filename
Or enter desired output filename:
```

Figure 34. bit_cost Report Prompt

BIT Placement Recommendation

The BIT Placement Recommendation function, **placebit**, indicates to the user where to place BIT modules and/or test point outputs to implement a particular BIT technique. BIT Placement Recommendation is invoked from the TEA top-level menu by selecting the **placebit** option. Selecting **placebit** invokes the BIT technique selection menu.

BIT Technique Selection

In order for **placebit** to determine placement of modules on a single board, the specific technique which is to be applied must be known. The user is prompted for the technique to be used as shown in Figure 35.

```
=== menu ===
%help          tp_sa          scan_set       parity
det_tp         boundary       gen_sa         compare
============
Placebit: Select a BIT technique:
```

Figure 35. placebit Technique Selection Menu

Only a single technique may be applied at a time. Selection of the desired technique invokes the next menu. The techniques available are

a. **det_tp** — continuous test point monitoring,
b. **tp_sa** — test point monitoring with data compression,
c. **boundary** — board-level boundary scan,
d. **scan_set** — scan-set,
e. **gen_sa** — test generation and response compression,
f. **parity** — on-line fault detection using parity generation/checking, and
g. **compare** — on-line fault detection using equality comparison.

Example Implementation Selection

TEA will provide an example implementaion if the user requests it. Selecting **yes** from the Example Implementation menu will cause the tool to generate a properly

connected implementation. The **no** option will bypass this example implementation and general instructions are given in a file whose default name is **placebit_n.help**. Figure 36 shows the terminal display for the **placebit** example implementation menu.

```
=== menu ===
  %help
  yes
  no
============
Placebit: Do you want TEA to attempt an example implementation?
```

Figure 36. placebit Example Implementation Menu

Subsystem Selection

The **placebit** function operates at the board level, so it is necessary that the user provide the name of the specific board where the BIT placement recommendation is to be made. The menu contains only those subsystems which exist in the current hardware graph hierarchy. Only one subsystem may be selected for **placebit**, and its selection invokes the next menu which contains the various boards contained in the selected subsystem. Figure 37 shows the terminal display for the **placebit** subsystem selection menu.

```
=== menu ===
%help          subsys4
subsys1        subsys5
subsys2        subsys6
subsys3
============
Placebit: Indicate the subsystem of interest:
```

Figure 37. placebit Subsystem Selection Menu

Board Selection

Only those boards which are contained within the selected subsystem are displayed on the **placebit** board selection menu. A single board selection identifies which nodes are to be involved in the BIT placement recommendation. Selecting a specific board invokes the next prompt which requests whether the wiring list should be automatically placed or printed to an output file. Figure 38 shows the terminal display for the

placebit board selection menu.

```
=== menu ===
%help          board4
board1         board5
board2         board6
board3
============
Placebit: Indicate the board of interest:
```

Figure 38. placebit Board Selection Menu

Manual Wiring List/Autoplacement Option

A wiring list, generated by **placebit**, indicates the proper connectivity of the BIT modules with the nodes on the selected board. **autoplace** will automatically update the ADAS data base with BIT modules and/or test points. Three options are provided for obtaining these results:

a. **autoplace** — for automatic placement and connection of the BIT modules on the current board,

b. **wirelist** — for printing a wiring list for manual placement and connection of the BIT modules on the current board, and

c. **%all** — for performing both an automatic placement as well as providing a wire list.

Figure 39 shows the terminal display for the **placebit wirelist** menu.

```
=== menu ===
%done
%help
%all
autoplace
wirelist
============
Placebit: Place and/or supply manual wiring list?
```

Figure 39. placebit wirelist Menu

When selections have been completed, the **%done** option invokes the output report filenamc prompt.

Report Selection

The user is prompted for a filename in which to store the results of the BIT Placement Recommendation. A default is provided if the user chooses not to enter a specific filename. The designation of a file invokes the tool and produces an output report. Program control is then returned to the TEA top-level menu. Figure 40 shows the terminal display for the **placebit** report prompt if **autoplace** is chosen. A ".dwg" will not be created if **autoplace** is not selected.

```
Enter <cr> or YES to accept placebit_01.rpt as the default filename
Or enter desired output filename:
Enter <cr> or YES to accept placebit_01.dwg as the default filename
Or enter desired output filename :
```

Figure 40. placebit Report Prompt

System Summary

The **compare** utility provides a method of accessing the System Summary tool and compares attributes from two ADAS hardware graph data bases. The specific attribute information includes incremental changes in

a. node counts,

b. I/O counts,

c. mean time to test, and

d. mean ambiguity group size.

Directory Selection

The **compare** utility is invoked from the TEA top-level menu using the **compare** command. The comparison will always involve the currently displayed hardware graph. The other graph may reside in some other directory and therefore must be identified by the user. The first prompt provided after selecting **compare** from the main TEA menu is a request for the directory of the second graph as shown in Figure 41.

```
Enter directory path to search.
<cr> or YES search current dir:
```

Figure 41. compare Directory Search Prompt

File Selection

Entering a valid directory (or using the current as default) invokes the **compare** menu which includes only the hardware graphs from the specified directory. The user must supply the filename which indicates the hardware graph to be compared with the current view graph. An additional option, called **%self** in the menu, allows **compare** to run only on the current view. The **%self** option will always be present as a selection no matter which directory is specified at the start of the utility. Figure 42 shows the terminal display for the **compare** options menu.

```
=== menu ===
%help
%self
file1.hwg
file2.hwg
============
Compare: Select the comparison graph:
```

Figure 42. compare Options Menu

Selecting a single file invokes the **compare** utility. Two output files are generated from the **compare** utility:

a. general output report (file.rpt) and

b. additional information report (file.dwg).

Report Selection

The user is prompted for filenames for each report. Default filenames have been provided. Figure 43 shows the terminal display for the filename prompts.

```
Enter <cr> or YES to accept compare_01.rpt as the default filename
Or enter desired output filename :
Enter <cr> or YES to accept compare_01.dwg as the default filename
Or enter desired output filename :
```

Figure 43. compare Output Filename Prompts

High-level summary information is placed in the ".rpt" file. Lists of detailed information is placed in the ".dwg" file.

The user is notified if any errors were detected in the comparison graphs, and program control returns to the TEA top-level menu.

Ambiguity Group Names

The Ambiguity Group Names utility, **ag_name**, is used to access the ambiguity group (AG) name of any node. These can be viewed and/or edited. **Ag_name** provides the ability to create, view, and edit *special* ambiguity groups. *Special* ambiguity groups are user-defined groups of one or more nodes which are to be grouped as a single ambiguity group. There is no limit on the number of *special* ambiguity groups as long as there are available nodes. A node may be assigned to only one group.

Subsystem Selection

The subsystem of interest must be selected. Since the **ag_name** utility operates on a single board at a time, only one subsystem may be chosen. Selecting a subsystem invokes the next menu which is a list of boards for the selected subsystem. Figure 44 shows the terminal display for the **ag_name** select subsystem menu.

```
=== menu ===
%help          subsys4
subsys1        subsys5
subsys2        subsys6
subsys3
============
Ag_name: Indicate the subsystem of interest:
```

Figure 44. ag_name Select Subsystem Menu

Board Selection

Only those boards which reside on the previously selected subsystem are included in the board menu. Only a single board may be selected from the menu. Selecting a board invokes the next menu which is a list of **ag_name** options. Figure 45 shows the terminal display for the select board menu.

```
=== menu ===
%help          board5        board10       board15
board1         board6        board11       board16
board2         board7        board12
board3         board8        board13
board4         board9        board14
============
Ag_name: Indicate the board of interest:
```

Figure 45. ag_name Select Board Menu

AG Name Selection

The **ag_name** utility allows the user to force special nodes in a hardware graph to a particular ambiguity group. In addition, regular ambiguity group nodes, which have been determined by the **brt** tool may be modified using the **ag_name** utility. The only restriction involved is that all nodes in an ambiguity group (regular or special) must be connected. A connectivity checker is used to determine if a new or modified group of nodes can form an ambiguity group.

Any category may be invoked by selecting it from the menu. Figure 46 shows the terminal display for the **ag_name** type menu.

```
=== menu ===
%help
spcl_ag
reg_ag
conn_check
============
Ag_name: Modify special or regular AG:
```

Figure 46. ag_name Type Menu

Special Ambiguity Groups

A special ambiguity group is a collection of nodes that the user wishes to be in a group together for testing. If **spcl_ag** is selected from the **ag_name** type menu, then a list of all the currently defined special ambiguity groups is displayed on a menu along with an option to add a new special ambiguity group. Adding a new group requires a unique ambiguity group name. Modifying an existing group is done by selecting the

group from the menu. Figure 47 shows the terminal display for the select **spcl_ag** menu.

```
=== menu ===
  %help
  %add_new
  spag_1
  spag_2
============
Ag_name: Indicate the special AG of interest:
```

Figure 47. spcl_ag Select Menu

Add New Special AG

A new special ambiguity group name must be unique to the existing ambiguity group names. Once a unique name has been entered, the next menu is invoked which gives a list of all the nodes for the currently selected board. Figure 48 shows the terminal display for the **spcl_ag add_new** name menu.

```
Ag_name: Input the name for the special AG:
```

Figure 48. ag_name Prompt

Nodes in New Special AG

The user is prompted to select those nodes from the selected board which are to be in the new special AG. The nodes listed in the menu do not represent any form of connectivity. When all nodes have been selected for the new group, and the **%done** option has been selected, the utility invokes the connectivity checker to verify that the entered nodes represent a valid (connected) group. If the group is valid, and creating it did not disturb the connectivity of the other AGs on the board, then program control is returned to the TEA top-level menu. If the group is invalid, however, a report is output and then control is returned to the top-level menu. Figure 49 shows the terminal display for the **add_new spcl_ag** menu.

```
=== menu ===
%help
%done
node_1
node_2
node_3
============
Ag_name: Indicate the node(s) of interest:
```

Figure 49. ag_name Node Menu

Modifiy Existing Special AG

Any of the existing special AGs may be edited by selecting the existing AG name from the special AG menu. The special AG menu contains all of the specially defined AGs for the current board. When exiting, the connectivity checker will run.

Regular Ambiguity Groups

Nodes which have been previously assigned to ambiguity groups via the **brt** tool may be reorganized by selecting the group from the **ag_name** group menu and the nodes to be included in the AG from the **ag_name** node menu. Figures 50 and 51 show the terminal display for the **ag_name** group and **ag_name** node menus, respectively.

```
=== menu ===
%help
ag_1
ag_2
ag_3
ag_4
============
Ag_name: Indicate the AG of interest:
```

Figure 50. ag_name Group Menu

```
=== menu ===
%help        node4
%done        node5
node1        node6
node2        node7
node3
============
Ag_name: Indicate the nodes of interest:
```

Figure 51. ag_name Node Menu

The choices will be checked for valid connectivity. **Ag_name** does not allow for the addition of new regular AGs.

Report Selection

Upon completion of **ag_name spcl_ag add_new**, the user is prompted for a report name in which to store the results of the tool if there is an error in the connectivity of the ambiguity groups. A default is provided if the user chooses not to enter a specific report name. Program control is then returned to the TEA top-level menu. Figure 52 shows the terminal display for the **ag_name** report prompt.

```
Enter <cr> or YES to accept ag_name_01.rpt as the default filename
Or enter desired output filename :
```

Figure 52. ag_name Report Prompt

Connectivity Check

This option allows the user to check for connectivity between ambiguity groups.

Report Access/Generation

The **log_file** utility allows the user to access the results of any of the TEA tools. Selecting **log_file** from the TEA top-level menu invokes the Log File Type selection menu.

The **log_file** utility provides two main functions for any on-line TEA tool reports or wiring diagrams:

a. **view** — for displaying a report, help, or wiring diagram on the terminal display and

b. **print** — for producing hardcopy reports on a local printer.

Figure 53 shows the terminal display for the **log_file** menu.

```
=== menu ===
%help
view
print
============
Logfile: View or print report?
```

Figure 53. log_file Menu

Viewing Reports

Any report may be viewed if the user has proper access. Selecting **view** from the **log_file** type menu invokes a prompt for the directory to be searched for report files. The default is the current directory and is used only when a 'yes' response or carriage return is entered in response to the directory prompt. Figure 54 shows the terminal display for the directory prompt.

```
Enter directory path to search.
<cr> or YES searches current dir:
```

Figure 54. log_file Directory Search Prompt

Indicating the appropriate directory invokes a search of that directory for report files. Those report files are displayed on the next menu. Figure 55 shows the terminal display of report files from a particular directory.

```
=== menu ===
%help
file1.rpt
file2.help
file3.dwg
============
Log_file: Select a report:
```

Figure 55. log_file Report Options Menu

Selecting a report file from the menu displays that file in page lengths on the terminal screen. The user may scroll through the file by entering a carriage return after each page is displayed. The **view** option is identical to a VMS *type/page* command.

Printing Reports

Printing a report is similar in operation to viewing except that the **print** option is selected from the **log_file** menu. The sequence of menus and prompts is identical. The printer used is the user's default printing device. Alternate printers may NOT be accessed via the TEA **log_file** utility.

On-line User Support System

All TEA selection menus provide on-line user support via the **%help** menu option. Selecting **%help** provides default information which corresponds to the most useful information based on the menu currently being displayed. The **%help** function operates identically to the VMS *help* facility. The user always has the option to access any part of the **%help** on-line information by typing a "?" at the topic prompt. A "?" will display the current areas of help which are available. A carriage return will either back the user out of the help hierarchy by one level or exit help if the user is already at the top level. Typing the name of a selection followed by a carriage return selects that information and any subtopic items will be displayed. Figure 56 shows the terminal display of on-line topics.

```
Information available:

Functions  Libraries  Menus      Tools      Utilities


Topic?
```

Figure 56. TEA On-line Topics Menu

From the On-line Topics menu, the user may proceed to any desired on-line information. On-line help is NOT provided for TEA prompts which do not generate a menu.

ALPHABETICAL LIST OF TEA ERROR MESSAGES

The following list of error messages is sorted alphabetically by command code (columns 6-7) and numerically by error number within a command. Each entry lists the *meaning* of the error message and the *action* that must be taken to correct the error. The standard ADAS error message format is explained the *ADAS User Manual*, Version 2.3 or later.

*** ZAG 002A — Out of memory.

Meaning: The computer's available memory has been exceeded.

Action: Re-run with more computer memory.

*** ZBB 001E — No valid nodes to make ambiguity groups.

Meaning: There are no valid nodes in the current graph.

Action: Check subsystem and board attribute values.

*** ZBB 002A — No solution found.

Meaning: Nodes cannot be formed into ambiguity groups.

Action: Contact RTI for assistance.

*** ZBB 004A — Could not set up initial basis for simplex.

Meaning: Could not form nodes into AG's of size one.

Action: Contact RTI for assistance.

*** ZBM 000W — Invalid directory path *path*

Meaning: Specified directory path does not exist.

Action: Re-enter correct path.

*** ZBR 001W — Can't access BRT library.

Meaning: The location of the BRT library has not been properly identified.

Action: Check TEA_BRT define in TEA setup file.

*** ZBR 002E — Illegal ambiguity group size.

Meaning: The integer entered for ambiguity group size must be positive.

Action: Re-enter the ambiguity group as a positive integer.

*** ZCA 001E — Node name could not be interpreted.

Meaning: The node selection was incorrect.

Action: Re-enter the node name at the keyboard or select the name from the display menu or graph.

*** ZCA 002E — Node (or bus) location could not be interpreted.

Meaning: The given coordinates contained an error.

Action: Re-enter the location.

*** ZCA 003E — Arc could not be interpreted.

Meaning: Arc selection was incorrect.

Action: Re-enter name.

*** ZCA 004E — An error occurred. Please check the input.

Meaning: An unspecified error was found.

Action: Check the input and re-enter.

*** ZCI 001E — *Command* is not a unique name.

Meaning: The command cannot be selected from the command list because there are ambiguous choices.

Action: Re-enter the command with enough information to make a unique selection.

*** ZCI 002E — An error occurred in your selection. *String* is an incorrect response. Please reenter this item.

Meaning: Command entered was not on menu or was an incorrect response.

Action: Re-enter command.

*** ZCM 004A — File *name* cannot be found.

Meaning: Source graph listed on the command line cannot be found.

Action: Check graph filename or location.

*** ZCO 001E — Too many repetitions requested.

Meaning: The upper limit for repeating a command was exceeded.

Action: Re-enter a smaller number.

*** ZDB 001A — There are no more unused graph descriptors.

Meaning: The graph is full.

Action: Contact RTI for help.

*** ZDB 003A — The template file cannot be accessed.

Meaning: Cannot open data base file for reading.

Action: Check read access of graph file.

*** ZDB 005A — Premature end-of-file encountered while reading template file *file*.

Meaning: It is likely that the file has been damaged, since part of its data is missing.

Action: Examine the template file using **EDIGRAF**. If it has been damaged, recreate it. Periodically creating backup copies of important data base files limits the extent of this kind of damage.

*** ZDB 006A — The template file cannot be read.

Meaning: The template file's contents are not interpretable.

Action: Make sure that the proper template file is being used.

*** ZDB 007A — Not enough memory to expand graph.

Meaning: Expanding this graph would exceed the computer's available memory.

Action: Divide the graph nodes into subgraphs. Contact RTI if further help is needed.

*** ZDF 000A — Can't start prolog servant.

Meaning: The prolog code can not be invoked.

Action: See local ADAS administrator for proper prolog installation.

*** ZDF 001W — Can't create control file in current directory.

Meaning: A prolog control file is needed and it cannot be written in this directory.

Action: Check write access in this directory.

*** ZDF 002A — Can't open user rules file.

Meaning: The user's prolog code can not be accessed.

Action: See local ADAS administrator.

*** ZED 001E — Value *name* has an invalid format.

Meaning: Format of the attribute value is inconsistent with its data type. Spaces and tabs are not accepted.

Action: Correct the information and retry.

*** ZED 002E — Value *name* is out of range.

Meaning: Value of a numerical attribute is out of range.

Action: Correct the information and retry.

*** ZED 003E — Attribute *att* is not modifiable.

Meaning: The modification status for the named attribute does not permit the value of the attribute to be changed in the editor (i.e., 'N').

Action: Use the **edit status** command to change the status.

*** ZED 004E — Value *name* is not a valid name.

Meaning: Name is not a correct ADAS graph element name.

Action: Check the *ADAS User Manual*, Version 2.3 or later, for correct name rules.

*** ZED 005W — Status of attribute *att* for *graph element* is not modifiable.

Meaning: The modification status for the named attribute does not permit the value of the attribute to be changed in the editor.

Action: Use the **edit status** command to change the status.

*** ZED 006W — Special characters -,+,^ ignored.

Meaning: The special characters are not used in the current edit mode.

Action: Enter a new value, cancel the edit session, or go on to the next selection with a <RET>.

*** ZED 007W — There is no *port_type number* for node *name*.

Meaning: The port is not of this type.

Action: No action required.

*** ZED 008W — *Count* more attempts. Try again.

Meaning: *Count* more attempts at modifying the current attribute will be allowed before going on to the next attribute.

Action: Refer to the *ADAS User Manual*, Version 2.3 or later, for proper values of the current attribute.

*** ZED 009W — No more attempts. Continuing edit.

Meaning: After previous attempts to modify the current attribute have failed, the editor is moving on to the next attribute.

Action: See the *ADAS User Manual*, Version 2.3 or later, for the valid attribute values of the current attribute and attempt to edit the attribute again.

*** ZED 010W — No *port types/junctions/connections* for node/bus *name*, skipping.

Meaning: The node/bus *name* does not have any ports of the named type.

Action: No action required.

*** ZED 011W — No arc on *port_type number* of node *name*, skipping.

Meaning: The node/bus *name* does not have any ports of the named type.

Action: No action required.

*** ZED 012W — Attribute *att* for *graph element*, is not modifiable.

Meaning: The modification status for the named attribute does not permit the value of the attribute to be changed in the editor.

Action: Use the **edit status** command to change the status.

*** ZED 013W — Attribute *att* is not present in the template data base.

Meaning: This data base does not have an attribute of this name.

Action: Check attribute name.

*** ZGD 001A — Can't create graph.pl.

Meaning: graph.pl is the prolog data base of the ADAS graph hierarchy and the file cannot be written.

Action: Check write access in this directory.

*** ZHC 001A — Can't reinitialize current graphics device *dev.*

Meaning: After the hardcopy device was initialized, an error occurred in reinitializing the normal display device.

Action: Begin a new TEA session.

*** ZHC 002E — Can't initialize hardcopy device *dev.*

Meaning: The initialization of the hardcopy device failed.

Action: Make sure the hardcopy device is plugged in, turned on, and properly initialized.

*** ZHC 003E — Can't open hardcopy file *name.*

Meaning: The file containing the current list of valid hardcopy devices cannot be accessed.

Action: Check with the local ADAS administrator.

*** ZHE 001E — Can't open help file *file.*

Meaning: The specified help file either could not be found or could not be read.

Action: Check with the local ADAS administrator.

*** ZIN 001E — Input exceeded maximum line length of *linelength.*

Meaning: Too many characters were given at the keyboard.

Action: Re-enter.

*** ZIN 002E — Input line ignored.

Meaning: An error was found in the input, or the entry was irrelevant.

Action: Re-enter.

*** ZIN 003E — Data tablet error occurred.

Meaning: An error occurred in the data tablet.

Action: Try again, or contact local systems help or RTI.

*** ZIN 004E — Unspecified error from graphics package.

Meaning: An error occurred in a graphics device.

Action: Try again, or contact local systems help or RTI.

*** ZIN 005A — Unexpected end of input encountered.

Meaning: The input to the program terminated unexpectedly.

Action: If the keyboard or data tablet was in use when the error occurred, no action is necessary. If the input was from a script file, the file contains errors and should be checked. If the input was from a file other than a script file, check for the proper session termination commands, e.g., **quit nosave.**

*** ZMA 001A — Unknown option *name.*

Meaning: The command line option *name* was invalid.

Action: Re-enter the correct command line.

*** ZMA 002A — No graphics display specified.

Meaning: No graphics device was specified.

Action: A graphics device must be specified. Re-enter the correct command line.

*** ZMA 004A — Couldn't open graphics device *name.*

Meaning: A valid graphics device was specified but is either inoperable or in use.

Action: Either use a different device or try again later.

*** ZMA 005A — No data base specified.

Meaning: No data base was specified on the command line.

Action: Specify the correct data base.

*** ZMC 001E — Macro not added. At maximum number of macros.

Meaning: The maximum allowable number of macros already exists.

Action: Either remove a current macro to add another, or do not try to add a macro.

*** ZMC 002E — Macro name not valid. 1st character non-alphabetic.

Meaning: The first character of any macro command must be a letter of the alphabet.

Action: Choose another macro name that is acceptable.

*** ZMC 003E — Macro name is already in menu.

Meaning: The macro name is in conflict with another identical name.

Action: Use a different and unique name.

*** ZMC 004E — No macro name provided.

Meaning: When defining a macro, the name was not included.

Action: Re-enter using a specific and unique name.

*** ZMC 005E — Too many macro parameters provided.

Meaning: There is a limit to the number of parameters that a macro definition may have.

Action: Re-enter the macro definition.

*** ZMC 006E — Macro is undefined.

Meaning: An error was found, and the macro was not defined as a result.

Action: Redefine macro.

*** ZMC 007E — Macro not added. Menu item limit exceeded.

Meaning: There is a limit to the number of menu items that are allowed.

Action: Delete a macro in order to add a new one.

*** ZMC 008E — No macros to be deleted.

Meaning: The macro list is empty.

Action: None.

*** ZMC 009E — No macros to be listed.

Meaning: There are no macros currently defined.

Action: None

*** ZMN 001A — Out of memory.

Meaning: TEA ran out of memory.

Action: Contact RTI for assistance.

*** ZMN 002E — Insufficient memory to create menu.

Meaning: TEA ran out of memory.

Action: Contact RTI for assistance.

*** ZMN 003E — Can't delete *item name*: not in menu.

Meaning: Deletion cannot be performed since item is not present.

Action: None.

*** ZMN 004E — Item *item_name* is already in menu.

Meaning: The item name is not unique.

Action: Rename the item with a unique name.

*** ZMS 000W — Reloading graph *graphname*.

Meaning: The graph *graphname* is being reloaded from the disk file.

Action: No action.

*** ZPB 000A — Fatal template load error.

Meaning: The template file corresponding to the current graph cannot be loaded properly.

Action: Make sure that a template file exists in the proper directory corresponding to the current graph.

*** ZPO 001E — Port mismatch between node ***name*** and subgraph ***name***.

Meaning: A graph output has been assigned to more or less than one node.

Action: Adjust subgraph consistency.

*** ZPO 002E — Too many node ports on graph port *number* in subgraph *name*.

Meaning: There is a graph port with too many node ports.

Action: Remove the extra ports or adjust subgraph consistency.

*** ZPO 003E — No arc connected to graph port *number* in subgraph *name*.

Meaning: There is a graph port with no attached arcs.

Action: Add arcs to graph port, appropriately remove graph port, or adjust subgraph consistency.

*** ZPS 000A — Can't open BIT template file.

Meaning: The template file associated with the graph is not present or cannot be accessed for some reason.

Action: Make sure that the template file has the correct name and proper read/write permissions.

*** ZPS 001E — Arc template not found. Can't reroute port *port#*.

Meaning: The arc template of the arc being rerouted is not present or cannot be accessed for some reason.

Action: Make sure that the template file has the correct arc template.

*** ZPS 002E — Arc template not found. Can't intercept port *port#*.

Meaning: The arc template of the arc being intercepted is not present or cannot be accessed for some reason.

Action: Make sure that the template file has the correct arc template.

*** ZPS 003W — Node *nodename* had Tag_name and Tsp_ag_name set.

Meaning: Both attributes are set in the current graph.

Action: Delete one attribute entry so that there is only one set.

*** ZRE 001E — Can't access subgraph data base *file*.

Meaning: The data base file associated with the subgraph is not present or cannot be accessed for some reason.

Action: Make sure that the file has the correct name and proper read/write permissions.

*** ZRE 002E — Can't read subgraph data base *name*.

Meaning: There was an error in reading the file.

Action: Contact RTI for assistance.

*** ZRE 003E — Premature EOF encountered while reading subgraph data base *name*.

Meaning: It is likely that the file has been damaged since part of the data is missing.

Action: Examine the file using **EDIGRAF**. If it has been damaged, recreate it. Periodically creating backup copies of important data base files limits the extent of this kind of damage.

*** ZRE 004E — Not enough memory to read in subgraph data base.

Meaning: Expanding the subgraph would exceed the computer's available memory.

Action: Divide the subgraph into lower level subgraphs.

*** ZRE 005E — Error in reading subgraph data base *name*.

Meaning: There was an error in reading the file.

Action: Contact RTI for assistance.

*** ZRE 006A — Can't reinitialize graph data base.

Meaning: The in-memory data base could not be initialized upon reloading.

Action: Contact RTI for assistance.

*** ZRE 007A — Can't access graph data base *name*.

Meaning: The data base file associated with the graph is not present or cannot be accessed for some reason.

Action: Make sure that the file has the correct name and proper read/write permissions.

*** ZRE 008A — Can't read graph data base *name*.

Meaning: There was an error in reading the file.

Action: Contact RTI for assistance.

*** ZRE 009A — Premature EOF encountered while reading graph data base *name*.

Meaning: It is likely that the file has been damaged since part of the data is missing.

Action: Examine the file using TEA. If it has been damaged, recreate it. Periodically creating backup copies of important data base files limits the extent of this kind of damage.

*** ZRE 010A — Not enough memory to read in graph data base.

Meaning: Expanding this graph would exceed the computer's available memory.

Action: Divide the graph into subgraphs.

*** ZRE 011A — Error in reading graph data base *name*.

Meaning: There was an error in reading the data base.

Action: Contact RTI for assistance.

*** ZRE 012W — Reloading subgraph *name*.

Meaning: The current subgraph is being reloaded from its external file.

Action: None

*** ZRE 013E — Can't reload subgraph *name.*

Meaning: There was an error in reading subgraph file *name.*

Action: Contact RTI for assistance.

*** ZRE 014W — Now editing parent graph *name.*

Meaning: Parent graph *name* is the current graph.

Action: None

*** ZRE 015W — Reloading graph *name.*

Meaning: The current graph *name* is being reloaded from its external file.

Action: None

*** ZRG 000A — Make report can't open *filename.*

Meaning: The file *filename* cannot be opened by the report generator.

Action: Check access permissions in the directory.

*** ZRG 001W — Can't create output file *filename.*

Meaning: The file *filename* cannot be opened by the report generator.

Action: Check access permissions in the directory.

*** ZSA 001W — Updated graph *name.*

Meaning: The current graph data base was saved.

Action: None.

*** ZSA 002W — Created new graph *name.*

Meaning: Current graph saved under a new filename.

Action: None.

*** ZSA 003W — Cannot open graph data base *name* for writing.

Meaning: The data base file and/or directory associated with the graph does not have write permission.

Action: Make sure that the data base file and/or directory has the correct name and proper write permissions.

*** ZSA 004W — Graph data base not saved.

Meaning: The current graph and results of this session were not saved.

Action: Run TEA again.

*** ZSA 005W — Template data base file not saved.

Meaning: The current template and results of this session were not saved.

Action: Run TEA again.

*** ZSA 006W — Created a copy of *name* as *name1.*

Meaning: A new copy of the graph has been stored as requested.

Action: None.

*** ZSA 008E — File *name* does not have a valid extension.

Meaning: File names must have a hardware graph name, including extension (.hwg).

Action: Check the filename.

*** ZSA 009W — File *name* already exists.

Meaning: A file with this name already exists.

Action: Choose a unique name.

*** ZSA 007E — File *name* was incorrectly copied.

Meaning: Report of unsuccessful termination of **save.**

Action: Retry **save** or check write access.

*** ZSC 001E — Input exceeded maximum line length of *max length.*

Meaning: There is a line length limit that was exceeded.

Action: Re-enter using shorter string.

*** ZSC 002E — Input line ignored.

Meaning: An error occurred or the input line was irrelevant.

Action: Check input and command prompt.

*** ZSC 003E — Cannot open script file *name*.

Meaning: The script file *name* does not exist or does not have the proper read/write permissions.

Action: Check for the file's existence and proper permissions.

*** ZSC 004E — Due to an input error all script files have been closed.

Meaning: An error has occurred in a script file.

Action: Fix script file.

*** ZST 001A — Out of memory.

Meaning: TEA ran out of memory.

Action: Contact RTI for assistance.

*** ZSU 001E — At maximum depth in hierarchy.

Meaning: Only 20 levels of hierarchy are allowed.

Action: Change the structure of the graph so that fewer levels are required.

*** ZSU 002E — No subgraphs found in this graph.

Meaning: The **subgraf** command cannot be issued since there are no subgraphs associated with the current graph.

Action: Use **EDIGRAF** to alter the structure of the graph.

*** ZTC 000W — File *filename* cannot be found.

Meaning: The second graph involved in the compare cannot be isolated according to the name entered.

Action: Check the filename and location of the second graph.

*** ZTR 001E — Couldn't open subgraph *name*.

Meaning: While trying to expand the root graph, a subgraph was encountered that exists but could not be opened.

Action: Check the permissions of the file containing the subgraph data base to see if it is readable.

*** ZTR 002E — Premature end-of-file while reading subgraph data base *name*.

Meaning: TEA hit end-of-file while reading the subgraph data base.

Action: Examine the template file using **EDIGRAF**. If it has been damaged, recreate it. Periodically creating backup copies of important data base files limits the extent of this kind of damage.

*** ZTR 003A -- Ran out of memory trying to read subgraph data base *name*.

Meaning: Graph too big.

Action: Reduce the size of the main graph by dividing it into subgraphs. Simulate individual subgraphs; then propagate information upwards by editing the parent nodes' attributes.

*** ZTS 001A — Can't find prolog results file.

Meaning: The results posted by prolog are not available.

Action: Check that prolog is properly installed and running.

*** ZTT 000A — Data base attribute errors.

Meaning: Some or all of the required attributes for TEA have not been entered for the current graph.

Action: Edit all required attributes and fill in the appropriate values.

*** ZTT 001W — Attribute *attr_value* missing for node *node*.

Meaning: A specific attribute is missing for a particular node.

Action: Edit the node attribute and fill in the appropriate value.

*** ZTT 002W — Attribute *attr_value* missing for port *port#*.

Meaning: A specific port attribute is missing for a particular node.

Action: Edit the port attribute and fill in the appropriate value.

*** ZTT 003W — Attribute *attr_value* missing for arc *arc_name.*

Meaning: A specific arc attribute is missing for a particular arc.

Action: Edit the arc attribute and fill in the appropriate value.

*** ZXX 001A — Not enough memory.

Meaning: There is not enough memory for the graph.

Action: Reduce the size of the graph.

*** ZXX 002W — Invalid filename extension *ext* on file *file.*

Meaning: The extension on the indicated file (e.g., *.swg*) is invalid.

Action: Change the filename extension.

*** ZXX 003E — Can't access template file *name.*

Meaning: The file *name* does not exist or does not have the appropriate read/write permissions.

Action: Check for the file's existence and proper permissions.

*** ZXX 004E — Can't initialize template data base.

Meaning: The maximum number of internal data bases has already been reached.

Action: Contact RTI for assistance.

*** ZXX 005E — Can't read template file *name*.

Meaning: The template data base *name* is not in the expected format.

Action: Contact RTI for assistance.

*** ZXX 006E — Premature EOF encountered while reading template file *name*.

Meaning: It is likely that the file has been damaged since part of the data is missing.

Action: Recreate the file.

*** ZXX 007E — Not enough memory to read in template file.

Meaning: There is not sufficient memory to create an internal template data base.

Action: Contact the local ADAS administrator.

*** ZXX 008E — No source ports for arc.

Meaning: No nodes have an available input port.

Action: No action required.

*** ZXX 009E — Error in reading template file *filename*.

Meaning: There was an error in reading the template file.

Action: Try again with the same or different file, or contact RTI for assistance.

*** ZXX 010W — No template file: creating new template data base *name.*

Meaning: A template data base is being created for the new subgraph data base. The template data base will have name *name.*

Action: No action is required.

*** ZXX 011E — No nodes or partitions found in this graph.

Meaning: There is no node or partition to choose.

Action: No action is required.

*** ZXX 012E — No buses found in this graph.

Meaning: There is no bus to choose.

Action: No action is required.

*** ZXX 020E — No filenames defined.

Meaning: No filename is listed by the attribute chosen.

Action: Check attribute value.

*** ZXX 022E — The file, *name*, does not exist.

Meaning: The filename listed by the attribute cannot be found.

Action: Check existence of file and read access.

*** ZXX 023W — Nodes with the same attribute value for *name* have different node templates.

Meaning: More than 1 node matched the attribute value and they have different templates.

Action: Correct inconsistency.

*** ZXX 024W — Default filename *name* already exists.

Meaning: A file with the default name value exists.

Action: None required.

*** ZXX 025E — Graph contains no nodes.

Meaning: No nodes found in graph of this class specification.

Action: None required.

*** ZXX 026E — Invalid node class for *name*.

Meaning: A node must have a class of value (internal, leaf, inport, outport, or biport) to be recognized.

Action: Check class value.

*** ZXX 027E — *name* is not a valid node name.

Meaning: Name is not a valid ADAS node name.

Action: Correct the entry and retry. Check the *ADAS User Manual*, Version 2.3 or later, for correct name rules.

*** ZXX 028W — Error allocating memory.

Meaning: TEA cannot access enough memory.

Action: Contact RTI for assistance.

*** ZXX 029E — Graph attribute *name* does not exist.

Meaning: This graph data base does not contain attributes of this value.

Action: No action required.

*** ZXX 030E — Unable to access directory *name*.

Meaning: Error reading directory.

Action: Check name.

*** ZXX 031E — No *extension* files found.

Meaning: Error creating the menu of files.

Action: Check entry and retry.

REFERENCE MANUAL
FOR THE
TEST ENGINEER'S ASSISTANT SYSTEM

Reference Manual
for the Test Engineer's Assistant System

TABLE OF CONTENTS

Paragraph Page

LIST OF FIGURES

LIST OF TABLES

APPENDICES

1. SCOPE

1.1. Identification. This Reference Manual provides operational information for users of the computer software configuration item (CSCI) identified as the *Integration of Very High Speed Integrated Circuit (VHSIC) System Functional Design and Design for Testability* tools, to be known as the Test Engineer's Assistant (TEA) system.

This document should be used in conjunction with the Software System User's Manual for the Test Engineer's Assistant system. This reference manual will help the user prepare input for the TEA tools; the User's Manual will aid in the mechanics of the use of the tools; this reference manual will be useful in the analysis of the outputs of the tools and the direction that the user should take dependent on that output.

1.2. Purpose. The Architecture Design and Assessment System (ADAS) provides a graphic schematic capture capability to VHSIC designers. TEA provides a design automation system for the incorporation of design for test (DFT) and built-in test (BIT) techniques into VHSIC digital hardware described using ADAS and VHSIC Hardware Description Language (VHDL). TEA provides the methodology and the tools for evaluating proposed designs for hard-to-test digital hardware constructs, for recommending alternative constructs, and for identifying hierarchical BIT techniques to increase the overall testability of a hardware system. TEA aids designers at the board, subsystem, and system levels. TEA estimates the costs associated with the BIT techniques to allow the designer to make informed choices about BIT incorporation. A library of reusable BIT modules supports TEA BIT techniques and enables the designer to simulate the entire system in VHDL. ADAS provides the interface to VHDL from the graphic connectivity schematic.

The main purpose of this document is to present helpful information about the TEA methodology of testable design, the purpose and use of the TEA tools and utilities, how to prepare ADAS graphs for use with the TEA tools, and the library of reusable BIT modules. The designers of the system have identified procedures and hints for use and compiled them into this document.

1.3. Introduction. This Reference Manual contains background information on TEA, describes the TEA design methodology, and fully documents the TEA tools. It will instruct in the use of the tools and the analysis of the results. An example will be illustrated and demonstrated in the final section. The doppler filter card of the signal processing subsystem of the Firefinder Radar system is used as an example of the use of the tools.

1.4. Background.

1.4.1. Army Maintenance Approach. One of the major goals of the Department of Defense is the implementation of a 2-level maintenance policy for next generation

electronic systems. Broadly speaking, 2-level maintenance is performed at two levels of organization or two locations, usually at the system in the field or at a depot, with essentially no maintenance performed at any other level. There are three primary prerequisites for the 2-level maintenance concept to be feasible.

a. The line replaceable units or modules should have high mean failure time and relatively low cost.

b. Fault detection and isolation must largely be a system-embedded function that is comprehensive and free of false alarms.

c. Line replaceable units or modules should be modular in design, easily accessible, and supportive of a hierarchical testability approach.

To achieve these prerequisites, a system must be implemented using highly reliable components and integration techniques, and testability/diagnosability must be considered as design requirements of equal importance to performance, size, and cost. To meet testability/diagnosability requirements in the VHSIC era, within the 2-level maintenance constraints, the system designer has to rely more on design for testability and built-in test approaches rather than external Automatic Test Equipment (ATE).

Design for testability and built-in test approaches exist at the chip and board levels. Unfortunately, while DFT techniques and BIT approaches are available for incorporation into higher levels of VHSIC system designs, this activity is not methodical and, in most cases, is separate from the functional system design process. In general, testability schemes are incorporated into the design after the functional design is completed. This practice is due partly to the lack of a unified system design for testability methodology and supporting tools.

1.4.2. Advanced CAD/E for Systems Testability (ACST). RTI developed a generic, top-down design for testability methodology that results in improved testability through judicious use of BIT and through use of design for testability at different levels of the system design abstraction. The developed design methodology is supported by a BIT implementation scheme to achieve, with high confidence, fault detection and fault isolation to a predetermined number of modules (e.g., chips, boards, or subsystems). The final report describes alternative approaches for using BIT at the board level to achieve the testability required to isolate a fault to an ambiguity group of chips. Additionally, an implementation of a board level BIT technique is also described. The proposed technique uses a special test module, designed and implemented by RTI, that performs tests on ambiguity groups with minimal outside intervention.

Finally, the proposed board level design for testability methodology was demonstrated on the AN/TPQ-36 (Firefinder) radar system. ADAS was used to obtain a hierarchical description of the doppler filter section of the signal processor to verify system functionality and to gain insight into system operation.

Testability analysis was used to determine the ambiguity group sizes of the board design under consideration, as well as the mean time to isolate a fault in these ambiguity groups. This was done assuming that tests are applied at the primary inputs of the board and the existing test points on the board are only monitoring points (i.e., not control points). In addition, fault simulations, assuming both pin level and gate level classical stuck-at faults, were performed to determine the fault coverage one can obtain using the existing design with test vectors provided by the manufacturer.

The design then was modified to meet certain ambiguity group size requirements that were lower than what the testability analysis revealed for the existing design. The new design was performed using the developed design methodology, and a BIT technique that can be implemented with the BIT module developed by RTI was recommended. The redesign approach taken demonstrated that ambiguity group size and mean time to isolate parameters that are specified as design requirements can be met when a structured DFT/BIT methodology is used. Furthermore, fault simulation results demonstrated that the fault coverage obtained with the BIT technique used during redesign was, in general, better than the results obtained when the manufacturer's test sets were used.

The design penalties, in terms of hardware and time overhead in using the proposed BIT technique, were also presented and different design configurations of the proposed test module were discussed with a goal of minimizing the design penalty for different applications with varying design requirements.

The structured design methodology provided by this effort allows the designer of Army systems to incorporate testability features into this design using a systematic approach, thus producing systems that are testable by construction. The single-chip test module developed and demonstrated by RTI can be used in a variety of board level BIT implementations of signature analysis that can cover a wide range of applications. TEA uses this module in an example implementation of a BIT technique for test pattern application and resultant data compression.

The work described here supplied the incentive and background for the work done on TEA. ACST provided the start for the TEA methodology development. During TEA, the ACST methodology was expanded and automated to meet the testability goals defined during ACST. This methodology will be described in Section 2.

1.4.3. Architecture Design and Assessment System (ADAS). A well-integrated set of tools is the key to improving system designer productivity. Tools are needed to support designers' activities, from conceptualizing the system to completing a chip.

ADAS, developed by the Center for Digital Systems Research at RTI, is an integrated set of computer-aided engineering tools which supports the synthesis and analysis of electronic systems in several ways. During synthesis, ADAS provides an interactive color graphics environment for creating and manipulating high level hardware and software designs. During analysis, ADAS provides detailed simulation results and performance analysis for verifying and evaluating the design. ADAS performs consistency

checks on both hardware and software graphs. ADAS uses both simulation-based and analytical techniques to calculate performance characteristics. ADAS supports top-down design and bottom-up analysis of systems. A common data base integrates the ADAS tool set, supporting the synthesis, analysis, and refinement of both software and hardware at high levels in the design process.

The ADAS graph data base provides the necessary attributes and structural information to support a wide range of simulation tool interfaces. These tools build functional simulators from ADAS graphs and functional module descriptions written in an associated language. A current contract supports the addition of interfaces to the VHDL tool set. For access to ADAS information from other systems the ADAS data base library provides a full set of documented data base access routines.

At the highest system levels, ADAS graphs capture an operational model of the system architecture, allow successive partitioning of software and hardware tasks, and provide performance information including software and hardware, component utilization, response times, and potential bottlenecks.

In an effort parallel with TEA, RTI has been working to integrate ADAS with the DoD's VHDL and Silicon Compiler System's Genesil software. The resulting CAD system is called the VSC or VHSIC Silicon Compiler. Together, the three components of the VHSIC Silicon Compiler support most of the activities required to design working application-specific integrated circuits.

At the highest system levels, ADAS graphs capture an operational model of the system architecture, allow successive partitioning of software and hardware tasks, and provide narrative information on system functionality. Performance information can also be obtained on component utilization and load balancing, timing, and potential bottlenecks.

Once hardware structure has been determined, modules can either be selected from the VHDL library or custom designed when necessary. At this point, VSC can provide estimates of chip power dissipation area and pinout, as well as predict system performance and functional correctness based on the selected VHDL modules.

When a satisfactory design has been achieved, the component and connection information is directed to Genesil to create complete fabrication specification information.

The design checks performed by Genesil are based on the creation of five chip models: physical layout, power dissipation, function, performance, and logic.

Once the chip layout demonstrates satisfactory behavior, a pattern generation tape is created with the fabrication process specified by the designer. At the present time, there are over twenty fabrication processes available from which designers may choose.

1.4.4. VHSIC Hardware Description Language (VHDL). VHDL has been fostered by the Department of Defense and adopted as the required hardware design system for use with VHSIC technology. VHDL provides a standardized, computer-assisted methodology to specify, verify, and archive hardware designs.

VHDL supports the design, description, and simulation of hardware from a behavioral level down to the gate level. A design environment that includes support tools such as a syntax-checking analyzer, a simulator, and a design library has been developed for the Air Force.

ADAS supports simulation in VHDL from hierarchical hardware graphs. TEA provides VHDL descriptions of BIT modules that are to be added to a design to increase testability.

1.4.5. Tester Independent Support Software System (TISSS). TISSS has a VHDL-to-fault simulation capability which TEA needs to assess the value of its recommendations. TISSS is currently coded to handle 7.2 VHDL.

2. TEA METHODOLOGY

This paragraph describes the Test Engineer's System (TEA) system, including the TEA methodology as implemented by the TEA utilities and tools.

To aid the user, Architecture Design and Assessment (ADAS) tools (e.g., EDIGRAF, the ADAS graph editor) will be shown in all capital letters, and functions within the tools (e.g., **edit**) will be shown in bold typeface.

The user is directed to the ADAS User Manual, version 2.3 and updates including the User Manual for the VHSIC Silicon Compiler System for information concerning ADAS tools besides TEA.

2.1. TEA System Overview. Utilization of the TEA system methodology and computer-aided design (CAD) tools enables testable digital hardware design to occur in parallel with system functional design and results in systems that are maintainable at a lower life-cycle cost.

Design for testability (DFT) techniques are available for incorporation into the board and subsystem levels of system designs and TEA provides a methodology and a supporting CAD system allowing the system VHSIC designer to meet predefined testability requirements. This is accomplished by supporting DFT and built-in test (BIT) techniques at all levels of design abstraction.

The methodology and tools support the concept of testable boards as building blocks of testable subsystems; these subsystems, in turn, can be used to create testable systems. The TEA methodology supports design for testability, choice of BIT technique for a board, which is also supported at the subsystem and system levels, and augmentation of the system function with predefined BIT modules to increase board testability.

Figure 2-1 depicts a system design methodology. The concurrent functional and testable design methodology addresses testability issues at all stages of systems design (i.e., preliminary, detailed, final) and at each level of the system hierarchy. As shown in Figure 2-1, the requirements analysis identifies those attributes that the system must meet. During preliminary design, hardware resources must be identified. At this stage in the system design process and in contrast to traditional design practices, test resources must also be determined. This is the area where TEA begins to have an effect. During detailed design, the specific functionality of system resources must be identified and verified through simulation and checked against system requirements, including testability. Trade-offs are made at this point to ensure that system requirements are met.

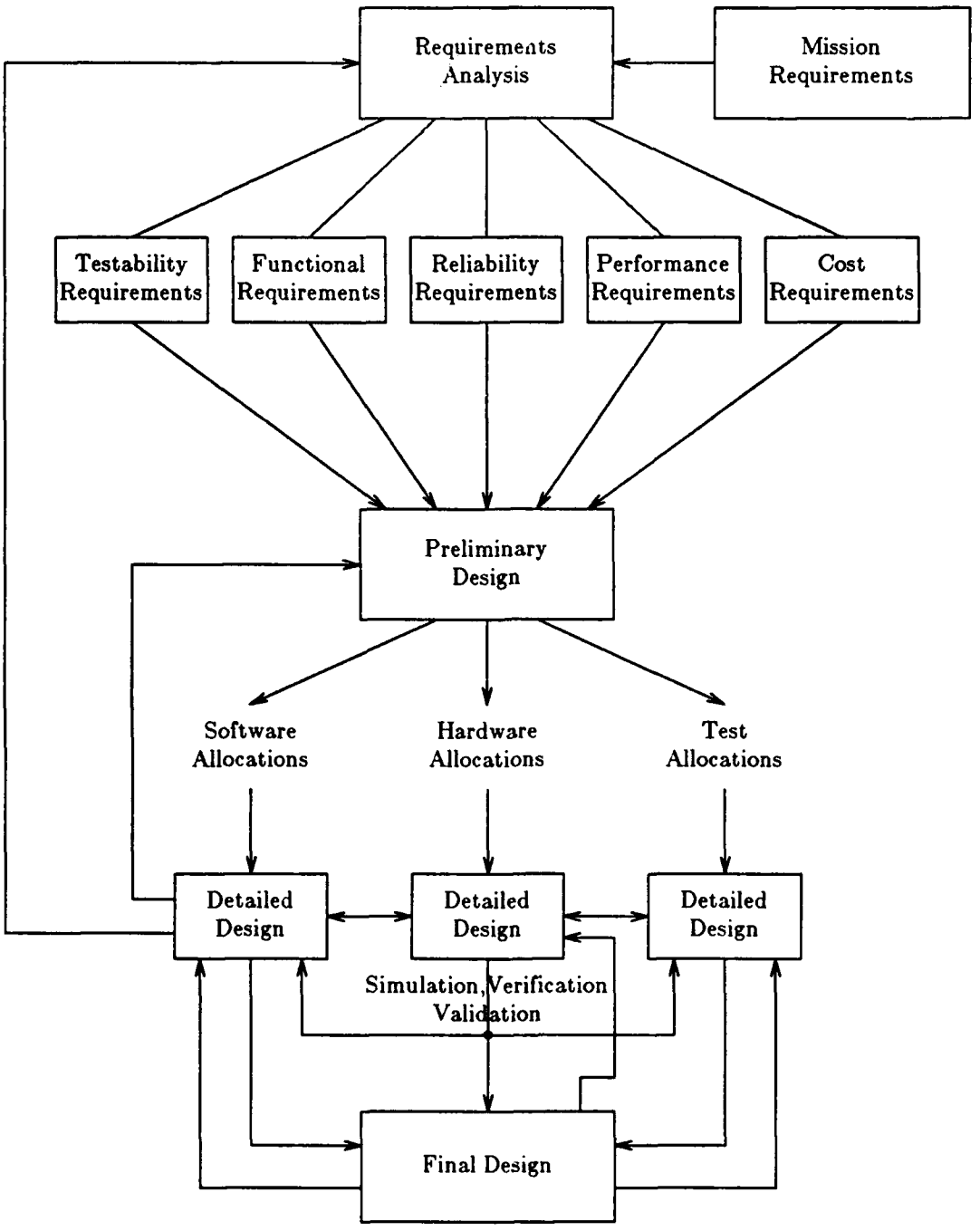


Figure 2-1. The System Design Methodology Used by TEA

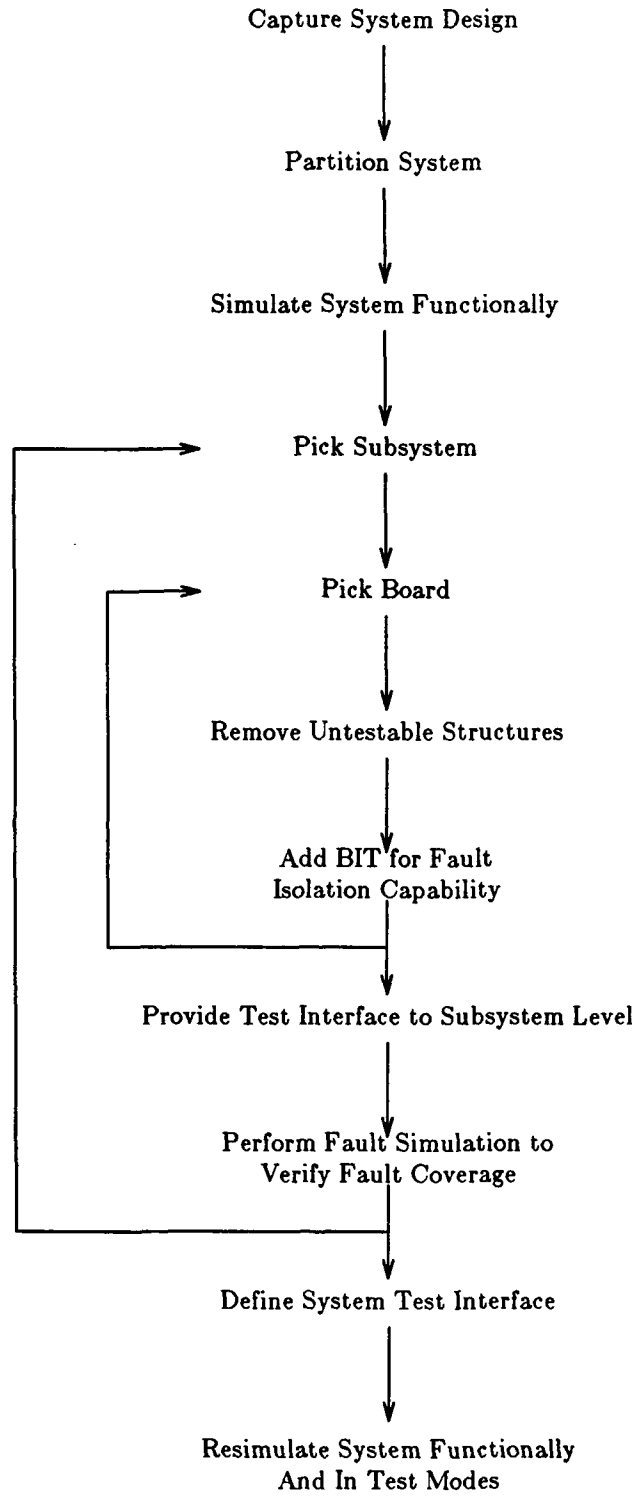


Figure 2-2. The User's Roadmap Through Detailed Design Using the TEA Tools

As noted in Figure 2-1, the TEA system is primarily used to support the detailed design phase of the methodology. Figure 2-2 shows the steps involved in using the TEA tools. This is a high-level description; however, the emphasis on design process for testable boards is shown as iterative loops in the figure. TEA system tools are shown in parentheses.

There are five newly developed tools. Design for Testability Guideline Checker identifies untestable structures and recommends alternative structures that are more testable. BIT Recommendation divides a board into ambiguity groups for fault isolation testing and recommends a class of BIT techniques for each ambiguity group. BIT Overhead Summary calculates the approximate hardware overhead (i.e., BIT support modules and additional I/O) associated with the implementation of a particular board level BIT technique. BIT Placement Recommendation generates a new schematic of the board with a sample implementation of the given technique. The three complementary BIT tools are shown in Figure 2-3. System Summary itemizes the incremental hardware overhead attributable to added testability.

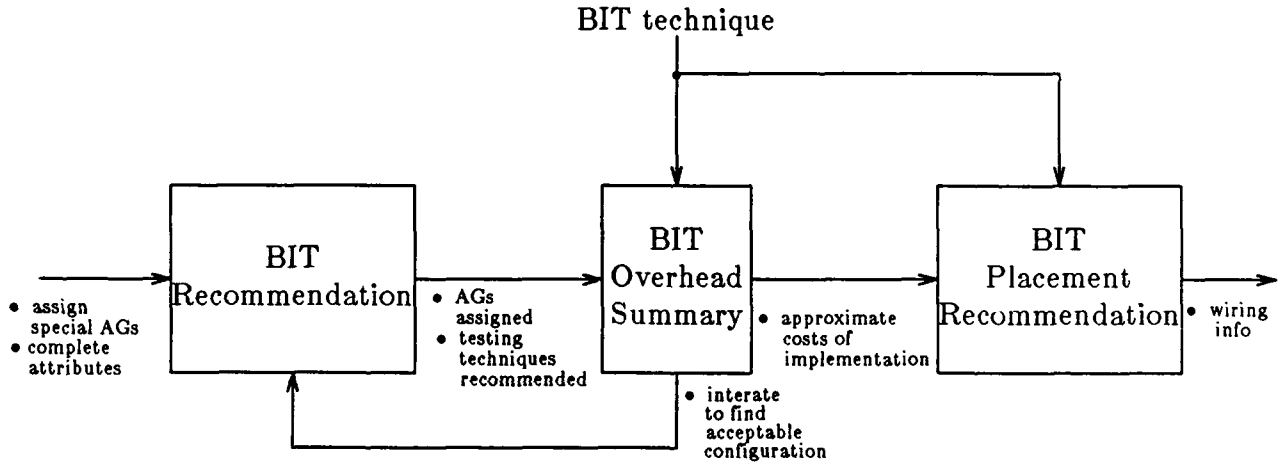


Figure 2-3. TEA BIT Recommendation Tools

The TEA user does not need to capture his design in any restricted way. TEA does not need to have each board described on an individual graph or even on graphs at the same level of ADAS hierarchy. TEA does, however, assume that "leaf" nodes are chips and makes recommendations accordingly.

Utilities enable the user to add his own guidelines for identifying untestable structures; to select special ambiguity groups; and to obtain on-line help about user functions, tool data bases, and tool outputs. TEA provides the user with a color representation or "current view" of the design and a menu-oriented user interface. These tools and utilities will be discussed in detail in Paragraphs 2.6 and 2.7, the user interface in 2.5.

The TEA system accepts an ADAS schematic of the system, which has been functionally verified using VHDL, as its input. After BIT has been added to the system using recommendations from the TEA tool, the TEA methodology supports resimulation of the system in VHDL with the added BIT features and test modes by providing VHDL descriptions of the BIT modules (see Paragraph 2.3.3) and by instructing in the use of the modules under normal and test modes.

TEA's concept of a testable system is illustrated in Figure 2-4. TEA aids in the design of testable boards such that any on-board built-in test reports to an on-board maintenance node which controls test application and reporting. To close the loop, the maintenance node communicates with a subsystem test control unit (TCU) via a subsystem test bus and a system TCU via a system test bus, which may be identical in certain applications. The system TCU may communicate with automatic test equipment (ATE) to receive inputs or process outputs.

TCU functions include

a. self-diagnosis

b. initialize and control the test of boards/subsystems

c. interpret test results

d. communicate test results to system TCU/ATE

Depending on the level of BIT implemented and fault detection/isolation capability required, automatic test equipment may or may not be needed to supplement the system level TCU. TEA recommends each of the features shown in Figure 2-4, but its primary function is leading the user towards testable board designs.

TEA handles both structured and unstructured logic within the same environment; therefore, the design can contain both unique and off-the-shelf components. TEA aids a designer, who is not necessarily a test expert, to meet a fault isolation requirement by identifying ambiguity groups (AGs) (An AG is the smallest interconnected unit to which a fault can be isolated.) which have heuristically similar test characteristics. When a fault is isolated to an AG, then all components of an AG are replaced and testing is restarted. Non-ambiguous isolation to an AG can minimize the cost and time of parts replacement.

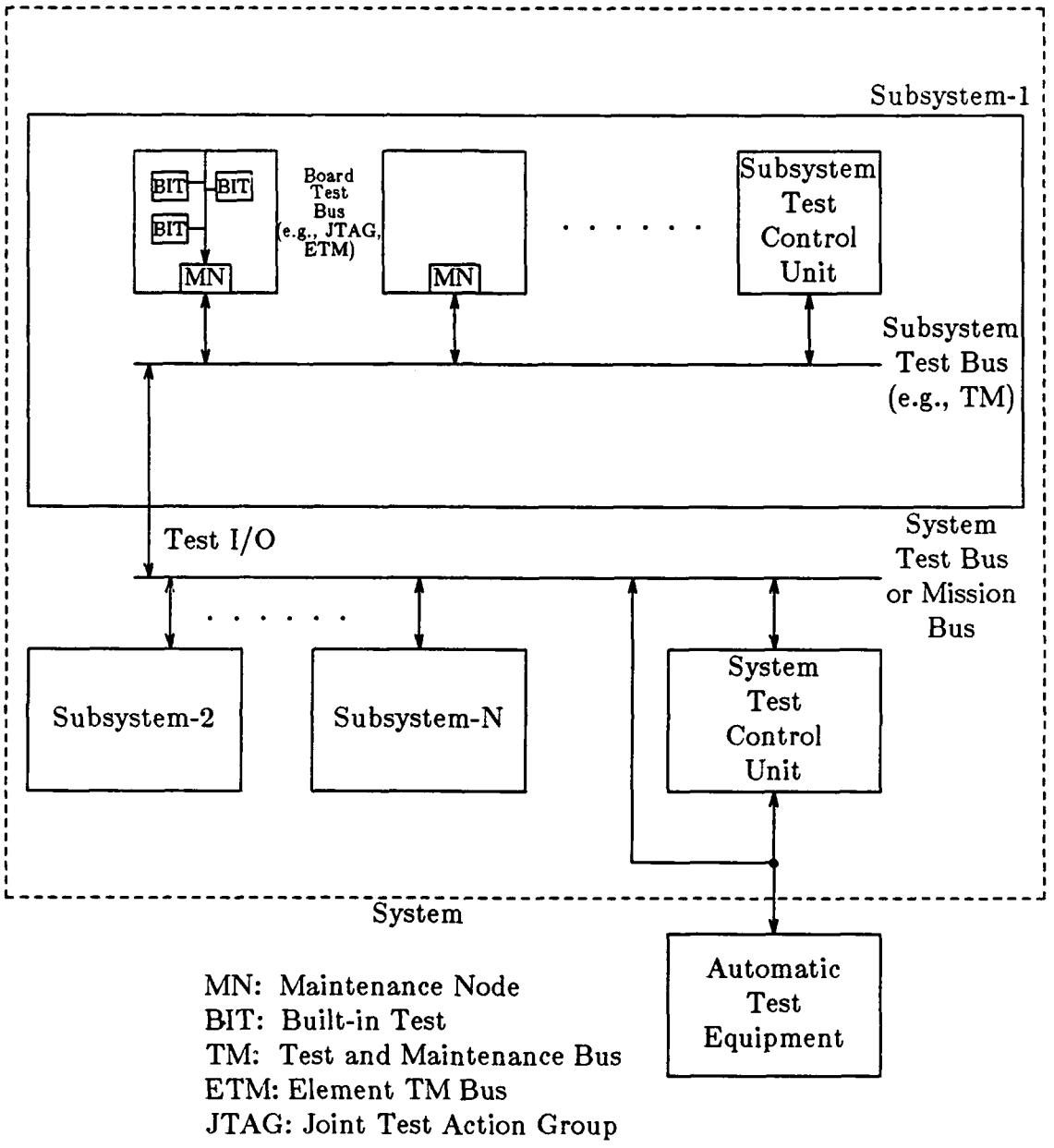


Figure 2-4. TEA's Hierarchical View of a System

2.2. The TEA Workstation and Requirements. The workstation on which TEA was developed and delivered is a DEC VAXstation II with GPX graphics. This workstation uses the DEC VMS operating system version 4.6. Some of the TEA tools output graphical information; this information is lost if the user's monitor does not support color. The DEC C compiler version 2.3 was used to prepare the code for delivery. The user needs Quintus Prolog version 2.0 to run the Design for Testability Guideline Checker and to add any guidelines to the Design for Testability Guideline data base. To run simulations using VHDL, the user must know how to use those tools. TEA supplies a BIT support module library written in both version 7.2 VHDL and 1076 VHDL. ADAS supplies a tool to obtain an ADAS template from a IEEE standard 1076 VHDL code segment, called TEMPGEN. VHDLGEN generates IEEE standard 1076 VHDL code for a simulation from IEEE standard 1076 VHDL code segments whose data flow is represented by an ADAS hardware graph.

If the user does not need to change any of TEA, then he just needs the machine, operating system, compiled TEA code, and Prolog. If the user needs to produce large ADAS-VHDL simulations of his system, a larger machine is recommended for that portion of the development. If speed of operation is a concern, a larger VAX could be used for the C-oriented portions of TEA (This excludes the Design for Testability Guideline Checker, which is written in Prolog.).

The user of TEA is not expected to be a testing expert, but is assumed to be a system and/or board designer competent with ADAS and its data base structure. A working knowledge of EDIGRAF is necessary to capture the design for use with TEA. The EDIGRAF supplied with TEA is different from the EDIGRAF commercially available currently with ADAS. TEA's EDIGRAF allows buses and 256 inports/outports/biports per node. The user is directed to RTI to obtain more information about these features. TEA ignores biports, but uses buses. The user is strongly encouraged to use ADAS buses instead of fanout nodes to show a signal propagating to multiple destinations. In general, TEA will give unpredictable results if fanout nodes are used. A message, is given at the start up of TEA if fanout nodes are detected.

******Warning
******This graph contains fanout nodes.
******TEA may generate unpredictable results.
******Buses are recommended for replacement of fanout nodes.

2.3. Inputs to TEA. The user communicates the features and connectivity mapping of his design by capturing the schematic in ADAS and by associating appropriate attributes to each node in the description. Nearly all information needed by TEA is held in the ADAS data base. Exceptions are noted in Paragraph 2.3.4.

2.3.1. ADAS Hardware Graphs. Just as other ADAS tools operate on a graphical representation of the user's system, called a graph, TEA needs a set of hierarchical hardware graphs from which to draw conclusions about the testability of the system. TEA is primarily concerned with digital circuit boards, but there is no inherent reason why the design for testability guidelines (Paragraph 3.2) could not be extended to the non-digital testability arena, if graph patterns and guidelines could be identified. Only synchronous digital BIT techniques are supplied with TEA.

To identify the partitioning of systems into subsystems and boards, two attributes are assigned to each node, **Tsubsystem** and **Tboard**. An alphanumeric string is expected in each of these to uniquely identify each node with a particular board. All nodes with the same values for **Tsubsystem** and **Tboard** are assumed to reside on the same board and should be considered together for identifying a coherent testability method for the board.

Other information about a node, which is normally a representation for a chip at the lowest level of description, is communicated via the node and port attributes.

2.3.2. ADAS Data Base Attributes. Each node, bus, and arc in an ADAS graph has attributes attached to it. Some obvious attributes include color, width, and height.

TEA has a unique set of node and arc attributes not used by other ADAS tools. These are listed here along with a table of particular attribute values and their associated meaning to the TEA tools. These values are very specific, but the case is ignored. All TEA attributes begin with a capital 'T'.

To make these attributes "visible" to EDIGRAF and TEA, the save status must be set to "S" using the **edit** command. Not all TEA tools need each of the attributes, but all are necessary for the accurate completion of the methodology. There is a checker run at the initiation of TEA to make sure the attributes are in the data base (they do not have to have values).

Note: Each node should have been assigned to a uniquely named board and subsystem. There is no check for this, but board names must all be unique -- even across subsystems. One of the tools to be described below can be called on multiple subsystems simultaneously and if two boards are named with the same name, an unresolved command interpreter conflict arises and unexpected results occur. Identical subsystem names result in the tool understanding that there is only one subsystem (with that name) and all boards assigned to it belong to the one subsystem. Board and/or subsystem names can not be left blank, as "null" is not a valid value (it would be impractical to select a "" from the menu). The board and subsystem names of leaf nodes have precedence over internal nodes. TEA supports the concept of ADAS buses, but

all buses are assumed to be a single bit in width. Most of the TEA tools work on "flattened" graphs. Hierarchy is lost and "fanout" node (and bus) connectivity becomes transparent. (Again, fanout nodes are not recommended.) TEA ignores biports.

Further notes on buses: since buses are a new feature to even the most expert ADAS users, a few points need to be made. An ADAS bus does not represent a protocol or directionality. An ADAS bus is a way to connect multiple inports/outports/biports of multiple nodes together. For instance, a "clear" signal generated by one node can be "bussed" to every flip-flop on the board without the need for the outdated "fanout" node concept. Each instance receives an identical copy of the original signal.

ADAS buses can have multi-bit widths, but TEA assumes that a bus represents a single bit.

ADAS buses *must* have at least one input. If a bus has no input, it is ignored, and TEA may give inaccurate results.

For accurate results, TEA users should label any pertinent arc attributes on the arcs that provide input to the bus. For example, if a test point is provided on a bus, an input arc to the bus should have its **Tdft_io** attribute set to test_point. If only output or sink arcs are labeled, TEA may give inaccurate results. If two or more input or source arcs to a bus are primary inputs, they should each have their **Tdft_io** set to primary input.

An ADAS bus cannot be directly connected to another ADAS bus. Watch out for this when subgraphs are created.

Table 2-1 shows all TEA attributes. Table 2-2 shows valid syntax for the TEA attributes.

Table 2-1. Meaning of Each TEA Attribute

Node attribute	Meaning
Tag_name	set by BRT, name of the AG to which this chip belongs
Tag_test	set by BRT, AG test scheme
Tag_test_time	time or test vectors needed to test this AG
Tasynch	does the user intentionally use this chip asynchronously?
Tbit	is this a node that contributes to the BIT?
Tboard	to which board does the node belong?
Tchip_maxfan	max lines to which any output can be connected
Tconfig	set by BRT, holds name of specific scan configuration
Tcount_bits	how many bits in this counter?
Tdevice_type	analog, digital, mix
Tfault_cov	user-provided fault coverage for **Ttest_len** vectors
Tgroup	set by BRT, node grouping scheme
Thw_module	number of gates of node
Tlogic_family	to what logic family is the node's I/O compatible?
Tlogic_type	combinational, sequential

Tnode_spec_type	specific device type
Tnode_type	general type of node
Tscan	is this chip scannable?
Tselftest	is this chip self testable?
Tsp_ag_name	name of the "special" AG to which this chip belongs
Tstates	number of single-bit memory, not including RAM/ROM
Tsubsystem	to which subsystem does the node belong?
Ttest	set by BRT, node test scheme
Ttest_len	number of user-provided test vectors
Ttri_state	is chip tristateable?

Arc attribute	Meaning
Tarc_type	data/control/clock
Tdft_io	is this a primary I/O or test point?
Ta_d	is this an analog or digital signal?

AG: ambiguity group
BRT: Bit Recommendation Tool

Table 2-2. Valid Syntaxes for Each Attribute

Node attribute	Syntax
Tag_name	Alphanumeric
Tag_test	Alphanumeric
Tag_test_time	Integer Default = 0
Tasynch	Alphanumeric Affirmative = async, asynch, asynchronous, y, yes
Tbit	Alphanumeric BIT scheme = ps,dt Maintenance Node = mn, m_n, maintenance BIT Support Node = bit_dt, bit_ps Node with BIT = deterministic, dt, ps, pseudorandom scan support modules = bit, yes
Tboard	Alphanumeric
Tchip_maxf·n	Integer Default = 0
Tconfig	Alphanumeric
Tcount_bits	Integer Default = 0
Tdevice_type	Alphanumeric Recognized options = a, analog, d, digital, mix

	Default = digital
Tfault_cov	Integer (1 - 100) Special case = 100 means "adequate coverage" Default = 0
Tgroup	Alphanumeric
Thw_module	Integer
Tlogic_family	Alphanumeric Recognized options = cmos, ecl, gaas, iil, nmos, ttl, other
Tlogic_type	Alphanumeric Recognized options = comb, comb_logic, combinational Default = sequential
Tnode_spec_type	Alphanumeric
Tnode_type	Alphanumeric
Tscan	Alphanumeric Recognized options = etm, etm_sup, jtag, jtag_sup, vhsic, vhsic_sup, scan, scannable, y, yes
Tselftest	Alphanumeric Affirmative = bist, bit, self, selftest, vhsic, y, yes
Tsp_ag_name	Alphanumeric
Tstates	Integer Default = 0
Tsubsystem	Alphanumeric
Ttest	Alphanumeric
Ttest_len	Integer Default = 0
Ttri_state	Alphanumeric Affirmative = tri, tristate, y, yes Default = not tristateable

Arc attribute	Syntax
Tarc_type	Alphanumeric Recognized options = clk, clk_gen, clock, clock_generator Recognized options = cntrl, control, ctrl, d, data Default = data
Tdft_io	Alphanumeric Recognized options = pi, po, primary_input, primary_output Recognized options = tp, test_point Recognized options = tpo, test_point_output, test_pt_output
Ta_d	Alphanumeric Recognized options = a, analog, d, digital Default = digital

In EDIGRAF or TEA, the **edit** command will allow the user to view and/or change the value for an attribute. **Stats** on a node will allow the user to view attribute values.

Besides connectivity, TEA uses attribute values to decide how to handle a node in a given situation. Attributes provide valuable insight into the function and purpose of a node. Since TEA tries to evaluate schematics from just this topological information, completing attribute values accurately is extremely important for accurate tool results. ***If attributes are not assigned, not assigned completely, or not assigned with proper syntax, the tools will, in general, give unpredictable results.*** Following are lists of attributes that should be completed for acceptable results from the given tools.

Table 2-3. Attributes and Syntax Needed for Design for Testability Guideline Checker

Node Attribute	Rule	Meaning
inport_id	g1_01	clear, clr, pre, preset
	g1_02	clear, clr, pre, preset
	g1_03	dsel, dselect, strobe
	g1_04	tctrl, trictrl
	g1_05	enable, hold
	g1_06	add_latch_en, address_latch_enable, ale, chip_sel, chip_select
	g1_06	cs, rd, read, read_write, readwrite, rw, wr, write
	g1_07	bus, bus_control, bus_ctrl, bus_node, busnode, ctrl, cntrl, control
	g1_14	clk, clk_gen, clock, clock_generator
	g2_03	clock, clk, clk_gen, clock_generator
node_name	all rules	alphanumeric
outport_id	all rules	alphanumeric
	g2_01	clk, clk_gen, clock, clock_generator
	g2_03	carry, carry_out, co
Tboard	all rules	alphanumeric
Tchip_maxfan	g2_05	integer
Tcount_bits	g2_03	integer
Tdevice_type	g1_12	a, analog, d, digital
	g1_13	a, analog, d, digital
Tlogic_family	g2_04	ttl, iil, ecl, cmos, nmos, gaas, other
Tlogic_type	g1_07	combinational, comb, comb_logic
	g1_08	combinational, comb, comb_logic
Tnode_spec_type	g1_09	alphanumeric
Tnode_type	g1_01	count, counter, ctr, ff, flip_flop, flipflop,
	g1_01	reg, register, shift_register, shiftregister, sr
	g1_02	counter, ctr, ff, flip_flop, flipflop,

	g1_02	shift_register, shiftregister. sr, reg, register
	g1_03	multiplex, multiplexer, multiplexor, mux
	g1_05	microprocessor, u_proc, up, uproc, processor
	g1_06	dram, fifo, mem, memory, prom, sram, ram, rom
	g1_07	bus, bus_control, bus_ctrl, bus_node, busnode, ctrl, cntrl, control
	g1_10	fanout, copy, null
	g1_13	da, data_acquisition, mix
	g1_14	clk, clk_gen, clock, clock_generator
	g1_16	one_shot, oneshot
	g2_01	clk, clk_gen, clock, clock_generator
	g2_03	counter, ctr, count
	g2_05	fanout, copy, null
	g2_06	dram
Tsubsystem	all rules	alphanumeric
Ttri_state	g1_04	tri, tristate, y, yes
	g1_11	tristate, tri, y, yes
Tasych	g1_14	asynchronous, y, yes, async, asynch

Arc Attribute	Rule	Syntax
Tarc_type	g1_14	clk, clk_gen, clock, clock_generator
Ta_d	g1_12	a, analog, d, digital
	g1_13	a, analog, d, digital

Table 2-4. Attributes Used by Design for Testability Guideline Checker to Determine Primary Input Controllability

Node Attribute	Syntax
Tnodetype	scan, scannable, scan_reg, scan_register, hi, lo, gnd, one, vdd, vcc, vcs, vss, zero, ground, constant
Tlogic_type	comb, combinational, comb_logic
Tboard	alphanumeric
Tsubsystem	alphanumeric

Arc Attribute	Syntax
Tarc_type	d, data, ctrl, cntrl, control
Tdft_io	pi, primary_input, po, primary_output, tp, tpo, test_point, test_pt_output, test_point_output

Table 2-5. Attributes Used by Design for Testability Guideline Checker to Determine Primary Output Observability

Node Attribute	Syntax
Tboard	alphanumeric
Tlogic_type	comb, combinational, comb_logic
Tnodetype	scan, scannable, scan_reg, scan_register, ff, flip_flop, flipflop, ctr, count, counter, reg, register, shift_register, shiftregister, sr
Tsubsystem	alphanumeric

Arc Attribute	Syntax
Tarc_type	d, data, ctrl, cntrl, control
Tdft_io	pi, primary_input, po, primary_output, tp, tpo, test_point, test_pt_output, test_point_output

Table 2-6. Node Attributes Set by BIT Recommendation Tool (BRT)

Attribute	Meaning
Tag_name	set by BRT, name of the AG to which this chip belongs
Tag_test	set by BRT, AG test scheme
Tconfig	set by BRT, holds name of specific scan configuration
Tgroup	set by BRT, node grouping scheme
Ttest	set by BRT, node test scheme

Table 2-7. Attributes Needed
for BIT Recommendation Tool (BRT)

Node Attribute	Meaning
inport_id	label for the input port of an ADAS node
output_id	label for the output port of an ADAS node
node_name	name of ADAS node
Tasynch	does the user intentionally use this chip asynchronously?
Tbit	is this a node that contributes to the BIT?
Tboard	to which board does the node belong
Tdevice_type	analog, digital, mix
Tfault_cov	user-provided fault coverage for **Ttest_len** vectors
Thw_module	number of gates of node
Tlogic_family	to what logic family is the node's I/O compatible?
Tlogic_type	combinational, sequential
Tnode_spec_type	specific device type
Tnode_type	general type of node
Tscan	is this chip scannable?
Tselftest	is this chip self testable?
Tsp_ag_name	name of the "special" AG to which this chip belongs
Tstates	number of single-bit memory, not including RAM/ROM
Tsubsystem	to which subsystem does the node belong?
Ttest_len	number of user-provided test vectors

Arc Attribute	Meaning
Tarc_type	data/control/clock
Tdft_io	is this a primary I/O or test point?
Ta_d	is this an analog or digital signal?

Table 2-8. Attributes Needed for BIT
Overhead Summary and BIT Placement Recommendation

Node Attribute	Meaning
node_name	name of ADAS node
Tag_name	set by BRT, name of the AG to which this chip belongs
Tboard	to which board does the node belong
Tnode_type	general type of node
Tsp_ag_name	name of the "special" AG to which this chip belongs
Tsubsystem	to which subsystem does the node belong?

Arc Attribute	Meaning
Tdft_io	is this a primary I/O or test point?

Table 2-9. Attributes Needed for System Summary

Node Attribute	Meaning
node_name	name of ADAS node
Tag_name	set by BRT, name of the AG to which this chip belongs
Tag_test_time	time or test vectors needed to test this AG
Tbit	is this a node that contributes to the BIT?
Tboard	to which board does the node belong
Tfault_cov	user-provided fault coverage for **Ttest_len** vectors
Tsp_ag_name	name of the "special" AG to which this chip belongs
Tsubsystem	to which subsystem does the node belong?
Ttest_len	number of user-provided test vectors

Arc Attribute	Meaning
Tarc_type	data/control/clock
Tdft_io	is this a primary I/O or test point?
Ta_d	is this an analog or digital signal?

2.3.3. VHDL Descriptions of ADAS Nodes. ADAS provides a facility for functional simulation of a system using VHDL. ADAS will provide the VHDL code of an entire system given the code segment for each node/module in a graph and the graph describing the connectivity of the modules. VHDLGEN uses attribute **entity_name** to find the entity description for a node and **arch_name** for the architecture body for that node.

TEMPGEN, another ADAS utility, creates an ADAS node template from a VHDL code segment. An entity description file is used as input and a hardware template data base is created. EDIGRAF's **merge** function can be used to get all the necessary hardware templates into a single template data base for TEA.

The TEA methodology encourages users to have a functionally correct description of their system before entering TEA so that when TEA has completed its task of suggesting changes to improve testability the user has a functionally correct target to benchmark. TEA provides a library of built-in test support modules, including pseudorandom test pattern generators, built-in logic block observers, and scannable registers, that could be placed in an ADAS hardware graph to implement a particular technique. The BIT tools know about these modules and use them to implement the BIT techniques which are known to the system. So that the user can resimulate his system for functional correctness, TEA provides VHDL code segments and ADAS templates for

each of these modules.

This manual is not intended to instruct in the use of VHDL or ADAS utilities other than those used in TEA. Other supplemental reference materials are required for this.

2.3.4. User Information via VMS Logicals. There are two very specific instances where additional information is provided to TEA outside the ADAS data base.

When the user wants to add his own design for testability guidelines to those known by the Design for Testability Guideline Checker (Paragraph 2.7.1), he must set a logical, **tea_rules**, to point to the file that contains the names of the available guidelines, one per line.

When the user wants to provide information about specific integrated circuits for the BIT Recommendation (Paragraph 2.7.2) tool to use in making decisions about how to group chips into ambiguity groups, he must set a logical, **tea_brt**, to point to a file that contains the name of the file where the library resides. The format and further information concerning this file is found in Paragraph 2.7.2.

2.4. TEA Output. TEA is a knowledge-based system that tries to match graphs against patterns known to inhibit testability. TEA will not try to work "behind the user's back" and change the user's schematic because an arbitrary pattern was matched. Rather, TEA will inform the user via output reports that certain configurations had been noted in the schematic and that the user has the option to do something about these features so that testability will be increased.

In later paragraphs, the specific tools will be described along with the output reports because this is TEA's primary means of disseminating information to the user. This is a design feature that affords the user flexibility and input into the solution of any testability inhibitors. TEA is not designed to be an arbitrator, but as a checklist for adherence to known "good" testability guidelines. It is hoped that this method of guidance will encourage cleverness on the part of the designer/test engineer rather than strict conformity to rules.

The only tool that has an option to change a user's graphs is the BIT Placement Recommendation tool, and then no change will be made to any graph except at the option of the user. BIT Placement Recommendation will add BIT support modules to a user's design to support an implementation of an on-board BIT technique.

This philosophy applies not only to the graphical version of the schematic, but to the data base. TEA will only change data base attributes as necessary to communicate from one tool to another. For instance, BIT Recommendation will fill in the node attributes that indicate the ambiguity group to which the node has been assigned and how that ambiguity group is to be tested.

TEA is designed to keep the user as well informed as possible. The on-line help information system is driven by the "position" of the user in the menu hierarchy. This is TEA's attempt to "keep a finger in the right place in the manual" for the user. The

written documentation is designed to supplement and expand on the on-line information.

2.5. The TEA User Interface. TEA provides the user with a color schematic or "current view" of the design and a menu-oriented interactive user interface, which is identical to the ADAS user interface. The user is prompted for choices at each step of the process and the user either chooses menu options from the graphics display with the mouse or types in his selection (only as many letters as necessary to uniquely identify the selection) at his keyboard. Many of the menu selection items on the TEA top-level menu are identical to EDIGRAF options. Refer to the Software System User Manual for TEA for more details on menu options.

As in ADAS, TEA users communicate with the data base through tools and utilities attached to the user interface. The user interface formats all requests and inquiries of the data base and receives all responses from the data base. As the user answers inquiries, the user is prompted for more information needed to invoke a tool or, finally the tool is invoked. When an action has been completed, process control is returned to the top-level TEA menu and another action can be initiated.

2.6. The TEA Utilities. Utilities enable the user to view and print output reports, to select special ambiguity groups, and to obtain on-line help about user functions, tool data bases, and tool outputs.

For the following discussion, menu selections and attributes (start with "T") are shown in boldface type. Unless otherwise stated, program control is returned to the user interface and the TEA top-level menu at the completion of the program's execution.

2.6.1. Ambiguity Group Names (ag_name). The Ambiguity Group Names (**ag_name**) TEA utility has three functions:

a. to check connectivity of ambiguity groupings.

b. to access the ambiguity group name of any node as input to the BIT Overhead Summary and BIT Placement Recommendation tools. The user may view and edit these groups.

c. to maintain a list of special ambiguity groups as input to the BIT Recommendation, BIT Overhead Summary, and BIT Placement Recommendation tools. The user may view, add, and edit these groups.

2.6.1.1. Connectivity Check (conn_check).

Conn_check is a validity check for ambiguity groups. All nodes in a valid ambiguity group must be directly connected. **Conn_check** is automatically called by both **reg_ag** and **spcl_ag**.

If nodes in an ambiguity group are not connected, then it will be very difficult to test those nodes as a group, and it will be extremely difficult to isolate faults to the group. If the user finds that the groups he has selected are unconnected, he should consider multiple groups or consider adding intermediary nodes to the group to maintain connectivity.

This utility should be run each time the user alters the grouping of a board's nodes. The BIT Recommendation Tool will only recommend groupings of nodes connected via data paths.

Inputs.

The user inputs the board and subsystem of interest via the user interface.

Processing.

This paragraph contains a high-level description of the process called **conn_check**.

```
get nodes on Tboard and Tsubsystem
find all ambiguity groups
check connectivity of each group
output message
if ok
        issue "no error" message
if not ok
        issue "error" message
```

Limitations.

ADAS data base graphs must have been previously created before TEA is entered.

Biports are ignored.

TEA supports the concept of ADAS buses, but all buses are assumed to be a single bit in width.

Conn_check works on a "flattened" graph so attributes set on non-leaf nodes will be ignored. Attributes assigned to "fanout" nodes will also be disregarded.

Outputs.

Conn_check will issue a message to indicate whether it found errors or not.

Conn_check will create an output report using the report generator. The following is a sample output report from **conn_check**. The standard header is shown here, but will not be repeated in further example output reports for the sake of brevity.

The standard header shows the date and time of creation, the user, the graph name, the data base filename, and the output report name. The body of the report shows that an error in choosing ambiguity group "foo" has occurred. The user needs to select nodes which are connected by some direct path to form an ambiguity group.

Test Engineer's Assistant (TEA) Output Report

ADAS/TEA Version: PROTO.TYPE
Research Triangle Institute
P.O. Box 12194
Research Triangle Park, NC 27709

Date/Time: Thu Oct 15 09:04:35 1987

User: JJH

Graph:
Current View Graph: Avionic_System_Hardware
Current View Filename: system.hwg

Output Report Filename = ag_name01.rpt

AMBIGUITY GROUP NAMES
(ag_name conn_check)

AG 'foo' containing the following nodes is not connected
n3
n10

ag_name conn_check completed

Error Message/Action.

TEA will allow the user to continue even if unconnected groups exist. The user should carefully consider the results of this action.

Additional Information.

See paragraph describing

a. report generation (Paragraph 2.6.2)

2.6.1.2. Regular Ambiguity Groups (reg_ag).

The purpose of **reg_ag** is to allow the user to

a. conveniently view nodes which are in regular ambiguity groups

b. edit the **Tag_name** of any node assigned to a regular ambiguity group

The **Tag_name** attributes are assigned by the BIT Recommendation; **reg_ag** influences the BIT Overhead Summary and BIT Placement Recommendation tools by allowing changes to the output of BIT Recommendation. The user may assign nodes from special ambiguity groups to regular ambiguity groups, further affecting the output of BIT Recommendation. **Conn_check** is automatically called at the completion

of **reg_ag**.

Inputs.

The board and subsystem are input to **reg_ag**. The user must input the ambiguity group to be edited. The next menu shows all nodes on the board. Those that are currently assigned to the regular ambiguity group are shown in one color (yellow), ones assigned to special ambiguity groups are shown in another (magenta), those assigned to other regular ambiguity groups are shown in a third (orange), and all others are shown in white.

Each ambiguity group on a board has a unique name. If the ambiguity group is "regular" (i.e., assigned by BIT Recommendation), the name is stored in **Tag_name**. If the ambiguity group is "special" (i.e., assigned by the user), the name is stored in **Tsp_ag_name**.

The **Tag_name** attribute of each node assigned to an ambiguity group is updated to reflect the ambiguity group name. If a special ambiguity group node is switched to a regular ambiguity group, the **Tsp_ag_name** attribute is cleared and the **Tag_name** is set appropriately.

Processing.

This paragraph contains a high-level description of the process called **reg_ag**.

```
get_user_input (Tsubsystem) = TSUBSYSTEM
get_user_input (Tboard) = TBOARD
display Tag_names of nodes on TSUBSYSTEM and TBOARD
get_user_input (Tag_name) = TAG_NAME
display nodes with TAG_NAME in color1
display nodes with any Tsp_ag_name in color2
display other nodes in color3
while get_user_input (node_name) = NODE_ID
        if color (NODE_ID) = color1
                color (NODE_ID) = color3
        color node NODE_ID in color1
        if get_user_input (done)
                while nodes that are color1
                        Tag_name (node_name) = TAG_NAME
                        if node_name (Tsp_ag_name)
                        Tsp_ag_name (node_name) = null
                        exit
                        endif
                        endwhile
                endif
endwhile.
conn_check.
```

Limitations.

ADAS data base graphs must have been created prior to entering TEA.

Ambiguity groupings are required to have unique names on any one board. If boards are divided among graphs, then **reg_ag** will have to be executed once per graph to

fully itemize all regular ambiguity groups.

The user may not *add* any new ambiguity groups with **reg_ag**.

Nodes that are part of a special ambiguity group cannot also belong to a regular ambiguity group. **Tsp_ag_name** gets cleared.

Only one ambiguity group may be edited or viewed with a call to **ag_name**. If the user needs to edit or view multiple groups, multiple calls must be made.

Outputs.

The **Tsp_ag_name** and **Tag_name** attributes are updated.

The following is a brief report that shows that **conn_check** is run after **reg_ag** is run to make sure that the user has not destroyed the connectivity that needs to be present for appropriate addition of a BIT technique. A report will only be generated if **conn_check** finds a connectivity error. Here, ambiguity group "ag1" has become disconnected. Nodes assigned to "ag1" are bit1, FF1, CTR1, bit3, and CTR2.

```
***Insert STANDARD HEADER here***

AMBIGUITY GROUP NAMES
(ag_name conn_check)

AG 'ag1' containing the following nodes is not connected
                bit1
                FF1
                CTR1
                bit3
                CTR2

ag_name conn_check completed
```

Error Message/Action.

Additional Information.

See paragraphs describing

a. data base attributes and valid syntaxes (Paragraph 2.3.2)

b. **conn_check** (Paragraph 2.6.1.1)

c. output report header information (Paragraph 2.6.2.1)

2.6.1.3. Special Ambiguity Groups (spcl_ag).

The purpose of having special ambiguity groupings is to influence the BIT Recommendation, BIT Overhead Summary, and BIT Placement Recommendation tools. The user has control over these special groupings and can change them using **spcl_ag** to see how these groups affect the outputs of these tools.

The user introduces special ambiguity groups because he has nodes that he wants to test as a group. These can arise because of computed test vectors or error analysis. The designer may have unique insight gathered from a previous design, or he may be more comfortable with predetermined groupings.

When the BIT Recommendation Tool recommends groupings, function is not considered very significantly. As an example of a special ambiguity group, the user may wish to combine address decoders with input latches as a standard practice. This is done using **ag_name spcl_ag** before running the BIT Recommendation Tool.

The purpose of **spcl_ag** is to

a. allow the user to conveniently view nodes in specific special ambiguity groups

b. add special ambiguity groups

c. edit the **Tsp_ag_name** of any node assigned to a special ambiguity group. Note that the output of BIT Recommendation is invalidated if the user makes changes with **spcl_ag**. The user may assign nodes from special ambiguity groups to regular ambiguity groups, thus further affecting the output of BIT Overhead Summary.

Inputs.

The user inputs the board and subsystem names. If the user wishes to add nodes to a new group, the user will have to signal the tool with the name of the new group. The next menu shows all nodes on the board. Those that are currently assigned to the special ambiguity group are shown in one color (violet), ones that are assigned to any other special ambiguity groups are shown in another (magenta), ones assigned to regular ambiguity groups are shown in a third (orange), and all others are shown in white.

The **Tsp_ag_name** attribute of each node assigned to an ambiguity group is updated to reflect the special ambiguity group name. If a regular ambiguity group node is switched to a special ambiguity group, the **Tag_name** attribute is cleared and the **Tsp_ag_name** is set appropriately. **Conn_check** is run to make sure that all ambiguity groups are connected.

Processing.

This paragraph contains a high-level description of the process called **spcl_ag**.

```
get_user_input (Tsubsystem) = TSUBSYSTEM
get_user_input (Tboard) = TBOARD
display Tsp_ag_names of nodes on TSUBSYSTEM and TBOARD
get_user_input (Tsp_ag_name) = TSP_TAG_NAME
display nodes with TSP_TAG_NAME in color1
display nodes with any Tsp_ag_name in color2
display other nodes in color3
while get_user_input (node_name) = NODE_ID
        if color (NODE_ID) = color1
                color (NODE_ID) = color3
        color node NODE_ID in color1
        if get_user_input (done)
                while nodes that are color1
```

```
                                    Tsp_ag_name (node_name) = SPTAGNAME
                                    if node_name (Tag_name)
                                    Tag_name (node_name) = null
                                    exit
                                    endif
                                    endwhile
                            endif
                    endwhile.
                    conn_check.
```

Limitations.

ADAS data base graphs must have been created prior to entering the TEA.

Ambiguity groupings are required to have unique names on any one board. If boards are divided among graphs, **spcl_ag** will have to be executed once per graph to fully itemize all special ambiguity groups.

Nodes that are part of a special ambiguity group cannot also be part of a regular ambiguity group. **Tsp_ag_name** get cleared.

Biports are ignored.

TEA supports the concept of ADAS buses, but all buses are assumed to be a single bit in width.

Only one ambiguity group may be edited or viewed with a call to **ag_name**. If the user needs to edit or view multiple groups, multiple calls must be made.

Outputs.

The **Tsp_ag_name** and **Tag_name** attributes are updated.

Error Message/Action.

Additional Information.

See paragraphs describing

a. **conn_check** (Paragraph 2.6.1.1)

2.6.2. Report Generation/Access (log file). Output from TEA is available from the "Report Generation/Access" routine. Each TEA tool generates a slightly different form of output during execution. The purpose of the report generator is to collect all significant data and generate a useful output report. **Dft** is different from the other tools, as it creates an intermediate file and the report generator transforms it into a "normal" output file. The **log_file** (report access) option of the TEA top-level menu is used to either view the report or print the report on a local printer.

TEA reports are created in files with extensions

a. rpt

b. dwg

c. help

depending on the tool and contents. The report access routine will only be looking for files with these extensions.

2.6.2.1. Report Generation. This is a unique TEA utility because it cannot be accessed from any menu; it is automatically called at the end of each tool run.

Inputs.

The "current view" ADAS data base, operating system variables, output from a TEA tool, and the TEA menu data bases are the inputs to report generator. The "current view" data base and TEA menu data base provide intermediate information to the report generator.

Information from the operating system (e.g., user's name), needed for the output report header, is obtained directly from the operating system.

The *results.tmp* file contains all information that identifies violations found by **dft**. The format has been defined such that the TEA report generator, written in C, can access the results of the **dft** Prolog code. This file is removed by the report generator after it has been transformed into a "normal" output report.

Each **dft** guideline executed inserts a predefined header to identify to the report generator which guideline violations are to follow. This is represented in the file as '*g1_02*'. The violations are then listed in one of two ways

a. single node violations,

b. list of node violations.

A single node violation is represented in the *results.tmp* file as a two line entry. The first line is the standard guideline header, '*g1_02*', and the second line identifies the specific node in violation of the guideline along with the specific type of violation, the port, and any other specific node information required to generate the output report.

A list of violations is represented in the same way as single violations, however, multiple nodes are listed after the guideline header instead of one.

In addition, information for the output report is contained within the *results.tmp* file which is not part of the violation. These are lists of nodes which represent a certain type or function. The information lists are represented by unique header, such as '*g1_02_ff_ctr_sr_list*', on a single line, followed by the list of nodes. Each node in the list is on a separate line and enclosed in asterisks so that the node is not highlighted, like nodes involved in violations.

A sample *results.tmp* file:

```
*DFT_Analyze*
*g1_02*
*g1_02*
U22 r1 port=7 clear
*g1_02*
U24 r1 port=7 clear
*g1_02*
U25 r1 port=7 clear
*g1_02_ff_ctr_sr_list*
*U10*
*U11*
*U12*
*U13*
*U14*
*U15*
*U16*
*U18*
*U2*
*U22*
*U24*
*U25*
*U4*
*U5*
*U6*
*U7*
*U8*
*U9*
*done_report_list*
*g1_02_END*
```

Explanation:

Node "U22" has a violation of guideline g1_02 at port 7. The **inport_id** (or name) of this port is "clear." The violation is "r1" which states that the "clear" inport of node "U22" is not primary input controllable. Node "U22" is highlighted.

A list of flipflops, counters, and shift registers found in the "current view." Asterics indicate that the node is NOT to be highlighted.

Processing.

The following presents the high-level algorithms, special control features, and error handling features of the report generator.

```
AskUserForFileName(Tool_n.extension)
If(FileName.extension EXISTS)
        IncrementVersionNumber(n)
If(Tool = dft)
        Open(results.tmp)
        If Open(results.tmp) = NULL
            Write(results.tmp: can't be opened)
            Exit
Open(File.extension)
GetHeaderInfo
Print(Header)
        Case Buffer
            Tool1/Option1:
                Call Print_Tool1_Option1()
            Tool1/Option2:
                Call Print_Tool1_Option2()

                .
                .
                .

        EndCase
Close(File.extension)
End.
```

Limitations.

The report generator is unaccessible via any TEA menu. At the completion of a TEA tool's execution, the process is started and a report is generated in the local directory.

An appropriate filename extension (**.rpt, .dwg, .help**) will be added to user-specified filenames, if they are not specified.

Outputs.

At the completion of report generator, a file is generated in the current directory for use by the report access routine. The filename is (by default) toolname_n.rpt, where n is the next higher integer of any other report file from the same tool. Wiring diagrams will be output by **placebit** and **compare** (see Paragraphs 2.7.4 and 2.7.5). These files will have the filename format toolname_n.dwg. **dft explain** (see Paragraph 2.7.1) will output files of the format rule_name.help. Examples of a output reports will be illustrated in the discussions of the TEA utilities and tools in Paragraphs 2.6 and 2.7.

If colors are changed on the "current view" as the result of tool output, original node and arc coloring can be restored by using the **window** utility, selecting the center of the screen and a scale factor of 0.

If the user does not want to receive one or more of the output reports, he should insert **nl:** in response to the prompt.

All output reports will have a standard header as shown in Figure 2-5. The standard header shows the date and time of creation, the user, the graph name, the data base filename, and the output report name.

Test Engineer's Assistant (TEA) Output Report

ADAS/TEA Version: PROTO.TYPE
Research Triangle Institute
P.O. Box 12194
Research Triangle Park, NC 27709

Date/Time: Thu Oct 15 09:04:35 1987

User: OTOOLE

Graph:

Current View Graph:	**Avionic_System_Hardware**
Current View Filename:	**system.hwg**

Output Report Filename = system.rpt

Tool name
(options list)

Figure 2-5. Example of Generated Report Header

Error Message/Action.

Message	Action
The results.tmp file does not exist	Make sure that dft results are available
The output file cannot be opened	Check read/write privileges
Report Generator is not available for <undefined function>	Check input

Additional Information.

2.6.2.2. Report Access (log_file).

Inputs.

The user must select **log_file** from the TEA top-level menu. From the submenu, the user selects either **view** or **print**. The user must then select the directory and file to be displayed.

Processing.

If the tool is in **view** mode, **log_file** will use the **type/page** VMS command. If the tool is in **print** mode, **log_file** will use the VMS **print** command with no options. The default printer is used. All files ending in **.rpt**, **.help**, and **.dwg** in the chosen directory will be listed.

Limitations.

Outputs.

Data is either spooled to the terminal or the default printer.

Error Message/Action.

Additional Information.

See paragraph describing

a. report generation (Paragraph 2.6.2.1)

2.6.3. On-line User Support System (help / %help). The On-Line User Support System is a readily-available reference for the user of the Test Engineer's Assistant. The material which can be referenced by the user has information about each of the tools and utilities, especially the User Interface. The on-line material references hardcopy documentation when available. Special attention is paid to each of the design for testability guidelines (e.g., statement, how they add to testability, alternative constructs, and attribute syntax).

Most menus associated with the User_Interface will have a **%help** choice available; this allows the user to indicate that he wishes further information. Depending on the menu from which **%help** is chosen, the user will be directed to the material most pertinent to the tool and to the submenu he had been investigating. In this way, the material that is most useful or timely will be provided first. TEA's top-level menu has a **help** option instead of **%help** like all other menus.

The storage of the on-line documentation is hierarchical; therefore, if the information provided initially to the user is not what is needed, the user can either step further into the hierarchy (more specific information) or back out (less specific) and take another path. **%help** is a short-cut into the on-line documentation. When **%help** is chosen as the menu option, the user is returned to the calling menu when the on-line documentation is exited. The **explain** option of the Design for Testability Guideline Checker allows a shortcut to the information concerning each of the guidelines implemented.

Inputs.

The user must chose **%help** from a menu, any of the Design for Testability Guideline Checker (**dft**) **explain** options, or **placebit <technique> no** to access the library.

Process Control.

Three possible process control scenarios can occur:

a. when a user selects **%help,** he has unlimited access to the on-line user documentation until he exits, then the process control is returned to the calling menu

b. when a user selects **dft explain,** he has unlimited access to the on-line user documentation until he exits, then process control is returned to the TEA top-level menu

c. when a user selects **placebit <technique> no**, he will get general information about the chosen technique dumped to a file called *placebit_n.help* by default

Processing.

Refer to Chapter 8 of the *VAX/VMS Utility Routines Reference Manual*, April 1986.

Limitations.

ADAS data base graphs must already have been created before TEA is entered.

Outputs.

No files are created by the On-line User Support System. If the user indicates that he wants to save the output of an **dft explain** session, the User_Interface invokes the Report Generator. The Report Generator is automatically invoked if **placebit <technique> no** is selected.

Error Message/Action.

Additional Information.

See paragraph describing

a. **dft explain** output report example (Paragraph 2.7.1)

2.6.4. Save (save).

The **save** function provides a method for the user to write changes made to the "current view" data base files. The TEA **save** function differs from the ADAS **save** function in that the user is allowed to save a copy of the current graph and associated subgraphs under a new filename. TEA will also allow overwriting of the existing file, just as ADAS does. The new feature allows the original graph data base to be retained, if desired. This option is valuable in cases where incremental changes must be assessed for one reason or another. An example would be a cost assessment trade-off study where the total computed cost is not always the linear sum of the individual changes. By retaining previous versions of the hardware graph, multiple combinations may be analyzed.

Inputs. The **save** unit has several inputs depending on the menu options selected by the user:

a. the "current graph" filename

b. the filename for the copied graph (if required)

c. the filename of all subgraphs (if required)

d. the ".hwg" filenames in the directory to which a copy of the "current graph" is to be made (if required)

e. user commands to traverse through the menu hierarchy

The "current graph" filename represents the original graph before any changes were made in the current TEA session. When a graph structure is copied to another file, TEA's user interface checks the user's VMS directory for existing hardware graph files of the same name. If a file already exists, the user is prompted to specify a new filename. When prompted, the user may also specify a filename, located in another directory, by specifying the full VMS path.

Processing. The **save** command has two subfunctions:

a. the **new_copy** command for saving a copy of the current graph, subgraphs, and template files to an external file/tree structure without altering the original data base files

b. the **overwrite** command to save the "current view" graph in the current data base files without making a copy of the structure

Graphs which are saved using the **new_copy** command are copies of the "current view" graph with all the associated subgraphs and template files. The user is prompted for the filename where the copied graph will reside. All subgraph name attributes are automatically modified to accommodate the new filenames.

To protect the user against the **quit save** option, the user is prompted to indicate whether the "current view" graph should be overwritten with the modified "current view" graph.

Limitations.

ADAS data base graphs must already have been created before TEA is entered.

Outputs.

Either new files or new versions of old files are created in the appropriate directory.

2.7. The TEA Tools. The set of tools developed as part of TEA includes five processes which will be described here. Figure 2-6 is a diagram depicting these tools and the information obtained from each. The following paragraphs will describe the inputs, outputs, limitations, and processing needed for each tool.

The TEA system accepts an ADAS schematic of the system, which has been functionally verified using VHDL, as its input. After BIT has been added to the system, the TEA methodology supports resimulation of the system in VHDL with the added BIT features and test modes by providing VHDL descriptions of the BIT modules and by instructing in the use of the modules under normal and test modes.

The Design for Testability Guideline Checker, or **dft**, uses a data base of commonly accepted design for testability guidelines to identify hard-to-test and untestable structures in a digital board design. The design topology and functions are compared to the guidelines (rules) and if any violations are found, a message is issued to the user. On-line documentation and support facilities aid the designer in removing or replacing the predefined untestable structures with more easily testable substitutes which guide the user to a testable design. Explanation of why a particular structure is untestable is also provided upon request.

The guidelines, or rules, are roughly divided into two groups: those that aid test pattern generation (e.g., make any buried control line primary input controllable) and those that aid test pattern application and fault isolation (e.g., break up long counter chains with logic that allows direct external control). Approximately 25 predefined guidelines are implemented in the current version of the tool. Additionally, a utility is provided so that dft can be field-upgradable with user-installed guidelines.

When a testability inhibiting factor (e.g., uncontrollable feedback) is found in the design, the user's "current view" is updated to show the violation in contrasting colors, and a report file is written showing the modules forming the pattern in violation and alternatives to that structure.

Dft uses a directed graph view of the system as its input. The tool then uses graph traversal and pattern matching procedures to analyze the graph for design for testability guideline violations. The graph traversal techniques are used so that many possible occurrence variations can be matched with the general guidelines.

In addition to the analyze function described above, **dft** provides both an identify and an explain function. The identify function allows the user to select a particular schematic component (e.g., all test points), and that component is highlighted on the "current view." The explain function allows the user to see all the on-line information available about a particular guideline, including the purpose of the guideline (e.g., avoid one-shots as delay elements because of their tendency to drift and provide unreliable timing pulses) and alternative constructions (e.g., use counters or timers instead of one-shots for delay elements).

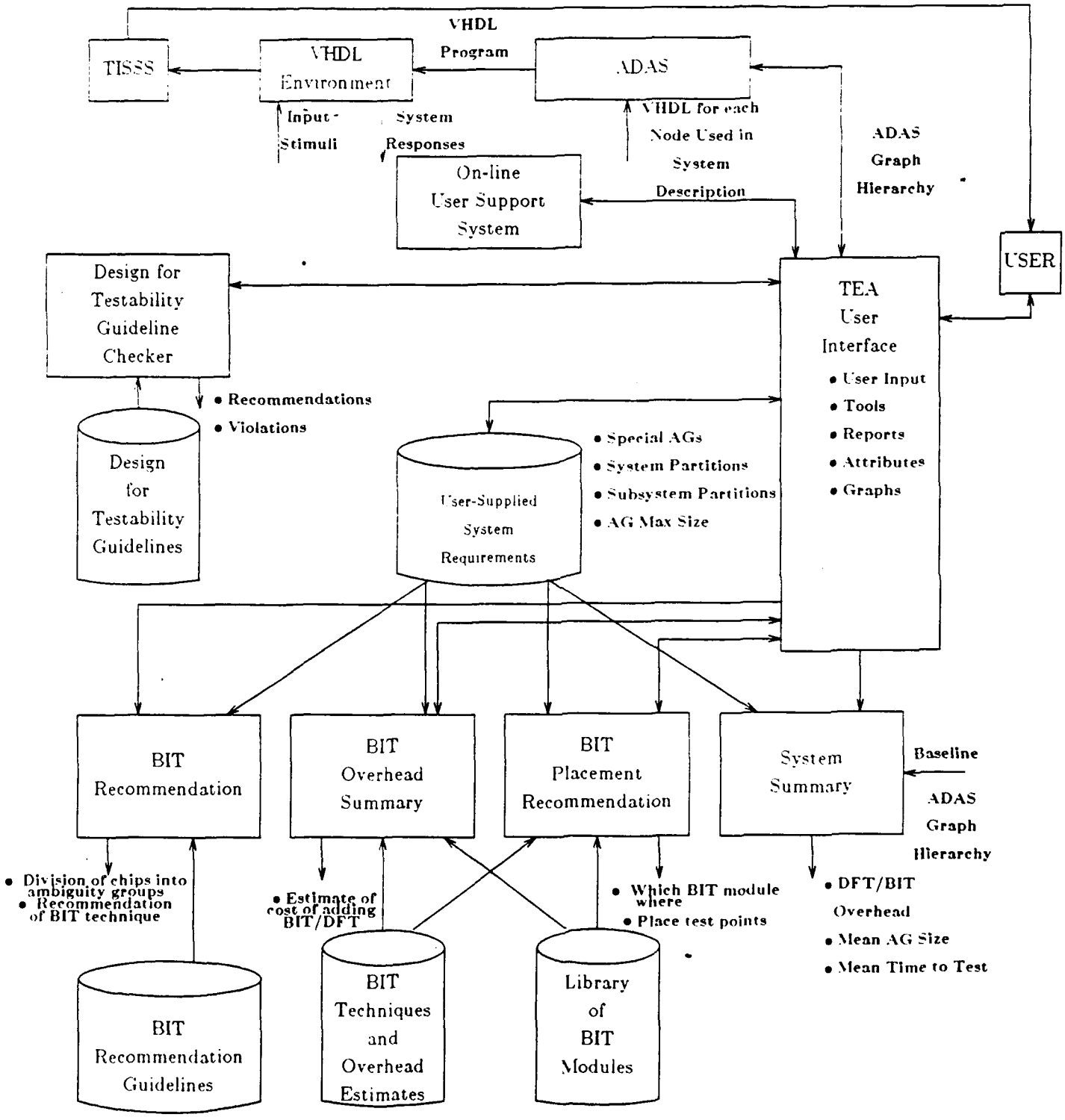


Figure 2-6. Block Diagram Description of the TEA System

Once untestable features have been removed from a design, BIT can effectively be added. Three tools provide the TEA BIT support functions. They can be invoked on each board as attributes of the system change so that the user can view the effect of the attributes on the results. For instance, the user may define special ambiguity groups, or groups of chips to be tested together, and the tool will assign the remaining groups to maintain these special groups. The tools communicate with each other by updating the internal schematic data structure. Each tool provides the user with reports of its recommendations and/or findings.

BIT Recommendation, or **brt**, uses linear programming techniques to assign chips to AGs, such that the AG maximum size requirement is met and the input/output to/from the AG is minimized so that the hardware needed to monitor the groups is also minimized. Secondary constraints on AG choices can include such testability factors as reliability, cost to replace, and difficulty in obtaining test vectors. After the AG partitioning is accomplished, the tool recommends a BIT technique for each AG; however, if sufficient information about the board is not available, the user must choose a supported BIT technique. TEA supports both on-line and off-line BIT techniques. The off-line techniques include deterministic, pseudorandom, and combinations of deterministic and pseudorandom test pattern techniques.

Once chips have been assigned to AGs, BIT Overhead Summary, or **bit_cost**, calculates the hardware overhead needed to adequately monitor the groups given the chosen BIT technique. This tool has knowledge of a data base of provided BIT modules and knows about how to assign observation and control points to the modules to obtain the fault isolation desired without over burdening the circuit.

BIT Placement Recommendation, or **placebit**, uses the chosen BIT technique and the ambiguity group information derived by BIT Recommendation to update the schematic to show the addition of the BIT support modules and required test points. The exact overhead required for this particular set of graphs is known after this tool has been run.

At any point in the TEA methodology the user can choose to "freeze" versions of the graph hierarchy to use as "baselines" or checkpoints on the design. In addition to being an aid to design documentation, System Summary, or **compare** measures any change in the graphs that possibly contributes to the testability of the design. **Dft** and **brt** give recommendations that may significantly affect these factors.

It is recommended that the user make frequent use of the TEA utility **save** so that intermediate results from the TEA tools are not inadvertently lost.

2.7.1. Design for Testability Guideline Checker (dft). One of the approaches used to reduce testing costs (e.g., length of test vector sequence, number of test points) is to recognize testing difficulties when designing the system, subsystem, board, or chip and incorporate certain features in the design. In other words, we must include testability requirements as design parameters at all levels of the design hierarchy. This procedure is known as design for testability (DFT), where testability is defined as a design characteristic which allows the unit's status (operable, inoperable, or degraded) and the fault locations within the unit to be confidently determined in a timely fashion.

The incorporation of design for testability methods in electronics first became practical with the onset of LSI logic in the 1970's. Though the military has devoted much effort to DFT, no integrated procedure for incorporating testability requirements existed in military programs until recently. This has been remedied by the issuance of MIL-STD-2165 in 1985 by the Navy Test and Monitoring Systems and the Joint Logistics Commander Panel on Automatic Testing. This standard, along with other guidelines and standards, will define and coordinate the inclusion of testability requirements in electronic systems of the future.

In dealing with the testability of a design, two key underlying concepts, *controllability* and *observability,* are often used. Controllability is defined as a relative measure of how easily a node or point can be forced to a specific logic value by setting the inputs. Likewise, observability is a measure of how easily the logical value of a node can be propagated through the unit to a primary output where it can be noted. Many of the common DFT techniques are used to enhance the controllability and observability of certain parts of a design in order to ease the testing problem.

The purpose of the DFT Guideline Checker is

a. to check features of a board level design captured in ADAS hardware graphs for untestable and hard-to-test constructs,

b. to identify connectivity and attribute information needed to assess the applicability of the DFT guidelines, and

c. to generate a report for guiding the designer toward design testability improvement by recommending different constructs and/or different modules.

The guidelines on which the Checker functions operate are divided into two groups: those that aid test pattern generation and those that aid test pattern application. If test patterns can be written in less time, the cost of creating and maintaining test program sets can be reduced. If the time to apply test patterns to a system can be reduced, then the time to detect and isolate a faulty component is shortened, thus aiding the maintenance function.

Inputs.

Before a **dft** action can be initiated, an ADAS hardware graph data base must be created using the ADAS graph editor, EDIGRAF. The data base must be initialized to contain attribute values required for the user-selected actions (see Paragraph 2.3.2). This can be accomplished by using node templates with the attributes preset or by explicitly setting the values.

The internal representation of the ADAS hardware graph data base serves three purposes. First, a number of attributes pertaining to a procedure are retrieved from this internal data base for evaluation based on user-selected options. Second, the data base is used to allow the tool to graphically indicate its outputs. Third, the data base is used as a reference for indicating its outputs for the generation of the output report.

Processing.

Dft analyze and **dft identify loops** construct two local files. *graph.pl* is the Prolog version of the ADAS data base. *control.pl* is the Prolog command file. Since **dft** is a function which operates on a hierarchical ADAS hardware graph data base, a local

data base is constructed and modified by actions taken while in a TEA session. The format of this local data base is identical to that described in Appendix D.

If *graph.pl* has been compiled since TEA was executed and the data base has not been changed, *graph.pl* will not be recompiled.

The following paragraph shows a very high-level description of the algorithm used for **dft**.

```
Get user_input(action)
If (explain)
        Get user_input(guideline)
        Output report
        Exit.
If (analyze)
        Get user_input(TSUBSYSTEMs)
        Get user_input(TBOARDs)
        Get user_input(guidelines)
        For each guideline, TSUBSYSTEM, TBOARD
                Evaluate;
If (identify)
        Get user_input(TSUBSYSTEM)
        Get user_input(TBOARD)
        Get user_input(structure)
        For structure, TSUBSYSTEM, TBOARD
                Identify;
Output report
Exit.
```

Limitations.

ADAS data base graphs must already have been created before TEA is entered.

For proper execution of the **dft** procedures, the required attributes must have a legal syntax. For each attribute there is a library of valid syntaxes or spellings (see Paragraph 2.3.2). The user can view the valid syntax for each attribute using the **explain** tool. An invalid syntax for an attribute can cause the tool not to execute or give invalid results, depending on the attribute.

TEA supports the concept of ADAS buses, but all buses are assumed to be a single bit in width.

Because *node_name* is not a normal node attribute, **dft identify attribute node_name** gives an error message: "Can't find attribute 'node_name'." rather than do what is expected.

Only one **dft analyze** can be running in any one directory at a time since the graph and control Prolog files are always called *graph.pl* and *control.pl* and the results are always placed in *results.tmp*.

Outputs.

Dft has two outputs, depending on the actions taken by the user

a. the current hierarchical ADAS hardware graph modified to graphically indicate the results of an action

1. nodes not involved will be colored dark gray
2. nodes involved, but not in violation will be colored light gray
3. violations will be shown in non-gray tones (cycling through the following color list: red, green, blue, cyan, yellow, magenta, dk_red, dk_green, dk_blue, violet, dk_yellow, orange)

b. a report file which contains the results of an action

Following are four example output files. The first demonstrates the output from **dft analyze** on multiple boards with multiple guidelines chosen. The second shows the output of **dft identify** when the user was looking for the structure "fanout > 5". The third, very similar to the second, shows the output of **dft identify** when the user was looking for the structure "test point". The final example shows the output of **dft explain**; the user has chosen the guideline g2_03's description. The header information has been removed.

Insert STANDARD HEADER here

DESIGN FOR TESTABILITY GUIDELINE CHECKER
(dft analyze)

Where:

Subsystem-

one

Board-

g15e1
g15e2
g15extra
g15n8
g15n9

Guidelines Selected-

g1_10
g1_11
g1_15

G1_10

No elements were observed to hold the properties for this guideline.

High Fanout (node:port:fanout_number):

No elements were observed to hold the properties for this guideline.

G1_11

G1_11_REC_1 - To aid test pattern generation, all unused inputs should be terminated.
During implementation, port 'inport=0' of node 'single1' should be terminated.

G1_11_REC_1 - To aid test pattern generation, all unused inputs should be terminated.
During implementation, port 'inport=0' of node 'extra1' should be terminated.

G1_11_REC_1 - To aid test pattern generation, all unused inputs should be terminated.
During implementation, port 'inport=1' of node 'extra1' should be terminated.

Tristate Devices:

No elements were observed to hold the properties for this guideline.

G1_15

```
G1_15_REC - To aid test pattern generation,
uncontrollable feedback should be avoided. Listed
below are nodes contained in uncontrollable feedback.

                combnode1a
                combnode1b
                combnode1c

G1_15_REC - To aid test pattern generation,
uncontrollable feedback should be avoided. Listed
below are nodes contained in uncontrollable feedback.

                node2a
                node2b
                node2c

There were 5 violations found.

**************************************************
Note to user:
If untestable structures exist in
your design, you should remove
or replace them before continuing
with the addition of built-in test.
Adding BIT to untestable circuits
does not necessarily increase testability.

Reminder:
Design for testability implemented at the board level
needs to be supported at the subsystem and system
levels. Add appropriate test communications
and control.
**************************************************

dft analyze completed
```

The user is directed to Paragraph 3.2 where each of the design for testability guidelines are discussed. g1_10 finds high fanout nodes; none were encountered on the boards analyzed. g1_11 finds unused inputs and unterminated tristate outputs; multiple unused inputs are noted, however no unterminated tristate outputs were found. g1_15 finds uncontrollable feedback loop patterns; two were noted and the nodes forming those loops are listed. Only the largest loops are reported to the user. Loops within larger loops are not found.

```
***Insert STANDARD HEADER here***

DESIGN FOR TESTABILITY GUIDELINE CHECKER
(dft identify)

What: Fanout > 5

Where:
        Subsystem-
                one

        Board-
                b1
                bX

Nodes having fanout greater than 5
        Tag_name2
        PO7
        Tag_name3
        n4a
        node5

dft identify completed
```

dft identify has identified five nodes that have an outport connected to 5 or more other nodes. These nodes are listed.

```
***Insert STANDARD HEADER here***

DESIGN FOR TESTABILITY GUIDELINE CHECKER
(dft identify)

What: test_point

Where:
        Subsystem-
                one

        Board-
                b3
                b8
                bX

The following arcs were found
        tpo8a/0_in
        tpo8b/0_in
        tpo8c/0_in
        tpo8d/0_in
        tpo8e/0_in

dft identify completed
```

dft identify has identified five arcs that have been assigned as test points. These nodes and their associated test point inport are listed.

```
***Insert STANDARD HEADER here***

DESIGN FOR TESTABILITY GUIDELINE CHECKER
(dft explain)

TOOLS
 DFT_Checker
  Rules
   G2_03

     G2_03 Rule: Break up long counter chains greater than 8 bits
     with logic that contains primary inputs or primary input
     controllable lines.

     Why?
        Because this results in reduction of the number of test
     patterns required to fully test the counter.

     NOTE: Follow this guideline when you use more than one
     counter chip to form long counters. Do not replace a single
     chip (long) counter with several short counter chips.

     How?
        Add shorter bit counters to the circuit and provide
     appropriate control.

     Syntax

          inport_id: clock, clk
          outport_id: co, carry_out
          Tarc_type: clock, clk, data, cntrl, ctrl, control
          Tdft_io: primary_input, pi, primary_output, po,
               test_pt_output, test_point, tp, tpo
          Tlogic_type: combinational, comb_logic, comb
          Tnode_type: counter, ctr

dft explain completed
```

dft explain has dumped the information stored in the On-line User Support System about guideline g2_03 (Break up long counter chains.) to an output report. Sections of this information include a statement of the rule, an explanation of why this rule has been included in the data base, an alternative structure to the one found in the graph, and the valid syntax of the attributes involved in the guideline.

Error Message/Action.

Additional Information.

See paragraphs describing

a. data base attributes and valid syntaxes (Paragraph 2.3.2)

b. results.tmp file (Paragraph 2.6.2.1)

c. output report header (Figure 2-5)

d. **dft** guidelines (Paragraph 3.2)

e. how to add user-defined guidelines to the data base (Paragraph 3.2.4)

2.7.2. BIT Recommendation Tool (brt). The BIT Recommendation Tool (**brt**) process advises the user on the "best" general choice of board level BIT technique given a description of a digital hardware board.

A testing methodology and grouping recommendation are assigned to each valid node on a particular board and subsystem, the nodes are divided into ambiguity groups (AGs) to meet a maximum size (number of nodes) requirement, and then a testing methodology recommendation is assigned to each AG. Valid nodes are those nodes not assigned to a special AG (see Paragraph 2.6.3) by the user or have other predetermined unique testing requirements (see section below entitled "REMOVE NON-VALID NODES").

The recommendation for each node takes on a value which refers to the "best" method of generating input, or test, vectors for the node and will take on the value of deterministic, pseudorandom, don't know (e.g., special case), or don't care (i.e., can take on either deterministic or pseudorandom). The node attribute **Ttest** holds this value. Additionally, the node attribute **Tgroup** takes on a value of either isolate (i.e., test alone or in a specified group) or by default, normal connect (i.e., group in a normal AG). This attribute can also take on the "eliminate" value, which means to not consider the node any further (non-valid nodes have this value).

No specific BIT technique will be recommended, rather a class of techniques will be recommended. Only off-line, or nonconcurrent testing techniques will be recommended by **brt**; however, TEA offers two concurrent techniques. No attempt will be made here to summarize any information specific to the techniques (Paragraph 3.4.2) or to the modules (Paragraph 3.4.3) supplied with TEA to support the techniques.

Inputs.

The user specifies the board, subsystem, and maximum ambiguity group size. For accurate results, the attributes listed in Paragraph 2.3.2 must be completed.

Processing.

The following paragraph presents the high-level algorithms, special control features, and error handling features of **brt**. More description is given for **brt** than the other tools, because **brt** uses many levels of rules for deciding ambiguity group partitioning which may cause confusion to the user.

SIX MAJOR MODULES:

remove non-valid nodes from consideration
connectivity map procedures
ambiguity group generator
branch and bound
simplex method
assign test method to ambiguity groups

REMOVE NON-VALID NODES

The objective of this procedure is to assign a testing recommendation (e.g., pseudorandom, or deterministic testing) to each node. This is accomplished by appropriately completing the **Ttest** node attribute of the ADAS data base. The valid assignments are shown in the following list. Besides the testing recommendation, the tool will recommend to eliminate or isolate to ambiguity groups nodes with special attributes (e.g., BIT modules, maintenance nodes, scannable chips, etc.), using the **Tgroup** node attribute of the ADAS data base. The tool assigns a testability recommendation and a grouping recommendation to each node.

In terms of testability (attribute **Ttest**):

DT: deterministic testing

PS: pseudorandom testing (for data lines)

DC: don't care situations (circuit can be tested efficiently with either PS or DT patterns)

DK: don't know situations (not enough information)

In terms of grouping a node to an ambiguity group (attribute **Tgroup**):

IS_ag: isolate the node in the "ag" ambiguity group

NC: the node will be grouped in an AG (default)

For both (**Ttest** and **Tgroup**):

EL: eliminate the node (no further testing considerations) The node will not be assigned to an ambiguity group.

This routine starts with a list of all nodes on a particular board and subsystem and then finds all the special testing situations itemized in the following rules and eliminates nodes from the list, leaving only "valid" nodes.

The following tests (a-u) will be examined in order, until there is a recommendation related to the testability and a recommendation related to the AG partitioning for each node. Once one of the recommendations is found, it will not be changed and the checking will continue until a recommendation is found for the other attribute. New suggestions for the recommendation attributes will not be considered.

a. Clear **Ttest**, **Tgroup**, and **Tag_test**. This implies that information from previous runs will be lost.

b. If the node is self-testable (see **Tselftest**), mark the node as **Ttest = EL**, **Tgroup = EL**.

c. If node is analog (**Tdevice_type = analog**) or a mixture of analog and digital (**Tdevice_type = mix**) mark the node as **Ttest = EL, Tgroup = EL.**

d. If the node belongs to a "special" ambiguity group (identified by the user in **Tsp_ag_name**), then mark the node as **Tgroup = IS_spec** (where **spec = Tsp_ag_name**).

e. Mark maintenance nodes as **Ttest = EL, Tgroup = EL.** Maintenance nodes can be identified by **Tbit = mn, m_n, maintenance.**

f. If a node is a JTAG/ETM support module (see **Tscan**), mark it as **Tgroup = EL** and issue message about connecting it appropriately. **Tscan** must be marked "JTAG_SUP", "VHSIC_SUP", or "ETM_SUP".

g. If a node is a BIT support module except scan-set (i.e., **Tbit = BIT_PS** or **BIT_DT)** mark it as **Tgroup = EL** and issue message about connecting it appropriately.

h. Check if the node is appropriately connected with (predecessor and/or successor) any standard test bus, shown in Paragraph 3.3. If this is true, mark the node as **Tgroup = EL.** If illegal connections are found, then the tool searches for valid subsets.

A standard bus structure contains a JTAG/ETM support module, primary inputs/outputs, or a maintenance node at the head and tail of the structure.

This case includes the following configurations:

1. ETM and JTAG (rings and stars).
2. JTAG/ETM with *sin* → *sout* pins connected in chains. **Tscan** must either be set to "JTAG", "VHSIC", or "ETM" to be considered.
3. Single nodes can form a star or ring, but they become indistinguishable from each other.

i. If the node is connected with any BIT support modules (**Tbit** of the support node is "BIT_PS", or "BIT_DT", but not **Tnode_type = scan** mark the node as **Tgroup = EL.** If the BIT support module supports a pseudorandom technique (i.e., BILBO or LFSR, **Tbit = BIT_PS**) then recommend **Ttest = PS**, else **Ttest=DT.** If a node is connected to multiple BIT support modules and the **Tbit** values are inconsistent, a message is issued and the node's **Tgroup** and **Ttest** are not set. Nodes considered are direct predecessors and successors.

j. If the node has SCAN capabilities (see if **Tscan = etm, jtag, scan, scannable, vhsic, y, yes**), and it is connected with other SCAN chips through a valid chain, mark the node as **Tgroup = IS_scan**, where "scan" is unique. **Tconfig** will also = **IS_scan.** If the node is not in a valid chain, then suggest that the user connects all the SCAN modules to a chain(s). Any scan support modules (**Tbit=BIT** or **"yes"**) connected in the chain will be isolated in the same ambiguity group with the scannable nodes. A valid chain is one which begins in and ends in a maintenance node **Tbit = mn, m_n, maintenance** or a primary input and output. Each node must have an inport with **inport_id = datain, scanin, sdi, sin** and an outport with **outport_id =**

dataout, scanout, sdo, sout. These chains can contain a combination of **Tscan = etm, jtag, vhsic** nodes.

k. If the node has BIT implemented on it (**Tbit** is PS or DT) then mark the node as **Tgroup = EL.** If the BIT scheme is a pseudorandom one, mark the node as **Ttest = PS**, else mark it as **Ttest = DT.**

l. If the node has (user-provided) test patterns (**Tfault_cov = 100**) and the inputs and the outputs of the node are directly accessible from the primary inputs and the primary outputs of the board, then mark the node as **Tgroup = EL, Ttest = DT.** If the inputs or the outputs are not directly accessible, and if this node is large (size $> 2^{16} = 65536$) then mark it as **Tgroup = IS, Ttest = DT.** Directly accessible means that all inports and outports have to be primary inputs or primary outputs by setting (see **Tdft_io**) the arcs or by setting the board names.

$$\text{Size} = (\textbf{gates})\ 2^{\textbf{states}}$$

where gates is stored in attribute **Thw_module** and states is stored in attribute **Tstates.**

m. If it is a large node (i.e., **Tnode_type** = processor or VHSIC, or size indicates large) with no test patterns available (**Tfault_cov** is not 100), mark the node as **Ttest = DK.** In this case the tool will suggest PS techniques for the data lines, and DT techniques for the control lines. Also the node is marked as **Tgroup = IS_ag** (make inputs, outputs of the directly accessible at the board pins). A warning will be issued.

n. If the node is asynchronous (see **Tasynch**) mark it as **Tgroup = EL, Ttest = DK** (Scannable asynchronous nodes have already been eliminated.).

o. If the node is a single gate or a collection of single gates (i.e., glue logic), mark it as **Ttest = DC, Tgroup= NC.** Use **Tnode_type** or **Tnode_spec_type = glue** or **gate** or **Thw_module <= 25** to identify these nodes. **Tnode_type** or **Tnode_spec_type = gate or glue** will trigger this rule. These nodes will eventually be combined with neighboring nodes.

p. Mark memory nodes (Use **Tnode_type**) as **Ttest = DC, Tgroup = IS_mem.** This indicates that memories will be tested separately as logical groups with either PS or DT test patterns.

q. If the node is combinational (see **Tlogic_type**) with less than 17 input data lines, then mark the node as **Ttest = PS, Tgroup = NC** (exhaustive testing can be accomplished in this case).

r. If none of the above applies to the node under test, mark the node as **Tgroup = NC** and then use the BRT library to identify a testing methodology. Use **Tnode_type** and **Tnode_spec_type** to index into the library. See a later section in this Paragraph for information about setting up this library. Both the **Tnode_type** and the **Tnode_spec_type** of the node must match a library entry to have it correctly identified.

s. If the user has simulation results that indicate that the expected fault coverage (according to the testing specification) is satisfied with the application of less than 2^{16} test patterns, then mark the node as **Ttest = PS**. If not, mark the node as **Ttest = DT**. If **Tfault_cov** and/or **Ttest_len** is blank or equal to 0, this rule is skipped.

The relationship between expected fault coverage and pseudorandom test length can be approximated with the following exponential function:

$$E_c = (1 - e^{-aL})$$

Evaluate the constant *a* from the simulation results given by the user (E_c = **Tfault_cov** / 100 and L = **Ttest_len**). After evaluating the constant *a* we evaluate the expected fault coverage for 2^{16}=65536 test patterns. If the E_c is high (> 90%), then mark node as **Ttest = PS**, else mark node as **Ttest = DT**.

t. If the sum of memory elements (see **Tstates**) and the input lines in a sequential node is less than or equal to 16, then mark the node as **Ttest = PS** (pseudoexhaustive testing). If a node has no other attributes other than **Tboard** and **Tsubsystem** set, this rule applies. The node will be considered not large and sequential.

u. If none of the above applies, then mark the node **Ttest = DT**.

The input to the remaining modules is the set of valid nodes that must be assigned to ambiguity groups and the maximum ambiguity group size.

CONNECTIVITY MAP PROCEDURES

The connectivity map procedures take an ADAS hardware graph and generate a connectivity map in memory that summarizes the connections between the valid nodes (as a result of REMOVE NON-VALID NODES). This connectivity map describes which nodes are connected to each node by DATA arcs and by how many input and output arcs. This information is then used to generate the ambiguity groups and calculate their associated cost factors.

For every node in the valid node set
Enter it in the connectivity map
Find all the DATA successors of this node
Count the number of input arcs to each successor from this node
Enter them in the connectivity map
Find all the predecessors of this node
Count the number of output arcs from each predecessor to this node
Enter them in the connectivity map

For every node in the connectivity map
Assign it a graph number
Recursively assign this graph number to every node
to which this one is connected

Construct a node set for each disconnected graph

AMBIGUITY GROUP GENERATOR

The ambiguity group generator is not run if AG_SIZE = 0 or 1 or if there are no valid nodes left to group. A value of zero for AG_SIZE makes the largest groups possible on the board. Non-positive and non-integer values for AG_SIZE are prohibited.

This module generates a list of all groups, without known testability problems of the size given and less. This list contains each group's name, cost, list of nodes in the group, size, graph number, and AG mask. (The AG mask has a bit set for each node that is in the group. The bit is determined from where the node occurs in the valid node set.)

Generate the connectivity map
From the connectivity map, generate a list of all groups of size 1
From the list of groups of size n, generate a list of groups of size n+1
For each group
For each node in the group
Find the node in the connectivity map,
and which nodes are connected to it
If a connected node is not already in the group,
add it to the group
check testability
If ok, insert this group in the new list
Calculate this group's cost and ag_mask
give it a name
Link the size n+1 list onto the beginning of the size n list

The guidelines used to form groups are stated here

a. Nodes in an ambiguity group are connected via data lines. The **Tarc_type** attribute is used to define a line as data, control, or clock. Nodes only

connected by control or clock lines do not represent possible connected nodes. The direction of flow of the line is not considered.

b. Maintain special ambiguity groups (i.e., nodes marked as **Tgroup** = IS_ag_name should be in the **Tsp_ag_name** = "ag_name" ambiguity group).

c. Number of nodes in an ambiguity group is limited to the user's maximum.

d. Optimize choice of ambiguity groups so that "cost of overhead" is minimized. This "cost" is an integer assigned to represent the relative ease of testing that particular group. Some groups will be eliminated due to "poor testability". These include

   1. multiple DK (**Ttest**) nodes in the same group
   2. mixture of DK and DT nodes in the same group
   3. mixture of DK and PS hard-to-test nodes in the same group
   4. mixture of DT and PS hard-to-test nodes in the same group
   5. multiple PS hard-to-test nodes in the same group

Hard-to-test nodes are nodes whose

   1. **Ttest_len** > 60,000 or
   2. Size > 2^{16}

The possible ambiguity groups are computed individually by the user-supplied AG_SIZE. Selecting an AG_SIZE of '3' will produce possible ambiguity groups of size 3, 2, and 1. Each possible ambiguity group has a particular "cost" associated with it. This "cost" represents the relative value of the group in term of testability. Groups having the higher cost values are groups which are less desirable than those groups with smaller cost. The ambiguity group algorithm finds the configuration of possible ambiguity groups which represents the least possible cost. The solution must contain all nodes on the board.

The initial cost of a possible ambiguity group is calculated by determining the sum of output arcs which leave the group and terminate in some other group on the current board of interest. The **Tarc_type** attribute is not used when computing the number of output arcs leaving a group. From this initial cost, the group cost may be modified if certain known testability features are present.

A group's cost is modified only when it can be determined that the group is unusually hard or easy to test. Several generalizations have been made concerning testability features and their relationship with a possible ambiguity group.

a. A group which contains feedback eliminates the occurrence of at least one feedback between ambiguity groups. Feedback between ambiguity groups significantly increases the effort needed to test the groups. All possible ambiguity groups which contain feedback (data lines only) have their initial cost DECREASED by 90%. This includes self-loops or nodes which use their own data output.

b. Testing becomes easier when a group contains all pseudorandomly testable nodes and the number of inputs to the group is less than or equal to the number of pins on standard test module. All groups which contains only

pseudorandomly testable nodes and have < = 16 inputs will have their initial cost DECREASED by 50%.

c. Groups which contain only glue logic are undesirable. The overhead associated with testing this type of logic as a group is too significant. All possible groups which contain only glue logic will have their initial cost INCREASED by 50%.

If none of these three testability features are present in the group, then the cost is simply the sum of the outputs from the group which terminate on the board. If more than one is present, then the increase or decrease is taken from the modified cost not on the original output arc sum.

The final cost is an integer, so after the multiplication of the penalty/reward factor, the result is truncated.

If any final costs are > 256, the costs of all the groups are normalized to be within the range

$$-1 < x < 257.$$

BRANCH AND BOUND

This module finds the optimum configuration of the ambiguity groups. An optimum configuration is one where each node is assigned to an ambiguity group, and the total cost of the ambiguity groups is minimized. This module uses a branch and bound approach, with the lower bounds calculated using the simplex method. A node list is generated that lists which ambiguity groups contain each node. This list is ordered by the nodes that are in the smallest number of AGs first. There are no duplicate ambiguity groups in this list. Hashing is used to speed this process.

Branch and bound must be run on each disconnected graph representing the board.

Generate the node list
For each group that the first node is in, calculate a lower bound on the cost if this group is used in the configuration
Branch using the group with the lowest bound
For the next node in the list that hasn't been covered by an ambiguity group, calculate lower bounds
Branch using the group with the lowest bound
This continues until a solution configuration is found, or it is determined that this configuration will not yield a valid solution
Backtrack, to try and find a better solution configuration

Note: The branch routine does not choose a branch where the lowest bound is greater that a solution obtained so far. This "branch and bound tree pruning" saves a significant amount of time, as compared with a brute force "branch" tree. Valid solutions have each valid node assigned to one and only one ambiguity group which is connected.

SIMPLEX METHOD

This module solves a linear program. It uses the revised simplex method with the explicit form of the inverse and the lexico minimum ratio rule to avoid cycling. (Cycling is very prominent in set partitioning problems.)

ASSIGN TEST METHOD TO AGs

After the division of nodes into ambiguity groups, **brt** assigns a testing method (PS, DT, or DK) to each AG. Only nodes with **Tag_name** or **Tsp_ag_name** set to some value are considered.

The following tests are performed in order

a. If an ambiguity group has any node(s) that is(are) deterministically testable, then the AG is DT testable, **Tag_test = DT.**

b. If the AG consists of one or more nodes marked as DK, then the AG is marked as **Tag_test = DK.** A recommendation to test this AG's data lines pseudorandomly and control lines deterministically will be issued.

c. If the ambiguity group has an embedded large (size $> 2^{16}$) PS node, most >= 90% of the inputs or the outputs of this node are not accessible from the inputs or the outputs of the AG) which is close to the pseudorandom limit (i.e., **Ttest_len**, is greater than 60,000 test patterns), then mark AG as **Tag_test = DT.**

d. An AG is marked as **Tag_test = DT** if any large (size $> 2^{16}$) nodes are marked as **Ttest = PS**, and they have test lengths greater than the pseudorandom limit (e.g., 64,000). A warning should be given to the user in this case since the last statement is not absolutely true (fault simulation is required for this case). All other groups with PS nodes will be marked **Tag_test = PS.**

e. If all nodes in an AG are **Ttest = DC**, then assign test method of adjoining AG to this AG. The first AG attached to the output of this AG which has **Tag_test** set, will be used unless there is no succeeding AG. If there is no succeeding AG, then the **Tag_test** of the preceding AG is used. If there are no other AGs, **Tag_test** is set to PS.

Size of a Node

The size of the chip in terms of sequential complexity is defined as:

$$\text{Size} = (\textbf{gates})\ 2^{\textbf{states}}$$

where gates is stored in attribute **Thw_module** and states is stored in attribute **Tstates**. Size is needed to determine when a node is "large", or Size >= 2^{16}. If either **Thw_module** or **Tstates** has a null value, a node is not considered "large."

BRT Library

The purpose of the BRT library is to provide the user with test attribute information for different structures. The library should have the following entries

a. The character "%" starts each record. It should appear as the first and last character of the library file on lines by itself. A line with a lone "%" should separate each record.

b. *Node Identifier:* This entry will contain the name of the circuit, as well as any aliases that represent similar structures (**Tnode_type**).

c. *Identifying Number:* Specific device types (**Tnode_spec_type**).

d. *Testing recommendation:* Possible entries are DT, PS, or DC (**Ttest**).

e. *Test Length:* Number of test vectors that give the fault coverage specified in the next entry (**Ttest_len**).

f. *Fault Coverage:* This entry is used in conjunction with the test length to determine whether a node is pseudorandomly testable or not (**Tfault_cov**).

g. *References - Comments:* This field gives information to the user concerning the entries specified in other fields, and the references from which these results where obtained.

h. The character "%" starts each record. It should appear as the first and last character of the library file on lines by itself. A line with a lone "%" should separate each record.

Tnode_type and **Tnode_spec_type** of a node must each match one of the entries on lines 1 and 2 of the record. **Tnode_spec_type** matches line 2 if both are null strings. If the library record matches a node, then **Ttest** is set to the value found on line 3, **Tfault_cov** is set to the value found on line 4, and **Ttest_len** is set to the value found on line 5.

Example BRT Library

name	spec_name	test	test_len	fault_cov	comments
PLA		DT			special PLAs may be PS
adder		PS	100	100	independent of input lines
multip	25s05	PS	200	100	4x2 cascadable modules 12x6 (9 modules) 200 pat: 93%
alu	xx181	PS	180	100	4-bit alu (xx: 74,54,40)
counter	xx163	PS	50	100	control lines DT
compar	xx85	PS	20	100	independent of input lines
encoder		PS	100	100	independent of input lines
decoder		PS	100	100	independent of input lines
mux		PS	100	100	independent of input lines
regist	xx174	PS	50	100	independent of input lines
shft-re		PS	200	100	independent of input lines
multi	1010	PS	100	93	16x16 Booth's algorithm
parity	xx180	PS	60	100	independent of input lines

Example BRT library file (tea_brt contains):

The record delimiters (%) are necessary. To find an entry in the BRT library, **Tnode_type** and **Tnode_spec_type** must match.

```
%
PLA

DT


special PLAs may be PS
%
adder

PS
100
100
independent of input lines
%
multip
25s05
PS
200
100
4x2 cascadable modules
%
```

Another example showing multiple values possible for **Tnode_type** and **Tnode_spec_type.**

```
%
two 2 second
```

2 to too
PS
88
50
just an example
%

Limitations.

ADAS data base graphs must already have been created before TEA is entered.

Biports are ignored.

TEA supports the concept of ADAS buses, but all buses are assumed to be a single bit in width.

Outputs.

The **Tag_name**, **Ttest**, **Tgroup**, and **Tconfig** attributes will be reset at the start of execution and will be set by the process. **Tag_name** holds the value of the ambiguity group to which the process assigns the node. **Tconfig** holds the name of any special configuration to which the node belongs, as identified by the process.

At the completion of **brt**, a report is generated to itemize the ambiguity groups and the testing methodology assigned to them.

Using the rules that find special testing situations, messages are issued in the following cases

a. If any of

1. BIT modules
2. nodes with BIT
3. selftestable nodes
4. scannable VHSIC nodes

exist on a board, output: "Make sure all BIT modules, nodes with BIT, selftestable nodes, and scannable VHSIC nodes have accessible control lines and observable output lines."

b. If no selftestable nodes exist, output: "No selftestable nodes were found on board "xyz." Use attribute **Tselftest** to indicate selftestable nodes."

c. If no chips with BIT exist, output: "No nodes with BIT were found on board "xyz." Use attribute **Tbit** to indicate nodes with BIT."

d. If no scannable nodes exist, output: "No scannable nodes were found on board "xyz." Scannable nodes should be used when possible because they increase testability. Use attribute **Tscan** to indicate scannable nodes."

Following is an example of the output from **brt** with AG_SIZE set to 4. The report has many sections, including the itemization of special testing situations. After all special cases have been itemized, a list of the remaining nodes that will be assigned to ambiguity groups is shown. Here, there are 17 nodes to be assigned. Following this list is the output and statistics from the "AMBIGUITY GROUP GENERATOR" code. In this example there are three disconnected graphs, and a solution has been found for

each one (Since ambiguity groups must be connected by **arc_type** = DATA arcs. disconnected graphs are not uncommon.) graph. Graph 1 has chosen nodes T21b. T21a, T19b, and T19a to be in ambiguity group "AG47", or **Tag_name** = AG47 and nodes T15d, T15c, T15b, and T15a to be in ambiguity group "AG43". or **Tag_name** = AG43. Graph 2 only had one node and this was assigned to ambiguity group "AG9". Graph 3's eight nodes were assigned to ambiguity groups "AG45" and "AG46". The cost of graph 1's division is one, the cost of graph 2's division is four, and the cost of graph 3's division is one. This will result in a total cost of six test points or observable arcs.

Insert STANDARD HEADER here

BIT RECOMMENDATION
(brt)

BRT will divide the nodes on subsystem 'phase1',
board 'p1three' into ambiguity groups of size '4' or less.

Modified cost function will be minimized.

Found a valid scan chain
T8mixa
T8mixb
T8mixc

Found a valid ETM star configuration
T8eSa
T8eSb
T8eSc
T8eSd

Found a valid ETM ring configuration
T8eRa
T8eRb
T8eRc
T8eRd

Found a valid JTAG star configuration
T8jSa
T8jSb
T8jSc

Found a valid JTAG ring configuration
T8jRa
T8jRb
T8jRc

Found a valid scan chain
T10a
T10b
T10c

Node 'T6b' is a scan support and should be properly connected
Node 'T6a' is a scan support and should be properly connected
Node 'T8jsup' is a scan support and should be properly connected
Node 'T8etmsup' is a scan support and should be properly connected

Make sure all BIT modules, nodes with
BIT, selftestable nodes and scannable
VHSIC nodes have accessible control
lines and observable output lines.

No selftestable nodes were found on this board.
Use attribute Tselftest to indicate

selftestable nodes.

Removing special AG nodes
- T4c
- T4b
- T4a

Removing maintenance nodes
- T4a
- T8mnc
- T8mn
- T10mn
- T8mnb

Removing JTAG/ETM support nodes
- T6b
- T6a
- T8jsup
- T8etmsup

Removing JTAG/ETM rings, stars and scan chain nodes
- T8mixa
- T8jSa
- T8jRa
- T8eSa
- T8eRa
- T8mixb
- T8jSb
- T8jRb
- T8eSb
- T8eRb
- T8mixc
- T8jSc
- T8jRc
- T8eSc
- T8eRc
- T8eSd
- T8eRd

Removing SCAN nodes in valid scan chains
- T10a
- T8mixa
- T10b
- T8mixb
- T10c
- T8mixc

Removing asynchronous nodes
- T4b
- T8mixa
- T8jSa
- T8jRa
- T8eSa
- T8eRa
- T8mixb

```
    T8jSb
    T8jRb
    T8eSb
    T8eRb
    T8mixc
    T8jSc
    T8jRc
    T8eSc
    T8eRc
    T8eSd
    T8eRd

Removing memory nodes
    T6a

These nodes will be assigned to ambiguity groups.
    T21b
    T21a
    T19b
    T19a
    T15d
    T15c
    T15b
    T15a
    T8POb
    notT18
    T18c
    T18b
    T18a
    T20c
    notT20
    T20a
    T20b

There are 53 possible ambiguity groups of size 4 or smaller.

---------- graph #1 of 3  Thu Oct 13 10:41:18 1988

Optimal configuration has cost function value = 1:
Number of output lines = 1:
The Tag_name attribute has been set for the following nodes:
        (output_lines,cost_fcn_value) ambiguity_group: nodes_in_AG
        (0,0) AG49: T21b T21a T19b T19a
        (1,1) AG44: T15d T15c T15b T15a


---------- graph #2 of 3  Thu Oct 13 10:41:19 1988

Optimal configuration has cost function value = 14:
Number of output lines = 14:
The Tag_name attribute has been set for the following nodes:
        (output_lines,cost_fcn_value) ambiguity_group: nodes_in_AG
        (14,14) AG9: T8POb
```

---------- graph #3 of 3 Thu Oct 13 10:41:19 1988

Optimal configuration has cost function value = 1:
Number of output lines = 1:
The Tag_name attribute has been set for the following nodes:
(output_lines,cost_fcn_value) ambiguity_group: nodes_in_AG
(0,0) AG48: notT18 T18c T18b T18a
(1,1) AG47: T20c notT20 T20a T20b

Start time = Thu Oct 13 10:41:12 1988
Finish time = Thu Oct 13 10:41:19 1988
Total configuration cost = 16
Total # of output lines = 16

Test methods set for each AG

AG44	PS
AG47	DT
AG48	DT
AG49	DT
AG9	DT
group1	PS
group5	DK

****

Note to user: Use compare to get a listing of the nodes in each ambiguity group.
To affect the groups themselves, use ag_name to set special ambiguity groups and rerun BRT.

Supported off-line BIT techniques

1. Continuous test point monitoring (det_tp)
   supports PS and DT strategies

2. Test point monitoring with data compression
   (tp_sa) supports PS strategies on
   input data lines.

3. Board level boundary scan (boundary) supports
   DT strategies.

4. Scan-set supports both PS and DT strategies.

5. Test pattern generation with data compression
   (gen_sa) supports both PS and DT techniques.

PS = Pseudorandomly generated test pattern
DT = Deterministically generated test pattern
****

brt completed

Error Message/Action.

Message	Action
Node *name* had **Tag_name** and **Tsp_ag_name** set, **Tag_name** cleared	No action required

Additional Information.

See paragraphs describing

a. data base attributes and valid syntaxes (Paragraph 2.3.2)

b. results.tmp file (Paragraph 2.6.2.1)

c. special ambiguity groups (Paragraph 2.6.3)

d. output report header (Figure 2-5)

e. standard test buses (Paragraph 3.3)

Brt should be used iteratively. The results from the tool are enhanced by user input about specific groupings (i.e., special ambiguity groups) and AG size. By using special AGs, the user can speed up the execution of **brt** by eliminating some possible groups because nodes can only be assigned to a single AG.

2.7.3. BIT Overhead Summary (bit_cost). The BIT Overhead Summary (**bit_cost**) process advises the user on the hardware overhead costs to implement a particular design for testability / built-in test technique given a set of ADAS graphs describing a digital hardware board.

Paragraph 3.4.2 describes each of the five BIT techniques that are implemented as part of **bit_cost**. The five techniques include

a. continuous test point monitoring,

b. test point monitoring with compressed data,

c. board level boundary scan,

d. scan-set, and

e. test pattern generation with signature analysis.

Paragraph 3.4.3 describes the TEA-supplied BIT modules, available as ADAS templates and implemented in VHDL. These include

a. scan-set register,

b. programmable feedback pseudorandom test pattern generators,

c. built-in logic block observer (BILBO),

d. testing-switch (TSWITCH), and

e. an example maintenance node.

Inputs.

The user must supply the board and subsystem names and the BIT technique(s) to be considered.

Processing.

The following paragraph presents the high-level algorithms, special control features, and error handling features of **bit_cost**.

```
Get user_input(Tsubsystems)
Get user_input(Tboards)
Get user_input(technique(s))
For each technique
        If (technique != 3)
                Find ambiguity group boundaries
        Apply technique(s)
        Accumulate costs
Output report
```

Accumulate Costs

Technique	Name	Costs
1-5		Add 3 control inputs for reset, tphi1, tphi2 if any modules added. Add a Maintenance Node.
1*	continuous test point monitoring (**det_tp**)	Add test points as necessary.
2*	test point monitoring with data compression (**tp_sa**)	Add 8 control inputs and 1 test outputs per BILBO. Add 5 control inputs and 1 test outputs per Test Pattern Generator. Add 2 control inputs for any 2x16 Multiplexors. Add 2 control inputs for each 4x16 Multiplexor.
3*	board level boundary scan (**det_tp**)	Add 6 control inputs and 2 test outputs per Scan-Set module.
4*	scan-set (**scan_set**)	Add 6 control inputs and 2 test outputs per Scan-Set module.
5*	test pattern generation with data compression (**gen_sa**)	Add 16 control inputs and 2 test outputs per Testing Switch.

*Many of the control inputs and outputs can be shared among the BIT modules. The I/O overhead counts predicted by the **bit_cost** are upper bounds.

Limitations.

ADAS data base graphs must already have been created before TEA is entered.

The process only applies to the five BIT techniques listed above. Example implementations of the techniques will only use BIT modules listed in this section or otherwise provided in the supplied library. There are many example implementation possibilities, but only one has been chosen for each of the techniques. Refer to Paragraph 3.4.2 for technique descriptions.

Biports are ignored.

TEA supports the concept of ADAS buses, but all buses are assumed to be a single bit in width.

Test points are not placed on board outputs.

Graphs are "flattened" during **bit_cost** execution. This may result in BIT module cost which is low compared to what is required for a simulation, due to the requirement of maintaining graph boundaries.

Outputs.

At the completion of **bit_cost**, a report is generated to itemize the costs of implementation of the chosen technique(s). A wiring diagram file is created to show, in general, how to add modules.

This example output report from **bit_cost** itemizes the costs for selecting four of the five available techniques, each considered individually. The user is directed to the sections describing the BIT techniques and supporting modules (Paragraphs 3.4.2 and 3.4.3). The numbers associated with primary inputs and outputs are given assuming the user controls each additional module separately and externally from the board. The negligible costs associated with the "boundary" technique result from only having one board on the test graph. The tool finds that it is not necessary to add any additional overhead to the board to observe the board's output. **bit_cost** only considers nodes that have either **Tag_name** or **Tsp_ag_name** set.

Insert STANDARD HEADER here

BIT OVERHEAD SUMMARY
(bit_cost)

The following nodes will be considered

n9
n2
n1
n0
n4
n3
n10
n7
n6
n5
n8

Generating hardware overhead for subsystem 'only', board 'first'

Counts for BIT technique 'tp_sa'

2 Scan-set Modules
0 Testing Switches
0 TPGs
1 BILBOs
0 2x16 Muxes
1 4x16 Muxes
0 Test Points
25 BIT Module Control Inputs
5 BIT Module Test Outputs
1 Maintenance Node

Counts for BIT technique 'boundary'

0 Scan-set Modules
0 Testing Switches
0 TPGs
0 BILBOs
0 2x16 Muxes
0 4x16 Muxes
0 Test Points
0 BIT Module Control Inputs
0 BIT Module Test Outputs
1 Maintenance Node

Counts for BIT technique 'scan_set'

4 Scan-set Modules
0 Testing Switches
0 TPGs
0 BILBOs
0 2x16 Muxes
0 4x16 Muxes
0 Test Points
27 BIT Module Control Inputs
8 BIT Module Test Outputs

```
                1 Maintenance Node

Counts for BIT technique 'gen_sa'
                0 Scan-set Modules
                2 Testing Switches
                0 TPGs
                0 BILBOs
                0 2x16 Muxes
                0 4x16 Muxes
                0 Test Points
               35 BIT Module Control Inputs
                4 BIT Module Test Outputs
                1 Maintenance Node

**************************************************
NOTE to user:
If the costs itemized here are excessive, consider
fixing some of the ambiguity group partitions with
ag_name and/or increasing ambiguity group size and
rerunning brt.
**************************************************

bit_cost completed
```

Insert STANDARD HEADER here

BIT OVERHEAD SUMMARY
(bit_cost)

Wirelist for 'tp_sa' on subsystem 'only', board 'first'

Data for AG 'ag9'
Add on board BILBO line to n9/O1
Data for AG 'longname'
Add on board BILBO line to n1/O1
Data for AG 'agbus'
Data for AG 'norm'
Add SCAN line to n10/I0
Data for AG 'a1'
Add SCAN line to n8/I0

Wirelist for 'boundary' on subsystem 'only', board 'first'

Wirelist for 'scan_set' on subsystem 'only', board 'first'

Data for AG 'ag9'
Add output SCAN line to n9/O1
Data for AG 'longname'
Add output SCAN line to n1/O1
Data for AG 'agbus'
Data for AG 'norm'
Add control input SCAN line to n10/I0
Data for AG 'a1'
Add data input SCAN line to n8/I0

Wirelist for 'gen_sa' on subsystem 'only', board 'first'

Data for AG 'ag9'
Data for AG 'longname'
Data for AG 'agbus'
Data for AG 'norm'
Add control input TSWITCH line to n10/I0
Data for AG 'a1'
Add data input TSWITCH line to n8/I0

bit_cost completed

Error Message/Action.

Message	Action
Node *name* had **Tag_name** and **Tsp_ag_name** set, **Tag_name** cleared	No action required
TPO count may be exaggerated due to fanout	No action required

Additional Information.

See paragraphs describing

a. data base attributes and valid syntaxes (Paragraph 2.3.2)

b. results.tmp file (Paragraph 2.6.2.1)

c. output report header (Figure 2-5)

d. BIT techniques (Paragraph 3.4.2)

e. BIT modules (Paragraph 3.4.3)

2.7.4. BIT Placement Recommendation (placebit). The BIT Placement Recommendation (**placebit**) process advises the user on how to implement a particular design for testability / built-in test technique given a set of ADAS graphs describing a digital hardware board. This advice can take three forms

a. general instructions for the chosen technique without specific implementation information about modules needed, I/O lines needed, or the current view data base

b. a wiring list from which the user can manually update his ADAS graphs to implement the TEA example implementation (using TEA-supplied library BIT modules) of the chosen technique

c. ADAS graphs which are updated to show the example implementation of the chosen technique

In this latter case, library BIT modules are "plopped" onto the graph and appropriate internal data, control, and clock lines are routed as best as possible for the user. For either of the two latter options, each of the modules on the selected board must be assigned to an ambiguity group (for BIT techniques 1, 2, 4, and 5). This implies that the user manually sets the attributes **Tsp_ag_name** and **Tag_name** using **ag_name** (see Paragraph 2.6.1) and/or automatically sets the attributes using **brt** (see Paragraph 2.7.2).

Paragraph 3.4.2 describes each of the seven BIT techniques that are implemented as part of TEA. The seven techniques include

a. continuous test point monitoring

b. test point monitoring with compressed data

c. board level boundary scan

d. scan-set

e. test pattern generation with signature analysis

f. parity generation/checking

g. on-line comparison of duplicated modules

Paragraph 3.4.3 describes the TEA-supplied BIT modules, available as ADAS templates and implemented in VHDL. These include

a. scan-set register

b. programmable feedback pseudorandom test pattern generators

c. built-in logic

d. testing-switch

e. an example maintenance node

f. 9-bit parity generator/checker

g. 8-bit equal comparator

Inputs.

The user must specify the subsystem and board names and the technique to use.

Processing.

The following paragraph presents the high-level algorithms, special control features, and error handling features of **placebit.**

```
        IF (OUTPUT = general)
                SupplyInfo (TECHNIQUE)
                Goto EndOfPlacebit
        IF (TECHNIQUE = 6)              /* parity generation/checking */
                GetReadyFor (TECHNIQUE)
        ELSEIF (TECHNIQUE = 7)                  /* comparison */
                GetReadyFor (TECHNIQUE)
/* check for special graph concerns including boundary
scannable parts, VHSIC parts, and asynchronous logic */
        ELSEIF (TECHNIQUE < 6)
                Startup (TECHNIQUE)
        IF (OUTPUT = wirelist)
                DoWireList (TECHNIQUE)
        IF (OUTPUT = autoplace)
                DoAutoPlace (TECHNIQUE)
EndOfPlacebit:
        WriteOutputReport ()
End.
        DoWireList (TECHNIQUE)
```

Limitations.

ADAS data base graphs must already have been created before TEA is entered.

The process only applies to the seven BIT techniques listed in this section. Example implementations of the techniques will only use BIT modules listed in this section or otherwise provided in the library. There are many example implementation possibilities, but only one has been chosen for each of the techniques.

The routine which updates the ADAS graphs will not generally make "pretty" graphs; additional BIT modules are placed by a "first available position" procedure.

Biports are ignored.

TEA supports the concept of ADAS buses, but all buses are assumed to be a single bit in width.

To minimize confusion, it is recommended that only one BIT technique of 2, 4, and 5 be added to a particular board. To accomplish this, BIT modules are not assigned to an ambiguity group so ambiguity group boundaries will not be identified for further BIT module placement.

Because board and ambiguity group boundaries may cross graph boundaries, more than the minimum number of BIT modules may be added.

Highly connected and tightly packed graphs will result in extremely long run times. **placebit autoplace** is provided to save the user time in updating graphs, so it is only at the option of the user that **autoplace** runs. A wirelist can be obtained in much shorter time and this can be used to manually update the graph with the necessary BIT support modules.

No test points are added at board boundaries.

To update a hardware graph data base, a corresponding "**.hwt**" (hardware template) file is required.

Outputs.

At the completion of **placebit**, a report is generated to illustrate the options implemented and the termination status. If the general output option was chosen, the report will also contain a statement about how to generally implement the chosen technique. Those general statements are listed here.

Node and port names are given when possible to aid the user in identifying structures.

1. continuous test point monitoring (**det_tp**)	Make all lines which leave an ambiguity group observable test points.
2. test point monitoring with data compression (**tp_sa**)	Provide pseudorandom input vectors for all primary input data lines and deterministic input vectors for all primary input control lines. Break feedback paths between ambiguity groups, such that all lines are settable under test control. To get better access to internal ambiguity groups, also break all fanout lines between ambiguity groups. All lines which are ambiguity group outputs should be diverted to a signature analyzer.
3. board level boundary scan (**boundary**)	Place scannable registers at all the primary inputs to the board.
4. scan-set (**scan_set**)	Completely isolate ambiguity groups with scannable registers.

General Output Messages (Continued)

5. test pattern generation with data compression (**gen_sa**)	Completely isolate ambiguity groups with appropriate test pattern generators and signature analyzers.
6. parity check/generate (**parity**)	Place parity checker or generator on the desired data lines. Choose either odd or even parity.
7. on-line comparison of duplicated modules (**compare**)	Place the comparison module such that the lines to be compared are routed to it and the error flag output is accessible.

If the wirelist option is chosen, **placebit** will provide a file with name of the format *placebit_n.dwg*, that provides instructions for manual update of the ADAS graphs to accommodate the example implementation of the BIT technique chosen.

If the autoplace option is chosen, **placebit** will provide an updated current view data base, which the user may save to permanent storage. To make the graph "pretty," the user is directed to use EDIGRAF, the graph editor.

Error Message/Action.

Message	Action
Node %s had **Tag_name** and **Tsp_ag_name** set, **Tag_name** cleared	No action required
TPO count may be exaggerated due to fanout	No action required
Fatal template load error	Get a template file
Size of sets not equal	Add more port choices

Two example output files follow. The first is the "report", or .rpt output of the tool. Information found here include the options chosen in the run of the tool, the nodes considered (**Tag_name** or **Tsp_ag_name** set), an itemized list of the BIT support modules to be added, and the arcs that should be routed to the modules. The .dwg information is more specific and shows the steps needed to reroute all the graph's arcs to gain the new test capabilities.

Insert STANDARD HEADER here

BIT PLACEMENT RECOMMENDATION
(placebit tp_sa wirelist)

Test Point Monitoring with Data Compression

Add pattern generation and signature analysis on subsystem 'only', board 'first'

The following nodes will be considered for this technique
n9
n2
n1
n0
n10
n7
n6
n5
n8

Add modules to graph 'test2.hwg' for ambiguity group 'ag9'
Add 0 TPG and MUX2 nodes for internal arcs
Add 0 TPG and MUX2 nodes for external/PI arcs
Add 0 SCAN-SET nodes
Need 1 BILBO and MUX4 port sets
Intercept the following outputs and route to the BILBO nodes:
n9/O1
n9/O2

Add modules to graph 'test2.hwg' for ambiguity group 'longname'
Add 0 TPG and MUX2 nodes for internal arcs
Add 0 TPG and MUX2 nodes for external/PI arcs
Add 0 SCAN-SET nodes
Need 1 BILBO and MUX4 port sets
Intercept the following outputs and route to the BILBO nodes:
n1/O1
n1/O2

Add modules to graph 'test2.hwg' for ambiguity group 'norm'
Add 1 TPG and MUX2 nodes for internal arcs
Add 0 TPG and MUX2 nodes for external/PI arcs
Add 1 SCAN-SET nodes
Need 0 BILBO and MUX4 port sets
Reroute the following control inputs through the scan nodes:
n10/I1
Reroute the following data inputs through the TPG/MUX2 nodes:
n10/I0

Add modules to graph 'test2.hwg' for ambiguity group 'a1'
Add 1 TPG and MUX2 nodes for internal arcs
Add 0 TPG and MUX2 nodes for external/PI arcs
Add 1 SCAN-SET nodes
Need 0 BILBO and MUX4 port sets
Reroute the following control inputs through the scan nodes:

n8/I1
Reroute the following data inputs through the TPG/MUX2 nodes:
n8/I0

placebit tp_sa wirelist completed

Insert STANDARD HEADER here

BIT PLACEMENT RECOMMENDATION
(placebit tp_sa wirelist)

Test Point Monitoring with Data Compression

Add pattern generation and signature analysis on subsystem 'only', board 'first'

Add modules to graph 'test2.hwg' for ambiguity group 'ag9'
  Add 0 TPG and MUX2 nodes for internal arcs
  Add 0 TPG and MUX2 nodes for external/PI arcs
  Add 0 SCAN-SET nodes
  Need 1 BILBO and MUX4 port sets
  Connect MUX4/out(1-16) to BILBO/d(1-16)
Delete on board arc from n9/O1
    Create bus to route through
    Add arc from n9/O1 to new bus
    Add arc from new bus to n8/I0
    Add arc from new bus to MUX4/Ain2
Delete on board arc from n9/O2
    Create bus to route through
    Add arc from n9/O2 to new bus
    Add arc from new bus to n10/I0
    Add arc from new bus to MUX4/Ain3

Add modules to graph 'test2.hwg' for ambiguity group 'longname'
  Add 0 TPG and MUX2 nodes for internal arcs
  Add 0 TPG and MUX2 nodes for external/PI arcs
  Add 0 SCAN-SET nodes
  Need 1 BILBO and MUX4 port sets
  Connect MUX4/out(1-16) to BILBO/d(1-16)
Delete on board arc from n1/O1
    Create bus to route through
    Add arc from n1/O1 to new bus
    Add arc from new bus to n10/I1
    Add arc from new bus to MUX4/Ain2
Delete on board arc from n1/O2
    Create bus to route through
    Add arc from n1/O2 to new bus
    Add arc from new bus to n8/I1
    Add arc from new bus to MUX4/Ain3

Add modules to graph 'test2.hwg' for ambiguity group 'norm'
  Add 1 TPG and MUX2 nodes for internal arcs

```
  Add 0 TPG and MUX2 nodes for external/PI arcs
  Add 1 SCAN-SET nodes
  Need 0 BILBO and MUX4 port sets
  Connect internal TPG/out(1-16) to MUX2/Ain(1-16)
Delete control arc from n10/I1
   Add arc from n10/I1 to SCAN/uutin1
   Add arc from SCAN/in1 to n1/O1
Delete data arc from n10/I0
   Add arc from n10/I0 to MUX2/out1
   Add arc from MUX2/Bin1 to n9/O2

Add modules to graph 'test2.hwg' for ambiguity group 'a1'
  Add 1 TPG and MUX2 nodes for internal arcs
  Add 0 TPG and MUX2 nodes for external/PI arcs
  Add 1 SCAN-SET nodes
  Need 0 BILBO and MUX4 port sets
  Connect internal TPG/out(1-16) to MUX2/Ain(1-16)
Delete control arc from n8/I1
   Add arc from n8/I1 to SCAN/uutin1
   Add arc from SCAN/in1 to n1/O2
Delete data arc from n8/I0
   Add arc from n8/I0 to MUX2/out1
   Add arc from MUX2/Bin1 to n9/O1

placebit tp_sa wirelist completed
```

Additional Information.

See paragraphs describing

a. data base attributes and valid syntaxes (Paragraph 2.3.2)

b. results.tmp file (Paragraph 2.6.2.1)

c. output report header (Figure 2-5)

d. BIT techniques (Paragraph 3.4.2)

e. BIT modules (Paragraph 3.4.3)

BIT modules have their **Tbit** attribute set to "yes", and their **Tboard**, **Tsubsystem**, and **Tag_name** attributes set appropriately. This helps if the user tries to add a BIT technique after one has already been implemented. BIT modules will form the AG boundaries and BIT modules are ignored when forming AGs.

There is no difference in nodes added to implement odd/even generate/check parity. This information is acquired from the user only to make the output report more complete. The simulation with these modules will require accurate manipulation of the control lines.

Tarc_type is used to determine the precedence of a bus and associated interconnect. Clock lines have precedence over control lines, which, in turn, have precedence over

data lines. If conflicting **Tarc_types** are connected to a bus, the highest available precedence prevails. This affects the number of modules added in many techniques.

2.7.5. System Summary (compare). Comparisons of ADAS data bases can be performed with the System Summary (**compare**) process. This feature allows the user to save a baseline data base for comparing with altered versions of that baseline. This feature is most helpful when various design for testability / built-in test techniques have been used and the incremental hardware overhead costs associated with the techniques are desired.

The process evaluates the ADAS data base entries and attributes to assess incremental changes in the modules and input and output parameters. Comparisons are made of the number of modules, primary inputs/outputs, test point outputs, and number of BIT modules. Calculations of mean test time and mean ambiguity group size are also compared.

At any point in the TEA design methodology, **compare** can be used. Obviously, it is most useful at the end of the design cycle (e.g., a BIT technique has been successfully added to a functional representation of a system which has been checked for untestable constructs.), but the user can also use **compare** mid-cycle to see incremental changes in testability costs.

The user can gain insight into the incremental hardware overhead costs associated with a particular step in the cycle. The user may wish to ask questions such as: "how many primary inputs have I added to fulfill the test generation DFT guidelines?" or, "If I use this particular flip-flop, will I save any overhead (BIT) modules?" (Paragraph 3.4.3).

Another less obvious use for **compare** is while developing subgraph trees; **compare** does not need to check entire hierarchy trees; it can compare any two sets of hardware graphs. If emphasis has been placed on a particular subfunction, earlier and current views may be helpful in determining progress towards good testability.

The process **compare** can also be used to obtain output about the "current view" graph only.

Inputs.

The only input to **compare** is the directory and filename of the graph to be compared with the "current view."

Processing.

The following paragraph presents the high-level algorithms, special control features, and error handling features of **compare.**

```
/* Ask which data_base to compare to the current view */

Get(OtherDB)
Compare(CV)
If OtherDB != nil
        Compare(OtherDB)

Compare(Graph)
        Open(Graph)
```

```
While(more Nodes)
        Gen_AG_List(Graph, Node, AG_type)
Endwhile

While (more TSUBSYSTEMs)
        Print SUBSYSTEM name
        While (more TBOARDs)
                Print BOARD name
                While (more TSP_AG_NAME)
                        Print SP_AG_NAME, Testtime
                        Print NODE_NAME
                Endwhile (more TSP_AG_NAME)
                While (more TAG_NAME)
                        Print AG_NAME, Testtime
                        Print NODE_NAME
                Endwhile (more TAG_NAME)
                Print TBOARD Stats
        Endwhile (more TBOARDs)
        Print TSUBSYSTEM Stats
Endwhile. (more TSUBSYSTEMs)
```

Processing Notes

The following calculations were performed for the above process:

a. $$\text{Mean\_AG\_Size} = \frac{\sum \text{AG Sizes}}{\#\text{AG}}$$

b. $$\text{Mean\_Test\_Time} = \frac{\sum \text{Test Times}}{\#\text{AG}}.$$

Error Handling

The process starts and requests the name of the other data base for comparison to the "current view." If no other data base is named, the process is just run on the "current view." If the other named data base does not exist, the user is warned and the process is exited. If the other data base name is valid, information is gathered from each node of the current view data base for the various categories. Two flags are used to control the program flow which depend on the existence of attributes **Tsp_ag_name** or **Tag_name** and **Tag_test_time**. If neither **Tag_name** nor **Tsp_ag_name** (e.g., node assigned to group "other") is existent for a given **node_name**, calculation of the mean AG size and mean test time is not performed. A variable is incremented each time a new **Tag_test_time** is added to the 'Test_Time' variable and each time a new **Tag_name** is found. An error occurs if multiple test times have been set for a single ambiguity group. After analysis of each data base node, the two variables representing the Number_of_AGs are compared. If the numbers do not match, a warning is reported, indicating that the test time value will not be accurate due to a missing **Tag_test_time** attribute; a flag indicating this error is set for use during the report

generation. The same procedure is performed on the other data base. After all information has been accumulated, a report is generated. If both **Tag_name** and **Tsp_ag_name** are set. a warning is given and **Tsp_ag_name** is used.

Limitations.

ADAS data base graphs must already have been created before TEA is entered.

Biports are ignored.

TEA supports the concept of ADAS buses, but all buses are assumed to be a single bit in widt...

It is assumed that the user provides the ambiguity group test times. If an AG does not have a test time entered, the user is warned. The **Tag_test_time** attribute should only be entered once for each AG. If multiple values of a particular **Tag_test_time** are given, then an error is output.

Test times are integers. A test time value of '0' is interpreted as a non-existent test time.

Outputs.

At the completion of **compare**, a report is generated to illustrate the characteristics of the "current view" and any data base compared to it. Included in this report are

a. number of modules

b. number of primary inputs/outputs/test point outputs

c. mean time to test

d. mean ambiguity group size

e. a table of names of modules in each special ambiguity group

f. a table of names of modules in each ambiguity group

The following is a sample output report (.rpt file) for the **compare** tool. The "current view" data base is analyzed first. Errors are reported first. Here there are no errors; however, there could have been unreported or inconsistent test times. A "bad" test time means that conflicting test times were reported by at least two of the nodes in the AG. Following is a breakdown of each subsystem by boards.

Insert STANDARD HEADER here

SYSTEM SUMMARY
(compare)

Output report for graphs:
PROJECT:[TEA.LAD.COMPARE.GRAPHS]BIG1.HWG;3 and
PROJECT:[TEA.LAD.COMPARE.GRAPHS]BIG2.HWG;1

Totals for graph PROJECT:[TEA.LAD.COMPARE.GRAPHS]BIG1.HWG;3

Subsystem 'subone'

Board name	Node count	BIT node count	Mean test time	PI/PO count	TPO count	Mean AG size
b1	4	0	3.0	2	0	2.0
b2	3	0	9.5	2	0	1.5
b3	4	0	4.7	3	0	1.3
Total	11	0	17.2			

Totals for graph PROJECT:[TEA.LAD.COMPARE.GRAPHS]BIG2.HWG;1

Subsystem 'subone'

Board name	Node count	BIT node count	Mean test time	PI/PO count	TPO count	Mean AG size
b1	2	0	3.0	2	0	2.0
b2	3	0	9.5	2	0	1.5
b3	4	0	4.7	3	0	1.3
Total	90	17.2				

compare completed

TPOs are arcs with **Tdft_io** = tpo, test_point_output, test_pt_output. A PI/PO is an arc with **Tdft_io** labeled as pi, primary_input, po, primary_output. BIT nodes are nodes with **Tbit** = bit, bit_dt, bit_ps, mn, m_n, maintenance, y, yes.

The following is a sample output diagram file (.dwg file) for the same test graph. The "wirelist" will show all nodes that comprise each ambiguity group of the system and then each interconnection between boards is itemized. This information can be used to help debug a graph if inconsistent or unexpected results are noted.

Insert STANDARD HEADER here

SYSTEM SUMMARY
(compare)

Output wirelist for graphs:
PROJECT:[TEA.LAD.COMPARE.GRAPHS]BIG1.HWG;3 and
PROJECT:[TEA.LAD.COMPARE.GRAPHS]BIG2.HWG;1

Structure for graph PROJECT:[TEA.LAD.COMPARE.GRAPHS]BIG1.HWG;3

Subsystem subone
Board b1
AG Other-ag
Node n10
Node n00
AG ag1
Node n1
Node n0
Board b2
AG ag2
Node n4
Node n3
AG ag3
Node n6
Board b3
AG ag4
Node n9
AG ag5
Node n7
AG ag6
Node n2
Node n5

List of connections for graph PROJECT:[TEA.LAD.COMPARE.GRAPHS]BIG1.HWG;3

Connections on subsystem 'subone'
Connections on board 'b1'
Found a boardedge connection from 'n1/O0' to 'n3/I0'
Found a boardedge connection from 'n1/O0' to 'n7/I0'
Connections on board 'b2'
Found a boardedge connection from 'n1/O0' to 'n3/I0'
Found a boardedge connection from 'n6/O0' to 'n7/I1'
Connections on board 'b3'
Found an explicit PI/PO connection from 'n7/O1' to 'n9/I1'
Found an explicit PI/PO connection from 'n7/O1' to 'n2/I0'
Found an explicit PI/PO connection from 'n7/O1' to 'n5/I0'
Found a boardedge connection from 'n1/O0' to 'n7/I0'
Found a boardedge connection from 'n6/O0' to 'n7/I1'

Structure for graph PROJECT:[TEA.LAD.COMPARE.GRAPHS]BIG2.HWG;1

Subsystem subone

```
Board b1
  AG ag1
    Node n1
    Node n0
Board b2
  AG ag2
    Node n4
    Node n3
  AG ag3
    Node n6
Board b3
  AG ag4
    Node n9
  AG ag5
    Node n7
  AG ag6
    Node n2
    Node n5


List of connections for graph PROJECT:[TEA.LAD.COMPARE.GRAPHS]BIG2.HWG;1

  Connections on subsystem 'subone'
    Connections on board 'b1'
      Found a boardedge connection from 'n1/O0' to 'n3/I0'
      Found a boardedge connection from 'n1/O0' to 'n7/I0'
    Connections on board 'b2'
      Found a boardedge connection from 'n1/O0' to 'n3/I0'
      Found a boardedge connection from 'n6/O0' to 'n7/I1'
    Connections on board 'b3'
      Found an explicit PI/PO connection from 'n7/O1' to 'n9/I1'
      Found an explicit PI/PO connection from 'n7/O1' to 'n2/I0'
      Found an explicit PI/PO connection from 'n7/O1' to 'n5/I0'
      Found a boardedge connection from 'n1/O0' to 'n7/I0'
      Found a boardedge connection from 'n6/O0' to 'n7/I1'

compare completed
```

Error Message/Action.

Message	Action
Node %s had **Tag_name** and **Tsp_ag_name** set, **Tag_name** cleared	No action required
One or more of the AG test times has not been set	Set test times
Multiple times have been set for ambiguity group *name*	Check test times

Additional Information.

See paragraphs describing

a. data base attributes and valid syntaxes (Paragraph 2.3.2)

b. results.tmp file (Paragraph 2.6.2.1)

c. output report header (Figure 2-5)

3. DESIGN FOR TESTABILITY

3.1. Historical Background.

Ever since the invention of the integrated circuit in the early 1960's, the rush has been on to place as many devices as possible onto a single chip. With the movement into VLSI, the ability to test and verify the functionality and performance of electronic components and chips has become extremely difficult.

Similarly, the rising complexity of modern electronic systems has heightened the difficulty of testing and maintaining these systems. It is now generally regarded that the cost of system maintainability is a major component of the overall life cycle cost. For this reason, much effort has been focused on reducing the costs associated with system testing and fault isolation. Without the results of some of these efforts, the total life cycle costs of many new and complex weapon systems might be too large to warrant their development.

One of the approaches used to reduce testing costs is to recognize testing difficulties when designing the system, subsystem, board, or chip and to incorporate certain features in the design to help these matters. In other words, we must include the testability as a design parameter at all levels of the design hierarchy. This procedure has come to be known as design for testability (DFT) where testability is defined as a design characteristic which allows the status (operable, inoperable, or degraded) of a unit and the location of the faults within the unit to be confidently determined in a timely fashion [1].

Unfortunately in the past, the testing issue is usually not addressed until after the design has been nearly completed. By this time, the designer is reluctant to make even the slightest change as he has spent considerable effort to ensure that it has already met the functional and performance specifications. Management is likely to veto any major changes for testability due to the added design costs that will occur.

For this reason, it is necessary that a systematic testability methodology be included throughout the design process at all levels of the design hierarchy. Such a structured design for testability methodology was recently developed at Research Triangle Institute [2] and enables the designer to formulate a design that is testable by construction. This in turn has a large impact on the total life cycle costs of an electronic system.

The incorporation of design for testability methods in electronics first became practical with the onset of LSI logic in the 1970's. With the subsequent push into VLSI and the resulting decrease in hardware costs, the importance and impact of DFT methods on the overall life cycle costs became significant. With the rapid rise in complexity of modern weapon systems, it is not surprising that the military services were early advocates of incorporating DFT guidelines into these systems. The Navy, Air Force, and Army have devoted much effort in recent years to this task under the coordinating efforts of the Joint Logistics Commanders Panel on Automatic Testing. The Air Force has especially been active in this arena since the creation of the Rome Air Development Center (RADC) Testability Program in 1977. Since its inception, RADC has been

working to create an organized testability engineering discipline to address the inclusion of testability guidelines in the design procedure [3].

Though the military has devoted much effort to DFT, there was no integrated procedure for incorporating testability requirements in military programs until recently. This has been remedied by the issuance of MIL-STD-2165 in 1985 by the Navy Test and Monitoring Systems and the Joint Logistics Commander Panel on Automatic Testing [4]. This standard, along with other guidelines and standards, will define and coordinate the inclusion of testability requirements in electronic systems of the future.

In dealing with the testability of a design, there are two key underlying concepts, controllability and observability, that are often used. Controllability is defined as a relative measure of how easily a node or point can be forced to a specific logic value by setting the inputs. Likewise, observability is a measure of how easily the logical value of a node can be propagated through the unit to a primary output where it can be noted. Many of the common DFT techniques are used to enhance the controllability and observability of certain parts of a design to ease the testing problem.

Design for testability methods can generally be classified into three groups: ad hoc guidelines, structured methods, and built-in test techniques. Ad hoc guidelines are not formal or structured approaches, but are instead a set of general rules and guidelines that have been followed in the past. These "common sense" methods include partitioning, bus-oriented design, reset capabilities for sequential elements, and test points to improve the controllability and observability of hard-to-test logic blocks. On the other hand, structured methods are a group of methodical design rules which can be implemented to improve testability. These techniques are mostly variations of the scan path method where all sequential memory elements are linked together into a long shift register. By serially shifting in test data, the internal memory elements can be controlled to ease the testing problem. Finally, built-in test (BIT) methods take advantage of redundant and/or on-board test circuits to monitor the operation of the network. When an error is detected, the test circuit sets an error flag or no-go signal.

The slow acceptance of DFT methods in modern electronic systems can be partly attributed to the notion that testing and testability are a reliability and maintainability function and not a part of the design process [5]. As mentioned earlier, testing considerations were not usually considered until after the design was almost finished. By this time, it was too late to incorporate most DFT methods without incurring large additional design costs. This is because DFT methods also consume area, power, and slightly degrade the performance of a design. The costs associated with a redesign at this point that still met all the performance criteria were unjustifiable. This relegated few options to the test engineer to improve the testability of the network.

For this reason, it is imperative that DFT methods be included in the design process from the beginning when changes can be made with minimal cost. While it is true that the incorporation of a structured DFT methodology will incur additional design effort and hence costs, one must remember that in the long run design costs are only a portion of the overall life cycle costs of an electronic component or system.

It has been stated that the "primary benefit of testability is the positive impact it has on system maintenance" [6]. This is due to the reduction in test time necessary to isolate a fault to a specific ambiguity group. Since faulty components can be readily identified and replaced in less time, the availability of the system is increased. In addition, fault tolerant methods that make use of hardware redundancy can improve system reliability as well as availability.

The impact of testability can also reduce costs in the manufacturing phase. By improving the fault detection capabilities, faulty chips and components can be identified in less time before they are incorporated in a board or system. This also tends to have a favorable impact on system reliability, maintainability, and availability.

Future efforts in DFT technology must be devoted to developing and refining a systematic testability methodology, enforcement of testability requirements in all levels of the design hierarchy, and the development of tools to assist the design engineer. The Test Engineer's Assistant (TEA) is just such a tool that is designed to assess the costs, benefits, and drawbacks of a given DFT/BIT technique in conjunction with a design. Through these efforts, it will be possible to integrate the testability discipline fully into the design process.

References

[1] W. Keiner and R. West, "Testability Measures", AUTOTESTCON 1977, pp. 49-55.

[2] N. Kanopoulos, N. Vasanthavada, J.W. Watterson, and J.J. Hallenbeck, "Advanced CAD/E for Systems Testability", Final Report, Contract DAAB07-85-C-H079, TMDE/ETDL U.S. Army, Ft. Monmouth, NJ, prepared by Research Triangle Institute, Center for Digital Systems Research, June, 1987.

[3] W. Keiner and A. Coppola, "Joint Services Program in Design for Testability", Proc. Annual Reliability and Maintainability Symposium, 1981, pp. 268-271.

[4] Naval Electronic System Command, Military Standard 2165, "Testability Program for Electronics Systems and Equipment", January 1985.

[5] R. Monroe, "Design for Testability", AUTOTESTCON 1981, pp. 225-228.

[6] D. Allen and B. Ferch, "New Emphasis on Testability", AUTOTESTCON 1981, pp. 210-215.

3.2. TEA Board Level Design for Testability Guidelines.

3.2.1. G1 Guidelines - Aid to Test Pattern Generation.

G1 guidelines are established to aid the generation of test patterns.

Items listed in the Syntax section of each guideline refer to attribute values.

G1-01

G1-01 Guideline: Use flip-flops, counters, and shift registers with a Preset/Clear capability.

Why?

Initialization is a necessary precursor to any practical test and/or simulation program. Ability to initialize a circuit minimizes the states the circuit has to go through to reach a known state which is used as a basis for developing tests. Failure to provide for circuit initialization control, may result in long test times with a large number of test vectors.

How?

Select off-the-shelf components with Preset/Clear capability.

Syntax

inport_id: preset, pre, clear, clr

Tnode_type: counter, ctr, ff, flip_flop, flipflop, reg, register, shift_register, shiftregister, sr

Processing

The following paragraph presents the guideline procedure.

```
While (more node_names)
        Get(next node_name)
        If ((node_name is on user selected Tsubsystem) AND
                (node_name is on user selected Tboard))
                If (Tnode_type is (FF or CTR or SR))
                                While (more inport_ids)
                                        Get(next inport_id)
                                        If (inport_id is (PRE or CLR))
                                                set FOUNDONE
                                Endwhile
                                If (NOT FOUNDONE)
                                        report G1_01 Recommendation
                Endif
        Endif
Endwhile.
```

G1-02

G1-02 Guideline: Make Preset/Clear a primary input or primary input controllable.

Why?

Because this provides direct control of the initialization function to the tester and can be used to initialize several circuits at once with a single pulse during test.

How?

(a) Avoid this. Although the tester can access node A (primary input) to initialize this component, the outcome will be indeterminate when the tester releases A.

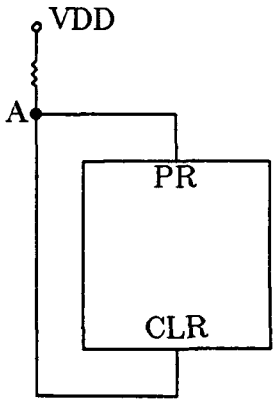


Figure 3-1. G1-02 Guideline Explanation: Avoid this Configuration

(b) Do one of these.

(i) The tester accesses this input directly or through other controlling circuitry.

(ii) The tester accesses the primary input and clears the circuit with override.

(iii) The circuit clears with the tester override and also on power-up.

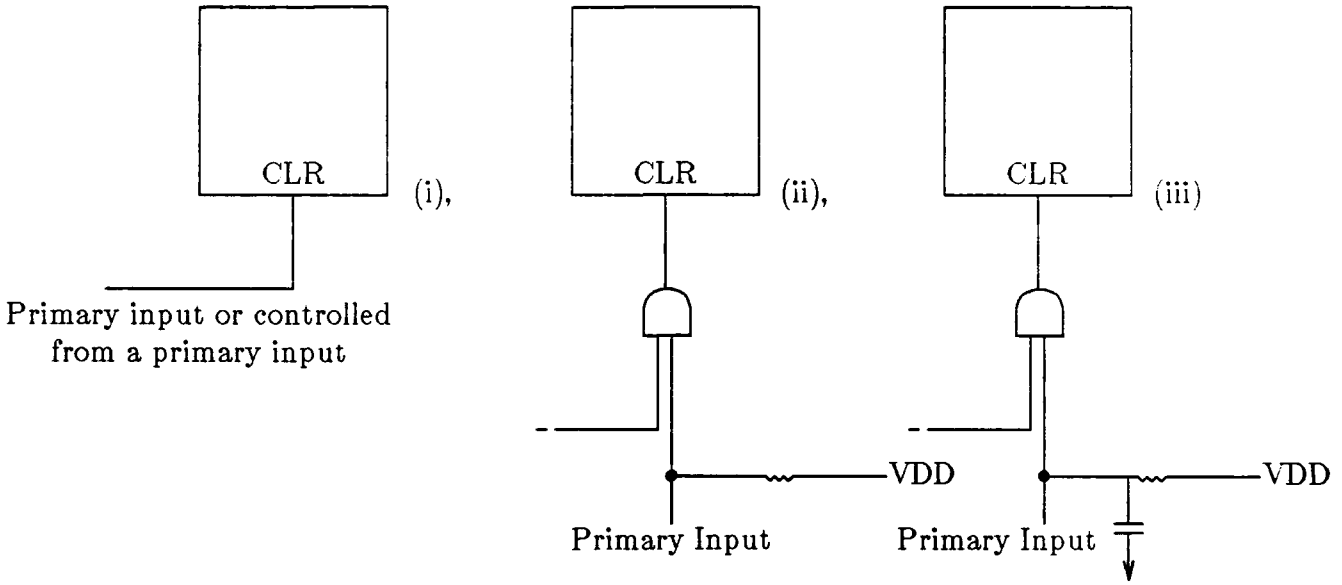


Figure 3-2. G1-02 Guideline Explanation: Do One of These

Syntax
inport_id: preset, pre, clear, clr
Tarc_type: clk, clk_gen, clock, clock_generator
Tnode_type: flip_flop, flipflop, ff, counter, ctr, shift_register, sr, shiftregister, reg, register

Notes

All inports labeled preset or clear are checked for primary input controllability.

Processing

The following paragraph presents the guideline procedure.

```
While (more node_names)
      Get(next node_name)
      If ((node_name is on user selected Tsubsystem) AND
            (node_name is on user selected Tboard))
            If (Tnode_type is (FF or CTR or SR))
                  While (more inport_ids)
                        Get(next inport_id)
                        If (inport_id is (PRE or CLR))
                              If (inport_id is NOT(PIC))
                              Report G1_02 Recommendation
                  Endwhile
Endwhile.
```

G1-03

G1-03 Guideline: Make Dselect of multiplexors a primary input or primary input controllable.

Why?

It improves circuit controllability and, indirectly, initialization and therefore can result in reduced number of test patterns and reduced test times.

How?

Select off-the-shelf components or:

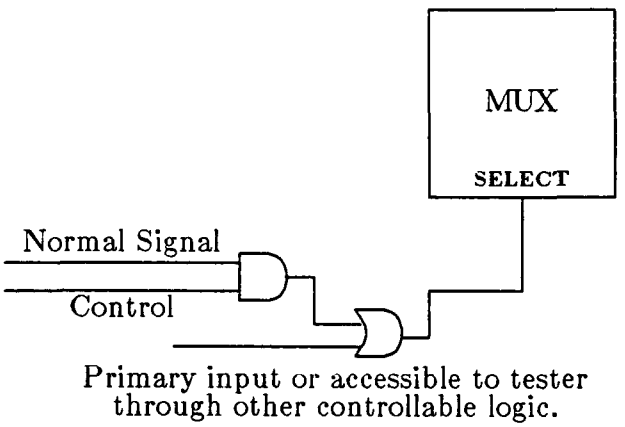


Figure 3-3. G1-03 Guideline Explanation

Syntax
inport_id: dselect, dsel, strobe
Tnode_type: multiplexer, multiplexor, mux, multiplex, scan, scannable, y, yes

Processing

The following paragraph presents the guideline procedure.

```
While (more node_names)
        Get(next node_name)
        If ((node_name is on user selected Tsubsystem) AND
                (node_name is on user selected Tboard))
                If (Tnode_type is (MULTIPLEXER))
                        While (more inport_ids)
                                Get(next inport_id)
                                If (inport_id is (DESELECT))
                                        If (inport_id is NOT(PIC))
                                        Report G1_03 Recommendation
                        Endwhile
                        If (no DESELECT found)
                                Report G1_03 Error
Endwhile.
```

G1-04

G1-04 Guideline: Make Tristate Control a primary input or primary input controllable.

Why?

It improves circuit controllabiiity and, indirectly, initialization and therefore can result in reduced number of test patterns and reduced test times.

How?

Select off-the-shelf components.

Syntax
inport_id: trictrl, tctrl
Ttri_state: tristate, tri, yes, y

Processing

The following paragraph presents the guideline procedure.

```
While (more node_names)
      Get(next node_name)
      If ((node_name is on user selected Tsubsystem) AND
            (node_name is on user selected Tboard))
            If (Ttri_state is (TRISTATE))
                  While (more inport_ids)
                        Get (next inport_id)
                        If (inport_id is (TRISTATE_CONTROL))
                              If (inport_id is NOT(PIC))
                              Report G1_04 Recommendation
                  Endwhile
                  If (no TRISTATE_CONTROL found)
                        Report G1_04 Error
Endwhile.
```

G1-05

G1-05 Guideline: Make Enable/Hold of microprocessors a primary input or primary input controllable.

Why?

It improves circuit controllability and, indirectly, initialization and therefore can result in reduced number of test patterns and reduced test times.

How?

Select off-the-shelf components.

Syntax
inport_id: enable, hold
Tnode_type: microprocessor, up, uproc, u_proc

Processing

The following paragraph presents the guideline procedure.

```
While (more node_names)
      Get(next node_name)
      If ((node_name is on user selected Tsubsystem) AND
            (node_name is on user selected Tboard))
            If (Tnode_type is (uPROCESSOR))
                  While (more inport_ids)
                        Get(next inport_id)
                        If (inport_id is (ENABLE or HOLD))
                              If (inport_id is NOT(PIC))
                              Report G1_05 Recommendation
                  Endwhile
                  If (no ENABLE or HOLD found)
                        Report G1_05 Error
Endwhile.
```

G1-06

G1-06 Guideline: Make Chip Select (CS), Address Latch Enable (ALE), Read, and Write of memories primary inputs or primary input controllable.

Why?

It improves circuit controllability and, indirectly, initialization and therefore can result in reduced number of test patterns and reduced test times.

How?

Select off-the-shelf components.

Syntax
inport_id: cs, chip_sel, chip_select, address_latch_enable, add_latch_en, ale, write, wr, read, rd, read_write, readwrite, rw
Tnode_type: memory, mem, dram, sram, ram, prom, fifo, rom

Processing

The following paragraph presents the guideline procedure.

```
While (more node_names)
        Get(next node_name)
        If ((node_name is on user selected Tsubsystem) AND
                (node_name is on user selected Tboard))
                If (Tnode_type is (DRAM or SRAM or RAM or ROM or PROM or FIFO))
                                While (more inport_ids)
                                        Get(next inport_id)
                                        If (inport_id is (CS or ALE or RW or RD))
                                                If (inport_id is NOT(PIC))
                                                Report G1_06 Recommendation
                                        Else if (Tnode_type is (ROM))
                                                Report G1_06 Error
                                        Else if (inport_id is (WR))
                                                If (inport_id is NOT(PIC))
                                                Report G1_06 Recommendation
                                        Else
                                                Report G1_06 Error
                                Endwhile
Endwhile.
```

G1-07

G1-07 Guideline: Make Control Access of any bus structure a primary input or primary input controllable.

Why?

It improves circuit controllability and, indirectly, initialization and therefore can result in reduced number of test patterns and reduced test times.

How?

Select off-the-shelf components.

Syntax
inport_id: control, cntrl, ctrl, bus_control, bus_ctrl, bus, busnode, bus_node

Processing

The following paragraph presents the guideline procedure.

```
While (more node_names)
      Get(next node_name)
      If ((node_name is on user selected Tsubsystem) AND
            (node_name is on user selected Tboard))
            If (Tnode_type is (BUS_CONTROL))
                  While (more inport_ids)
                        Get(next inport_id)
                        If (inport_id is (BUS_CONTROL))
                              If (inport_id is NOT(PIC))
                              Report G1_07 Recommendation
                  Endwhile
Endwhile.
```

G1-08

G1-08 Guideline: Make buried (i.e., not primary input or primary input controllable) control lines of the types Preset/Clear: Dselect of multiplexors: Tristate Control: Enable/Hold of microprocessors; chip select (CS), address latch enable (ALE). Read. and Write of memories; and Control Access of bus structures primary outputs.

Why?

Because it increases the observability of the circuit control and subsequently the fault isolation capability of the test. This is particularly true if a control signal is distributed to more than one ambiguity group. In such a case, a buried control line must become observable at a primary output.

How?

Bring the control line in question to an unused pin to the board I/O.

Syntax

Processing

The following paragraph presents the guideline procedure.

```
While (more node_names)
      Get(next node_name)
      If ((node_name is on user selected Tsubsystem) AND
            (node_name is on user selected Tboard))
            While (more inport_ids)
                  Get(next inport_id)
                  If (Tarc_type is (CONTROL))
                        If (inport_id is NOT(PIC))
                              Report G1_08 Recommendation
            Endwhile
Endwhile.
```

G1-09

G1-09 Guideline: Make the output of all logic redundant nodes a primary output.

Why?

Because errors on redundant nodes are, in general, undetectable. An error on a redundant node can cause an otherwise detectable error to be undetectable. Therefore, there is a need to be able to generate tests for errors on redundant nodes.

How?

Make redundant nodes primary outputs or use one of the scan BIT modules that can serialize a parallel path and thus, provide observability of a wide path through a single pin.

Notes

Two recommendations are available.

Certain conditions must be met to identify logic redundant nodes. Logic redundant nodes have identical **Tnode_types** (non-blank) and have the same list of predecessors. A "strong" warning is issued if the **Tnode_spec_types** match (non-blank) and the ordering of inputs is maintained, otherwise a "weak" warning is issued.

Processing

The following paragraph presents the guideline procedure.

```
while (more nodes)
  If (node is fanout)
        Find the children of the fanout node
        Find the list of parents for each child
        Sort the list of parents for each child
              Alphabetize list
              Remove duplicates
        For each of the children with identical sorted parent lists
              If Tnode_spec_type of children do not match
                                Report G1_09_weak Recommendation
              Else
                                If unsorted parent list match
                                      Report G1_09_strong Recommendation
                                Else
                                      No warning
              If (node not OO)
                                Report G1_09 Recommendation

        Endfor
Endwhile.
```

G1-10

G1-10 Guideline: Make the trunk section of high fanout nodes or junctions a primary output.

The threshold for this guideline is stored in the system test requirements; the default value is 5.

Why?

To increase the observability of the fanout node. An error on this node will decrease the fault isolation capability of a test.

How?

Make this node a primary output.

NOTE: Use this guideline if you don't use guideline G2-02.

Syntax
Tnode_type: fanout, copy, null

Processing

The following paragraph presents the guideline procedure.

```
While (more nodes)
      Get(next node)
      If ((name is on user selected Tsubsystem) AND
            (name is on user selected Tboard))
            If (large fanout device)
                  If (device is NOT OO)
                        Report G1_10 Recommendation
Endwhile.
```

G1-11

G1-11 Guideline: Terminate all unused device inputs and tristatable outputs.

Why?

From a design point of view, unused inputs to logic devices should always be terminated to VDD or Ground through a suitable resistor to remove the risk of noise that can be picked up by floating inputs.

From a testing point of view, termination through a proper resistor may allow control of the device's unused inputs by the tester. Termination of tri-state device outputs through a pull-up resistor can prevent inconsistent logic values leading to inconsistent node signatures.

How?

Use an appropriate resistor connected to the appropriate logic level.

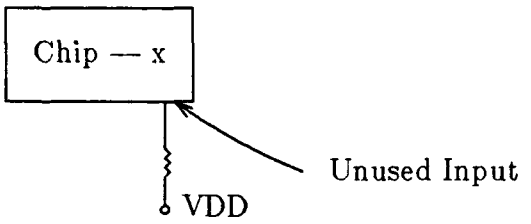


Figure 3-4. G1-11 Guideline Explanation

Syntax

Ttri_state: tristate. tri, yes, y

Processing

The following paragraph presents the guideline procedure.

```
While (more node_names)
      Get(next node_name)
      If ((node_name is on user selected Tsubsystem) AND
            (node_name is on user selected Tboard))
            While (more inport_ids)
                  Get(next inport_id)
                  If (inport_id has no source)
                        Report G1_11 Recommendation
            Endwhile
            If (Ttri_state is (TRI_STATE)
                  While (more outport_ids)
                        Get(next outport_id)
                        If (outport_id has no sink)
                              Report G1_11 Recommendation
                  Endwhile
Endwhile.
```

G1-12

G1-12 Guideline: Keep analog and digital circuits physically apart.

Why?

For two reasons:

(a) To avoid crosstalk problems in digital lines which run close to analog lines

(b) The testing requirements and strategies for analog circuits are substantially different from those for digital circuits and therefore it is preferable to test them separately.

How?

Use physical separation when it is necessary to mix digital and analog on the same board. Use separate boards if possible. If digital signals have to be routed close to analog lines, then the digital lines should be properly balanced and shielded transmission lines.

Syntax

Tdevice_type: a, analog, d, digital, mix

Processing

The following paragraph presents the guideline procedure.

```
While (more node_names)
        Get(next node_name)
        If ((node_name is on user selected Tsubsystem) AND
                (node_name is on user selected Tboard))
                If (Tdev_type is (ANALOG))
                                Append to analog_list
                Else
                                Append to digital_list
Endwhile

If (analog_list AND digital_list NOT(EMPTY))
        Report G1_12 Recommendation
End.
```

G1-13

G1-13 Guideline: Make analog lines used as input to digital data acquisition circuits a primary output.

Why?

Because in this way the analog and digital sections of these circuits can be tested separately and with different test equipment, if necessary.

How?

For the case of an analog signal, make its line a primary output. In the case of digital inputs to a D/A, either make them primary outputs or use a scan-set module to sample the signal in parallel and shift it out serially thus minimizing pin overhead.

Syntax
Ta_d: a, analog, d, digital
Tdevice_type: a, analog, d, digital, mix
Tnode_type: da, data_acquisition, mix

Processing

The following paragraph presents the guideline procedure.

```
While (more node_names)
      Get(next node_name)
      If ((node_name is on user selected Tsubsystem) AND
            (node_name is on user selected Tboard))
            If (Tnode_type is (DIGITAL_DA))
                  While (more inport_ids)
                        Get(next inport_id)
                        If (Ta_d is (ANALOG))
                              If (outport_id is NOT OO)
                              Report G1_13 Recommendation
                  Endwhile
Endwhile.
```

G1-14

G1-14 Guideline: Avoid asynchronous logic.

Why?

Although the performance benefits can be significant by using asynchronous logic, the testing problems can also be significant. This is due primarily to the possibility of races introducing non-deterministic behavior. Providing test patterns for such circuitry can be very difficult.

How?

Use synchronous counters rather than asynchronous ripple counters or even asynchronously-coupled synchronous counters. Avoid self-timed logic as much as possible and tie everything to the clock or its derivatives as much as possible.

Syntax
inport_id: clock, clk
Tarc_type: clock, clk, clock_generator, clk_gen
Tasynch: async, asynch, asynchronous, y, yes
Tdevice_type: a, analog, d, digital
Tlogic_type: combinational, comb_logic, comb
Tnode_type: clock, clk, clock_generator, clk_gen

Note

If the user does not eliminate an asynchronous construction and wants it identified by the other tools, the user should set the **Tasynch** attribute on the identified nodes.

Also, this guideline is not intended to identify intentionally used asynchronous logic; therefore, nodes with **Tasynch** set affirmatively will not be flagged.

Processing

The following paragraph presents the guideline procedure.

```
While (more node_names)
      Get(next node_name)
      If ((node_name is on user selected Tsubsystem) AND
            (node_name is on user selected Tboard) AND
            (NOT VISITED))
            node is VISITED
            While (more inport_ids)
                  Get(next inport_id)
                  If (inport_id is (CLOCK))
                        If (inport_id source is (COMB_LOGIC))
                              Get(source node)
                              While (more inport_ids)
                              Get (next inport_id)
                              If (inport_id is ancestor of clock)
                              Report G1_14 Recommendation1
                              Else
                              Report G1_14 Recommendation2
                              Endwhile
            Endwhile
Endwhile.
```

G1-15

G1-15 Guideline: Avoid uncontrollable feedback.

Why?

Uncontrollable feedback paths complicate both test generation and fault diagnosis (isolation). In some circuits, establishing a known state may be difficult in presence of uncontrollable feedback paths.

How?

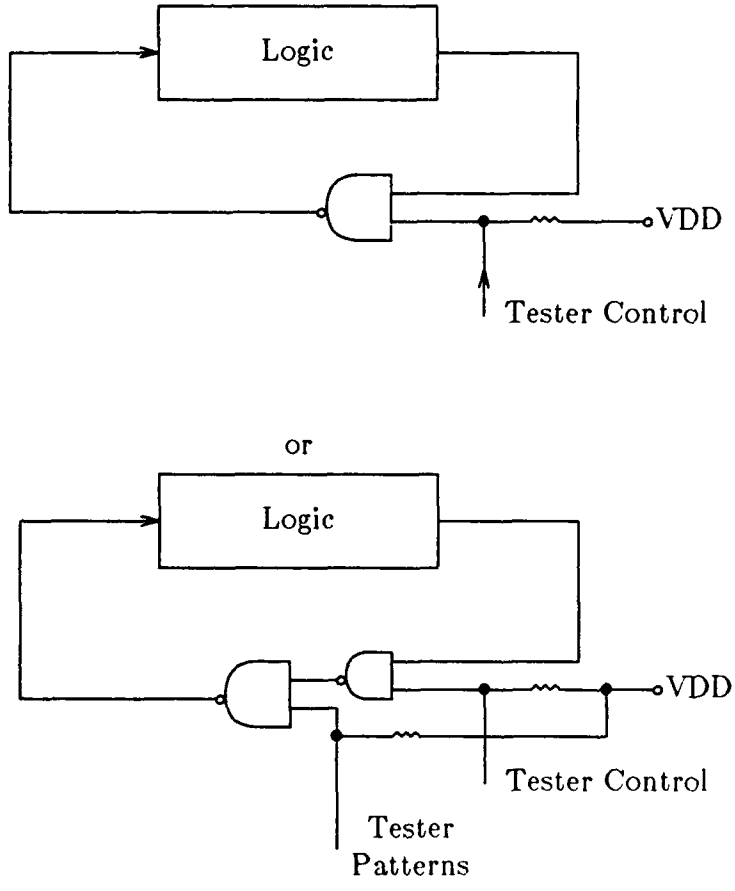


Figure 3-5. G1-15 Guideline Explanation

Hints.

a. If all nodes in a loop are combinational, an error condition exists.

b. If **Tlogic_type** is blank, it is assumed that the node is sequential.

c. If a loop is self-contained on a single board without any primary inputs/outputs, an error condition exists.

d. A node with no inports is considered primary input controllable.

e. Don't set the **Tboard** and **Tsubsystem** attributes on a fanout node.

f. All controllable loops *must* have one primary input controllable arc.

g. Small loops that are part of bigger loops will not be detected.

Procedure for finding loops

```
while(more nodes)
        pick a node
        find all nodes you can reach from this node
        if you can get back from the new node to
                the original node, it's part of a loop
```

Processing

The following paragraph presents the guideline procedure.

```
Find (nodes connected by data arcs)
Get (node from loop)
While (more inports)
      If (inport is NOT part of loop)
            If (inport is NOT PIC)
                        Report G1_15 Recommendation
      Goto next loop of nodes
Endwhile.
```

G1-16

G1-16 Guideline: Avoid one-shots as delay elements.

Why?

One-shots drift and can give undependable timing pulses.

How?

Use counters or timers as delay elements.

Syntax
Tnode_type: oneshot, one_shot

Processing

The following paragraph presents the guideline procedure.

```
While (more node_names)
        Get(next node_name)
        If ((node_name is on user selected Tsubsystem) AND
                (node_name is on user selected Tboard))
                If (Tnode_type is (ONE_SHOT))
                                Report G1_16 Recommendation
Endwhile.
```

3.2.2. G2 Guidelines - Aid to Test Pattern Application. G2 guidelines are established to aid the application of test patterns.

G2-01

G2-01 Guideline: Make all on-board clocks a primary input or primary input controllable.

Why?

If a circuit contains an on-board free-running clock (oscillator), it may be necessary to replace it with a tester-provided clock to reduce the speed of the circuit (if necessary) for testing purposes or to debug the test patterns.

How?

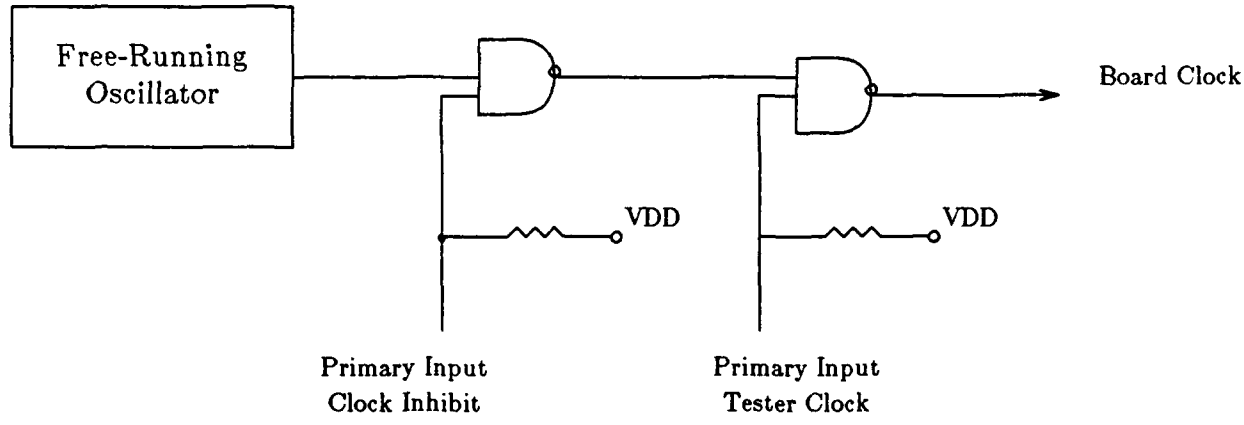


Figure 3-6. G2-01 Guideline Explanation

Syntax
outport_id: clock, clk
Tnode_type: clk_gen, clock_generator, clk, clock

Processing

The following paragraph presents the guideline procedure.

```
While (more node_names)
      Get(next node_name)
      If ((node_name is on user selected Tsubsystem) AND
            (node_name is on user selected Tboard))
            If (Tnode_type is (CLOCK))
                  While (more outport_ids)
                        Get(next outport_id)
                        If (outport_id is (CLOCK))
                              If (outport_id is NOT(OO))
                              Report G2_01 Recommendation
                  Endwhile
Endwhile.
```

G2-02

G2-02 Guideline: Avoid fanout greater than 5.

Why?

High fanout lines decrease the fault isolation capability of a test set.

How?

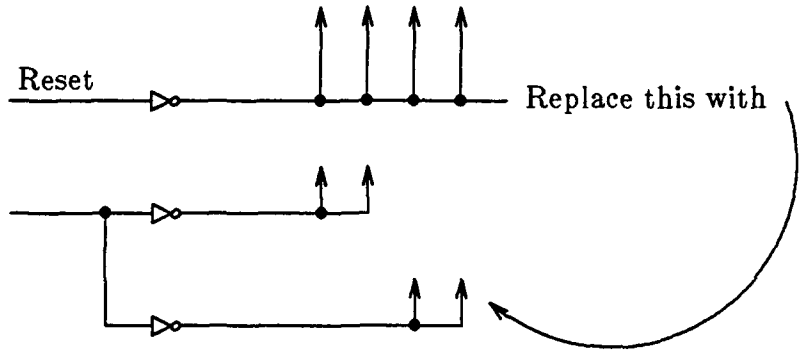


Figure 3-7. G2-02 Guideline Explanation

Processing

The following paragraph presents the guideline procedure.

```
While (more node_names or bus_names)
        Get(next name)
        If ((name is on user selected Tsubsystem) AND
                (name is on user selected Tboard))
                If (fanout device)
                                While (more outport_ids)
                                        Get(next outport_id)
                                        Find children of fanout
                                        If (# of children > 5)
                                                Report G2_02 Recommendation
                                Endwhile
Endwhile.
```

G2-03

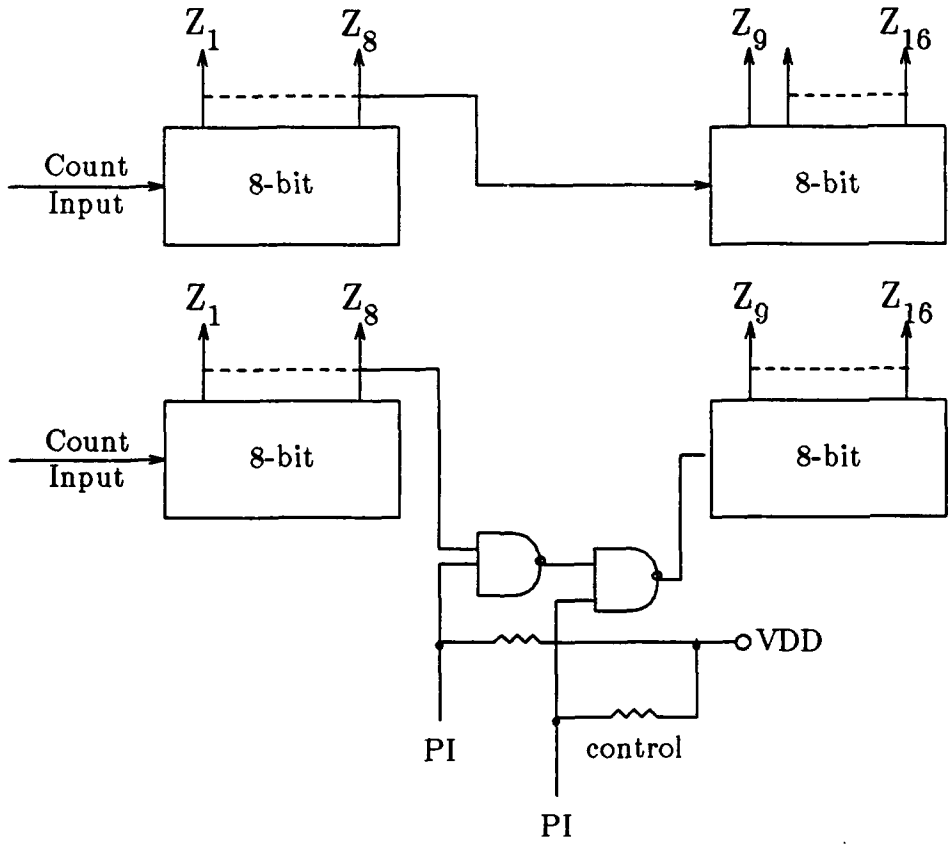
G2-03 Guideline: Break up long counter chains greater than 8 bits with logic that contains primary inputs or primary input controllable lines.

Why?

Because this results in reduction of the number of test patterns required to fully test the counter.

NOTE: Follow this guideline when you use more than one counter chip to form long counters. Do not replace a single chip (long) counter with several short counter chips.

How?



(Controlled from PI. In this case the resistors and the connection to VDD can be removed.)

Figure 3-8. G2-03 Guideline Explanation

Syntax
inport_id: clock, clk
outport_id: co, carry_out, carry
Tnode_type: counter, ctr, count

Processing

The following paragraph presents the guideline procedure.

```
While (more node_names)
        Get(next node_name)
        If ((node_name is on user selected Tsubsystem) AND
                (node_name is on user selected Tboard))
                While (Tnode_type is (COUNTER))
                        Append (node_name to Counter_List)
Endwhile

Cleanup (lists)

While (Counter_lists)
        Clear total_count_bits
        While (more nodes on Counter_List)
                Get (Node_name off Counter_List)
                If ((next node_type is NOT combinational) AND
                        (next node is NOT PIC))
                        If (next node is COUNTER)
                                add Tcount_bits to total_count_bits
        Endwhile

        If (total_count_bits > 8)
                Report G2_03 Recommendation
Endwhile.
```

italics

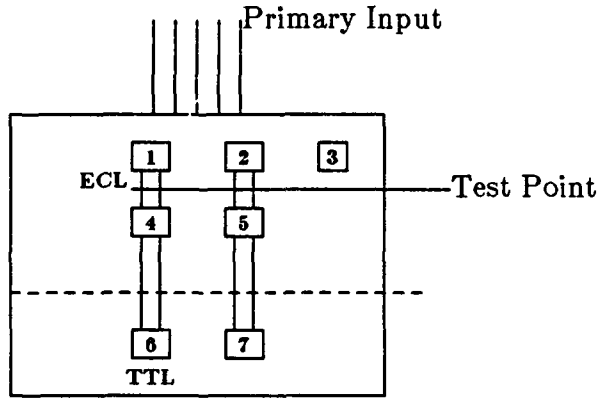
G2-04

G2-04 Guideline: Avoid mixed logic families on the same board unless all I/O is compatible in logic levels and rise and fall times.

Why?

I/O level capability of all components on a board makes it feasible for the tester to supply the required logic levels. It will be difficult for the tester to supply tests to components expecting different logic levels during the run of a single test set.

How?



Choose 1,2,3 to also be TTL-compatible so the same test set can be applied or monitored at the TP interface.

Figure 3-9. G2-04 Guideline Explanation

Syntax
Tlogic_family: ttl, iil, ecl, gaas, cmos, nmos, other

Processing

The following paragraph presents the guideline procedure.

```
Logic_list = NULL

While (more node_names)
      Get(next node_name)
      If ((node_name is on user selected Tsubsystem) AND
            (node_name is on user selected Tboard))
            Get (Tlogic_family of node_name)
            Append (Tlogic_family to logic_list)
            Remove (duplicates from logic_list)
Endwhile

If (logic_list > 1 Tlogic_family)
      Report G2_04 Recommendation
```

G2-05

G2-05 Guideline: Limit chip fanout at test points to one less than the specified maximum.

Why?

To allow for the tester loading when the tester accesses the test points.

How?

After test points have been determined, check the chip fanouts against the chip data sheets to identify violations of this guidelines.

Syntax
Tnode_type: fanout, copy, null

Processing

The following paragraph presents the guideline procedure.

```
While (more node_names)
      Get(next node_name)
      If ((node_name is on user selected Tsubsystem) AND
            (node_name is on user selected Tboard))
            While (more outport_ids)
                  Get(next outport_id)
                  If (Tdft_io is (TPO))
                        Find children of node_name
                        If (# children > of node_name_Tchip_maxfan - 1)
                              Report G2_05 Recommendation
            Endwhile
Endwhile.
```

G2-06

G2-06 Guideline: Provide the refresh circuitry for DRAMs on the same board as the RAM.

Why?

If the DRAM refresh circuitry is not on-board, then the tester has to provide it and this can be a problem for two reasons:

(a) The tester may not be fast enough to both provide the refresh cycle and also perform the required tests.

(b) In terms of test programs, there is a need to incorporate regular calls to refresh subroutines.

How?

By design, place the refresh circuitry on the memory board.

Syntax
Tnode_type: dram

Processing

The following paragraph presents the guideline procedure.

```
While (more node_names)
      Get(next node_name)
      If ((node_name is on user selected Tsubsystem) AND
            (node_name is on user selected Tboard))
            If (Tnode_type is (DRAM))
                  Report G2_06 Recommendation
Endwhile.
```

3.2.3. Recommended Order of Application. The user has the opportunity to run all or any of the DFT guidelines on each of the boards of his system. If some resource is limited (e.g.. computer time or budget), RTI system designers recommend running the G1 guidelines in the following order

a. G1-15
b. G1-01
c. G1-14
d. G1-11
e. G1-02
f. G1-03
g. G1-04
h. G1-05
i. G1-07
j. G1-06
k. G1-08
l. G1-09
m. G1-10
n. G1-13
o. G1-12
p. G1-16

and the G2 guidelines in the following order

a. G2-01
b. G2-06
c. G2-03
d. G2-04
e. G2-05
f. G2-02 (only need if G1-10 not run)

The above order of application only has significance if the guidelines are run one at a time. If multiple guidelines are run at the same time, TEA determines the order of application -- G1. the G2 guidelines, starting low to high.

3.2.4. User-defined Guidelines. When the user wants to add his own design for testability guidelines to those known by the Design for Testability Guideline Checker (dft -- Paragraph 2.7.1), he must

a. write prolog code to implement guideline

b. write a report generator to interface with the report access utility (Paragraph 2.6.2)

c. either add documentation about the guideline to the TEA-supplied VMS help library (Paragraph 2.6.3) or provide some other information source to users

d. let **dft** know about the existence of the code by setting a VMS logical. **tea_rules**. to point to the file that contains the names of the user-supplied guidelines. one per line.

An example of the procedure to let **dft** know about the existence of a user-supplied guideline follows. The code that follows is an example of a guideline that could be added to the guideline data base. It is a rewrite of an existing guideline. so this example can be copied and run, as is, including the .com and .txt files. This file should be called my_rules.pl.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                                                                            %%
%% Dispatch code for a selected rule.                                         %%
%%                                                                            %%
%% The main routine for DFT analyze calls routines like the one below %%
%% for each rule selected from the main menu. The first parameter to %%
%% the dispatch clause is the name of the rule and the second is the %%
%% file pointer to the results.tmp file.                                      %%
%%                                                                            %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
dft_analyze_dispatch_rule(my_rule, Outfile) :-
   write('running my_rule'), nl,
   write(Outfile,'*my_rule*'), nl(Outfile),
   do_my_rule(Outfile),
   write(Outfile,'*my_rule_END*'), nl(Outfile).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                                                                        %%
%%  Main body of rule definition.                                         %%
%%                                                                        %%
%%  This rule will find nodes with a fanout from a given port of %%
%%  greater than 1 (i.e. any port that connects to a fanout node %%
%%  or bus)                                                               %%
%%                                                                        %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
do_my_rule(Outfile) :-
   get_valid_node(Node),
   outputs(Node, Outputs),
   check_high_fanout(Outfile, Node, Outputs, 0),
   fail.
do_my_rule(_).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                                                                      %
%%  Check each outport of a node for a high fanout value.              %
%%                                                                      %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

check_high_fanout(Outfile, Node, [Head|Tail], Port_num) :-
    length(Head, Fanout),
    Fanout =< 1,
    New_port_num is Port_num + 1,
    !,
    check_high_fanout(Outfile, Node, Tail, New_port_num).

check_high_fanout(Outfile, Node, [_|Tail], Port_num) :-
    write(Outfile, 'Node '),
    write(Outfile, Node),
    write(Outfile, ' port '),
    write(Outfile, Port_num),
    write(Outfile, ' has fanout greater than 1'),
    nl(Outfile),
    New_port_num is Port_num + 1,
    !,
    check_high_fanout(Outfile, Node, Tail, New_port_num).
```

This is the text of the file called my_rules.txt. This file will be used by the command file to set the VMS logical tea_rules. The command file is listed below.

```
my_rule
```

This is the text of the file called my_rules.com. This file is run by typing

```
$ @my_rules.com
```

This file sets the VMS logical tea_rules to find the name of the new guideline to be added to the guideline data base. In this example, the logical will point to the file my_rules.txt, given above.

```
$ !                                                                    !
$ !   Define TEA_RULES to tell TEA where to find the list of my rules.  !
$ !   If the servant is built, you only need to define this logical, the second  !
$ !   half of this file is for building the prolog servant file.       !
$ !                                                                    !
$       define tea_rules my_rules.txt
$ !                                                                    !
$ !   Build a prolog servant file with my rules added in the current directory.  !
$ !   Uses shortcut method to start prolog and pass it commands from a command  !
$ !   file.                                                            !
$ !                                                                    !
$       quintus_engine tea_prolog:dft.prolog
          [my_rules].
          ['tea_prolog:build_servant'].
          halt.
$       exit
```

The user is directed to the VMS manual set for information about adding information to a help library. The user will not be able to access his information through **dft explain**, but will be able to access it through normal use of the On-line User Support System (Paragraph 2.6.3).

The user is given no guidance in writing prolog code with TEA. Refer to the Quintus manuals.

Appendix D shows ADAS-related information, including data base file formats and data base access routines.

Warnings to the User

a. No user-defined guideline should have the character "%" in its name. There is a possibility that there would be a conflict with TEA-supplied menu items.

b. The user will have no interface to the report generator (Paragraph 2.6.2) unless he provides it. It is recommended that the user generate his own report, with a filename ending with the extensions .rpt, .dwg, or .help so that the report access utility can print the user's reports to the screen or printer automatically from the TEA user interface.

3.3. TEA Hierarchical Approach to Design for Testability. The reader is directed back to Figure 2-4 for this discussion. Figure 2-4 illustrates the TEA concept of a testable system. On-board BIT support modules are shown communicating to an on-board maintenance manager which, in turn, talks to a maintenance bus to report board test results. A test control unit monitors the subsystem test results for the system and may also control/communicate with any automated test equipment needed to apply tests.

One possible scenario involves the user signaling the system test control unit that it is to run a standard diagnostic. The control unit sends necessary signals along the test bus to signal each subsystem to start its diagnostic procedure. The subsystem test control unit then provides input vectors (from some storage area) and/or control signals to each board to begin the board testing. The maintenance node may issue start commands to self-testable chips, feed scan chains with appropriate input data and control and start test pattern generators for the rest of the circuitry. When an appropriate amount of time has passed, the controller polls each board maintenance node for test results, which may include a faulty ambiguity group, and this information is passed to the system test controller for relay to the user.

This system of hierarchical responsibility and reporting supports a testable design in both fault detection and fault isolation. This system is the result of applying the TEA methodology described in Paragraph 2.

TEA supplies a library of BIT support modules (Paragraph 3.4.2) to provide the kind of tasks necessary to implement a board test strategy, including one version of a board maintenance controller, which may be used to control the BIT support modules. TEA recognizes the Element Test and Maintenance (ETM, from the VHSIC Phase II contractors) and Joint Test Action Group (JTAG) bus structures at the board level.

The following figures are used to illustrate the standard configurations that TEA recognizes. TEA always wants to see either of

a. an ETM or JTAG support module

b. primary input and outputs

c. a maintenance node

at the head and tail of these rings and stars.

Figures 3-10 and 3-11 show the ETM configurations. The nodes must have their **Tscan** attribute set to **etm** or **vhsic** and must have inports and outports named just as they are in the figure. Depending on the particular configuration, ring or star, either "DATAIN"/"DATAOUT" are chained or bused. If "DATAIN"/"DATAOUT" is chained (ETM ring), then there is a single SELECT line. There is always a bused JMODE line. (Note: "MODE" is a reserved word in Quintus Prolog so TEA looks for "JMODE" instead of "MODE" as in the standard.)

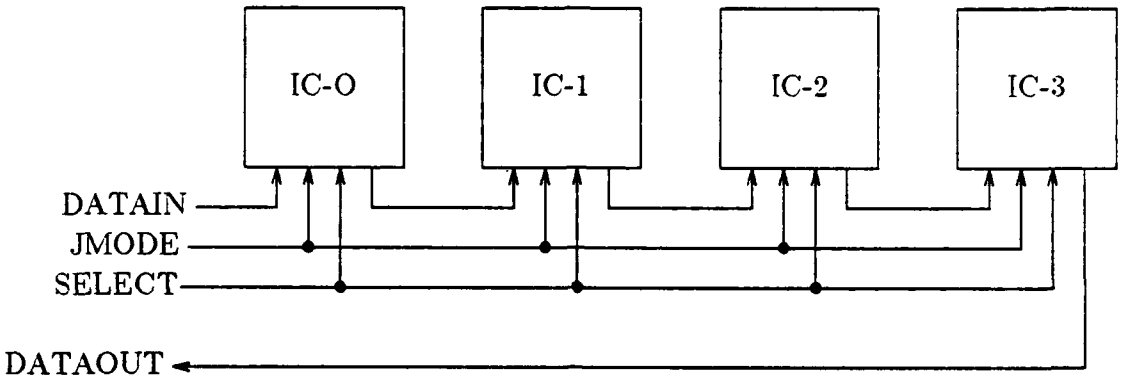


Figure 3-10. ETM Ring Configuration

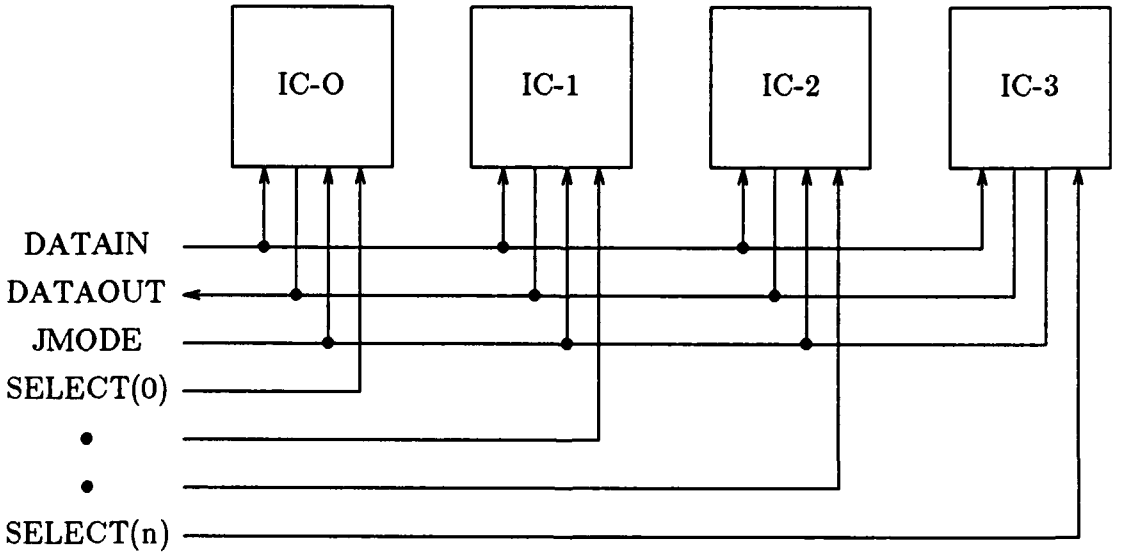


Figure 3-11. ETM Star Configuration

Figures 3-12 and 3-13 show the JTAG configurations. The nodes must have their **Tscan** attribute set to jtag and must have inports and outports named just as they are in the figure. Depending on the particular configuration, ring or star, either "SDI"/"SDO" are chained or bused. If "SDI"/"SDO" is chained (JTAG ring), then there is a single JMODE line. There is always a bused SCK line.

The user needs to carefully study his requirements and limitations and decide on subsystem and system level test bus protocols and test control units. This is not done by TEA.

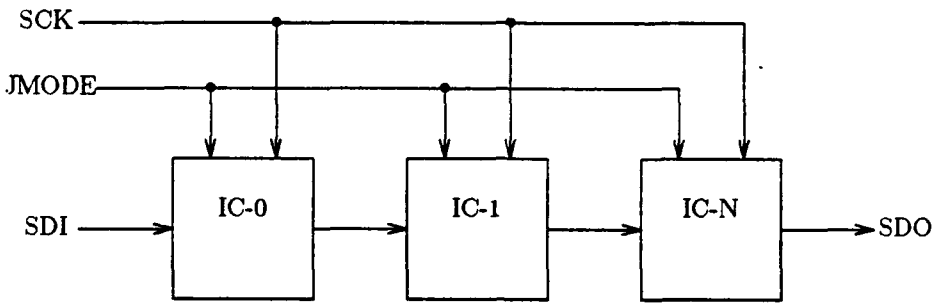


Figure 3-12. JTAG Ring Configuration

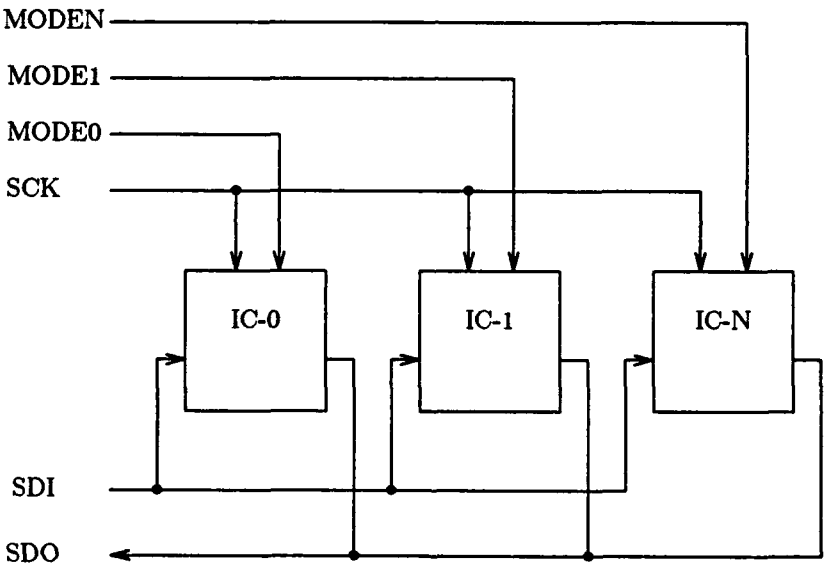


Figure 3-13. JTAG Star Configuration

3.4. Built-in Test (BIT).

3.4.1. Historical Background.

The tremendous increase in complexity in electronic systems over the last decade has been paralleled by a rise in complexity in Automatic Test Equipment (ATE). Even so, testing technology has failed to keep pace with the growth of system technology creating enormous problems for the test community. As a result, the time and costs associated with test generation and application have increased significantly over this period.

As early as 1959, the Third Annual Joint Military-Industrial Electronic Test Equipment Symposium stated that "the need remains for built-in performance monitoring devices in electronic systems. The degree of omission of this class of devices from prime equipments constitutes a striking example of immature design in electronic systems. There is no 'method' of testing more automatic than continuous monitoring; still this technique seems to have been ignored in those types of systems where it could be employed most effectively." This statement was made three years before the invention of the integrated circuit. Thus, in today's world where thousands of logic gates can be placed on a single chip, the need for built-in test (BIT) is greater than ever. In fact, recent studies have concluded that the incorporation of BIT and DFT methods at all design levels has a positive impact on system reliability, maintainability, and supportability.

The high costs of testing can generally be attributed to the costs of test generation and test application. It has been observed that the computational time (and costs) associated with test pattern generation varies approximately as the cube of the number of gates. With today's VLSI/VHSIC technology, these costs can be prohibitive in the development of a system. Likewise, the costs of applying a long serial test sequence through a slower external tester are undesirable.

For these reasons, the concept of BIT has been the subject of much research due to its potential for reducing these costs. For example, random or pseudo-random patterns can be generated on-board the module thereby eliminating the need for generating test vector sets. In addition, the test results are often compressed on-board using data compaction methods that reduce the amount of storage and analysis needed to interpret them. This use of built-in test pattern generators and data compactors can greatly reduce the need for expensive ATE. In most cases, a simple external tester is sufficient to obtain the results of the test sequence.

Built-in test is most often used to reduce the mean time necessary to detect and isolate faults to a specific ambiguity group within a system. In such a manner, it can speed the replacement of faulty components and reduce unscheduled maintenance time. This impacts favorably on system availability as well as maintainability. In the manufacturing phase, built-in test can be used to verify the functionality of a component before its incorporation into a higher level, thus improving overall system reliability.

However, BIT is not without some drawbacks. At all levels of the design hierarchy, the inclusion of BIT modules consumes area and power. As a consequence of this at the

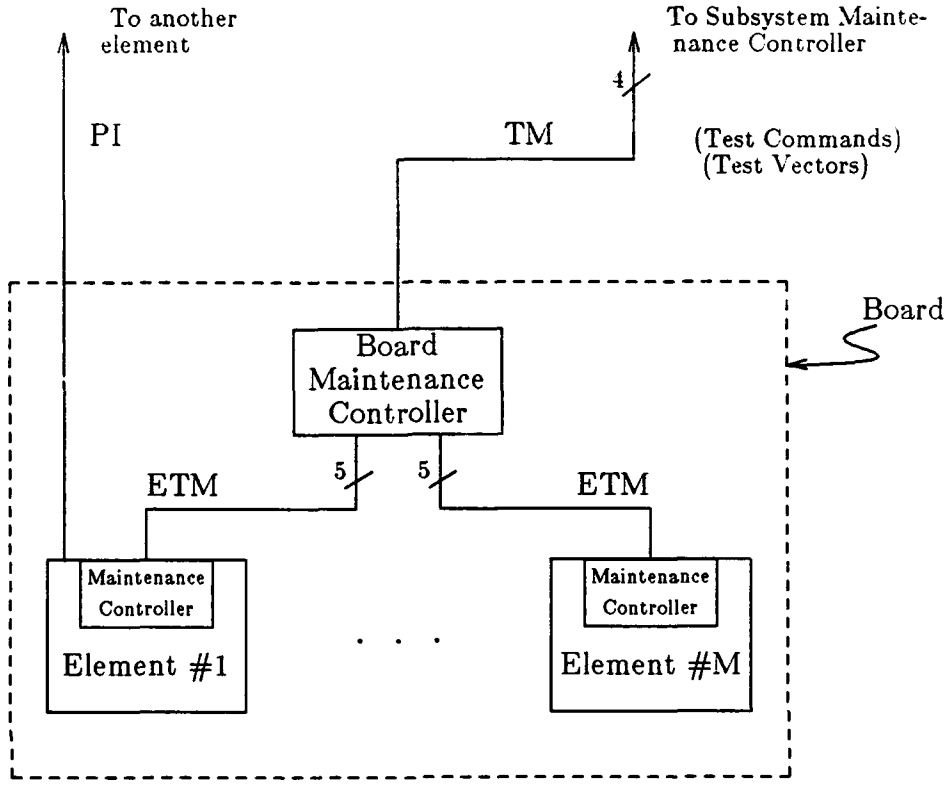
chip level, the expectant yield of IC's will be reduced and the production costs will subsequently increase [7]. Nevertheless, the resultant savings in all phases of testing throughout the system life cycle warrant the inclusion of built-in test as a desirable feature.

In the early days, the performance of BIT circuits in detecting and isolating faults did not meet expectations. Numerous false alarms caused by the BIT circuitry significantly increased the maintenance costs above expected levels during the 1970's. However, recent improvements in testing technology through advanced processing capabilities have improved the rate of false alarms and consequently, the number and costs of field service pulls.

The military services have been especially interested in the capabilities of BIT to reduce the maintenance and support costs for modern weapon systems in the field. At present, the Army uses a three-tier maintenance concept which relies on initial isolation of a fault to occur at the unit level. The faulty module is replaced and shipped to the intermediate level for additional testing and isolation. It is then shipped to the depot where the faulty component(s) are replaced and placed back in the spare parts inventory line. Recent studies undertaken by the Army have concluded that BIT technology has not yet advanced to the point where a two-tier maintenance concept can be supported. To promote the advancement of BIT capabilities, the Army has recently formed a BIT/BITE (Built-In Test Equipment) Center of Excellence within the Army Test, Measurement, and Diagnostic Technology Laboratory. Their goal is to further the built-in test capabilities of systems to provide better isolation capabilities at the unit level, thus eliminating the need for an intermediate maintenance level [8].

Currently, the armed forces are in the midst of the VHSIC program to upgrade and modernize military electronic systems through the incorporation of reliable, *testable*, VLSI components. One of the goals during the initial phase of the VHSIC program was to implement BIT techniques that would assure a high fault coverage for chips, boards, and subsystems. However, independent efforts by the various contractors led to a variety of techniques being implemented. Consequently, it was realized that it would be difficult to construct testable systems due to the lack of a structured scheme to coordinate the test activities [9].

This problem is currently being addressed in Phase II of the VHSIC program through the development of two maintenance busses. In addition to these, the Phase II contractors are developing a PI-Bus that will be used for module to module communications that will allow chips from different chip sets to be interconnected easily without the need for additional logic. Figure 3-14 on the next page graphically depicts the VHSIC bus implementation.



TM = Test and Maintenance Bus
ETM = Element TM Bus
PI = PI Bus (Interoperability Bus)

Figure 3-14. VHSIC TM/ETM/PI Buses

The Test and Maintenance (TM) Bus [10] provides a communication path between the board maintenance controller and a subsystem maintenance controller. The TM Bus functions to transfer test control/data to and from the higher level controller. Similarly, each of the subsystem controllers can be connected via a system maintenance bus to a system level controller which oversees and conducts the testing of the overall system. The Element Test and Maintenance (ETM) Bus [11] accomplishes the same function at a lower level of the design hierarchy. The ETM Bus provides a serial data path to transfer test data and control information between the board level maintenance controller and the maintenance control unit built-in to a VHSIC chip or to a module.

The incorporation of a maintenance bus architecture across different design levels is an integral part of a structured design for testability methodology. It provides a

systematic procedure to conduct and coordinate testing at the various operational levels. Each maintenance controller is responsible for performing tests at its level, control the testing of any underlying levels, receive and process any error messages, and communicate to its appropriate master controller. At the system level, the master controller is the operator. Such a top-down structured approach is important in reducing the time and costs associated with detecting and isolating faults utilizing built-in monitoring units.

Built-in testing may be classified as one of two types: off-line or on-line. Off-line testing makes use of specific test data which is applied to the system during a special test mode of operation. On-line, or concurrent testing, generally makes use of hardware redundancy and coding techniques to test the system during normal operation. Thus, there is no need to generate test data for concurrent testing since it uses the normal operational data. Since redundancy is included in the system hardware, the occurrence of a fault is not necessarily fatal. This method of duplicating hardware in a configuration to improve the reliability of the system is known as fault tolerance. Fault tolerant systems were motivated largely by the desire to have highly reliable space-based systems since they are virtually inaccessible for normal maintenance procedures. Considerable efforts are now underway to further development of fault tolerance methods and to improve their cost effectiveness.

With the increasing usage of complex electronic systems in every facet of our society, it is imperative that their reliability and maintainability be improved. One must now consider the total life cycle costs (LCC) as as important selection criteria when choosing a system for a particular application. It has been observed [12] that the costs incurred with the design and acquisition of a system are less than 50% of the total LCC. Therefore, more than half of the LCC are associated with system logistics costs. The use of BIT and DFT have been shown to be cost effective and instrumental in reducing the development, maintenance, and support costs of modern electronic systems.

The objective of TEA is to develop an automated top-down methodology utilizing built-in test and design for testability to improve system maintenance and supportability. It will be used to aid the designer who is not a testing expert to effectively assess the cost and impact of a particular BIT or DFT technique on his design. As such, it will allow him to meet specific testability requirements at the system, subsystem, and board levels that will result in an overall system that is testable, maintainable, and supportable by construction.

References

[7] P. Varma, A. P. Ambler, and K. Baker, "An Analysis of the Economics of Self Test", Proc. International Test Conference, 1984, pp. 20-30.

[8] D. H. Barclay and M. J. Simpson, "Army ATE Trends and Plans",

AUTOTESTCON 1985. pp. 191-195.

[9] N. Kanopoulos, N. Vasanthavada, J.W. Watterson, and J.J. Hallenbeck, "Advanced CAD/E for Systems Testability", Final Report, Contract DAAB07-85-C-H079, TMDE/ETDL U.S. Army, Ft. Monmouth, NJ, prepared by Research Triangle Institute, Center for Digital Systems Research, June, 1987.

[10]"VHSIC TM-Bus Specification", Version 1.3, prepared jointly by IBM, Honeywell, and TRW, August 31, 1986.

[11]"VHSIC ETM-Bus Specification", Version 1.1, prepared jointly by IBM, Honeywell, and TRW, August 31, 1986.

[12]S. H. Ernst, "A Quantitative Approach to Top Down Testability for Very Large Scale Integration Devices", AUTOTESTCON 1982, pp. 509-514.

3.4.2. TEA Board Level BIT Techniques.

TEA supports one implementation of each of seven board level built-in test techniques. designed to achieve fault detection and/or fault isolation to a particular set of chips determined by user interaction with the BIT Recommendation Tool (Paragraph 2.7.2). Tools in the TEA system are used to estimate the overhead involved in adding a technique to a particular board (Paragraph 2.7.3) and to give instructions for adding TEA's support BIT modules (Paragraph 3.4.3) and/or test points to that board to get an updated (testable) representation. VHDL descriptions of the BIT modules are provided to support functional simulation of the system with added BIT features. These modules are represented by ADAS templates.

The techniques are provided to complement the TEA testing methodology shown in Figure 2-4. BIT support modules (or test points) are added to a board which communicate with an on-board maintenance node, which communicates to the subsystem test strategy.

The techniques implemented as part of TEA are intended for application to board designs of digital synchronous circuitry. The techniques are applied in a strict manner of *adding* logic (or test points) to accomplish the fault isolation requirement. The tools do not make recommendations for *replacing* logic nor do they tell the user how to perform tests; however, the BIT support module application notes show examples of accepted methods. Typically these techniques facilitate fault isolation to an ambiguity group plus the BIT support modules added for that group.

A maintenance node is recommended for each board; however, the tools do not make recommendations about connecting the maintenance node to the on-board BIT circuitry because this depends a great deal on how the user decided to test his system. The Maintenance Node application note shows examples of different testing styles.

In some cases there are many possible implementations of a general BIT technique: only one has been selected for each. The chosen implementation may result in very high overhead (e.g., test modules, additional control lines) that makes the technique appear unacceptably costly, but this technique should not be ruled out entirely. In some cases a combination of techniques may result in the best overall testability and the lowest cost. TEA does not consider combinations of its techniques in itemized cost lists, but they could be used when getting an updated schematic. Appendix B shows diagrams of other possible implementation of the TEA techniques. These are included for the informational use only; they have not been implemented as part of TEA. The user could use these diagrams for manual addition of a particular technique to a board since the modules needed are in the BIT module data base and the ADAS templates are provided.

Non-optimal use of BIT support modules may also contribute to high cost estimates. TEA has a support library of modules expecting a 16-bit data path. If the data path is 18 bits, for example, the TEA tools recommend "ganging" two full-sized modules together, underutilizing at least one of the modules. The user could possibly determine two bits of that path that would not require observation, thus saving the

overhead associated with that module.

A single example is used throughout the rest of this paragraph to illustrate the TEA techniques. Figure 3-15 shows a simple system consisting of three undetermined size (number of chips) ambiguity groups (AG) which are highly interconnected. Each AG is connected to both of the others. Each AG accepts input from the board edge and delivers output to the board edge. This example does not show any special AGs (Paragraph 2.6.1), but any of the AGs could be considered special. Simple one-line interconnections are shown for the purposes of illustration.

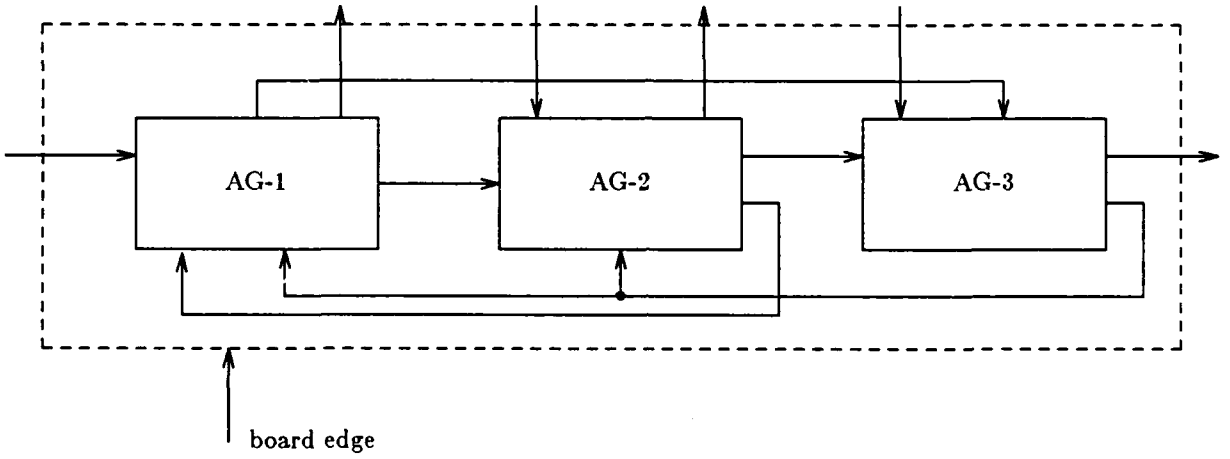
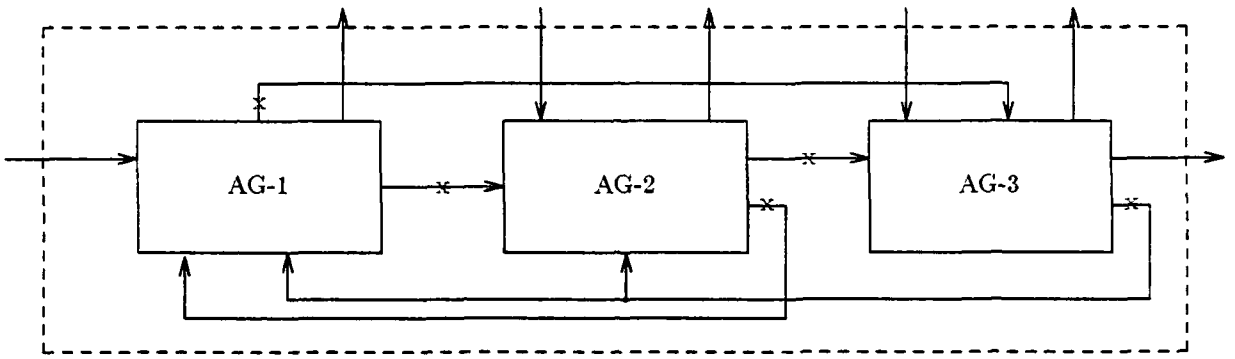


Figure 3-15. Board with No BIT Technique Applied.

This paragraph describes each of the techniques. Instructions for running the tools are found in the *TEA User Manual.*

Continuous Test Point Monitoring is abbreviated **det_tp** in the tool menus. This is an off-line testing technique. Figure 3-16 shows how this technique is applied to the example board shown in Figure 3-15. **det_tp** adds no BIT support modules besides the maintenance node. Test points are added at ambiguity group boundaries, except at board outputs. TEA optimizes the number of test points for high testability and low overhead and gives exact locations.

Careful selection of a test program set can eliminate the need for test points, but procedures for doing so are not supported in TEA since test programs are not required to run TEA.



x indicates test point output

Figure 3-16. BIT Technique 1-a: Test Point Monitoring — Continuous (Cycle by Cycle).

This technique does not do any test pattern generation, so it can support both deterministic and pseudorandom testing.

Test Point Monitoring with Data Compression is abbreviated **tp_sa** in the tool menus. This is an off-line testing technique. Figure 3-17 shows how this technique is applied to the example board shown in Figure 3-15. **tp_sa** adds

a. pseudorandom test pattern generators (TPG) (exclusive-or implementation) for intercepting incoming data lines

b. scan-set registers for intercepting incoming control lines

c. built-in logic block observers (BILBOs) to compress output lines into signatures

d. multiplexors as appropriate so that no more TPGs and BILBOs than necessary are added to the design

Clock lines are not trapped and brought through BIT modules.

The overhead for this technique could be quite high. Each TPG has an associated multiplexor. There is at least one TPG. Multiple TPGs are needed if the data path is $>$ 16 bits. Highly connected AGs further penalize the overhead because enough multiplexors are needed to break feedback paths. Scan-set modules are needed for each AG that has control inputs. These modules do not need multiplexors since the library model has normal and "test" modes. Each set of lines that need to be compressed go to a multiplexor port and each of these output multiplexors has a BILBO associated with it. Data lines are kept separate from control lines going to these output multiplexors. AGs may have multiple data and control sets because the technique keeps

lines going to a particular AG separate from lines going to a different AG and these are all kept separate from lines leaving the board. If less than total isolation can be tolerated, it is recommended that these restrictions be relaxed.

This technique supports pseudorandom test pattern generation on the data inputs and unrestricted control of the control lines.

Board Level Boundary Scan is abbreviated **boundary** in the tool menus. This is an off-line testing technique. Figure 3-18 shows how this technique is applied to the example board shown in Figure 3-15. **boundary** adds

a. scan-set registers for intercepting incoming control lines

b. scan-set registers for intercepting incoming data lines and outgoing board signals

This technique has relatively low overhead associated with it, but fault isolation to the AG is no longer guaranteed unless the user has a carefully selected test program set and the order of application of those test sets is carefully controlled. This technique keeps data and control lines separate. All board outputs are routed back to the scan-

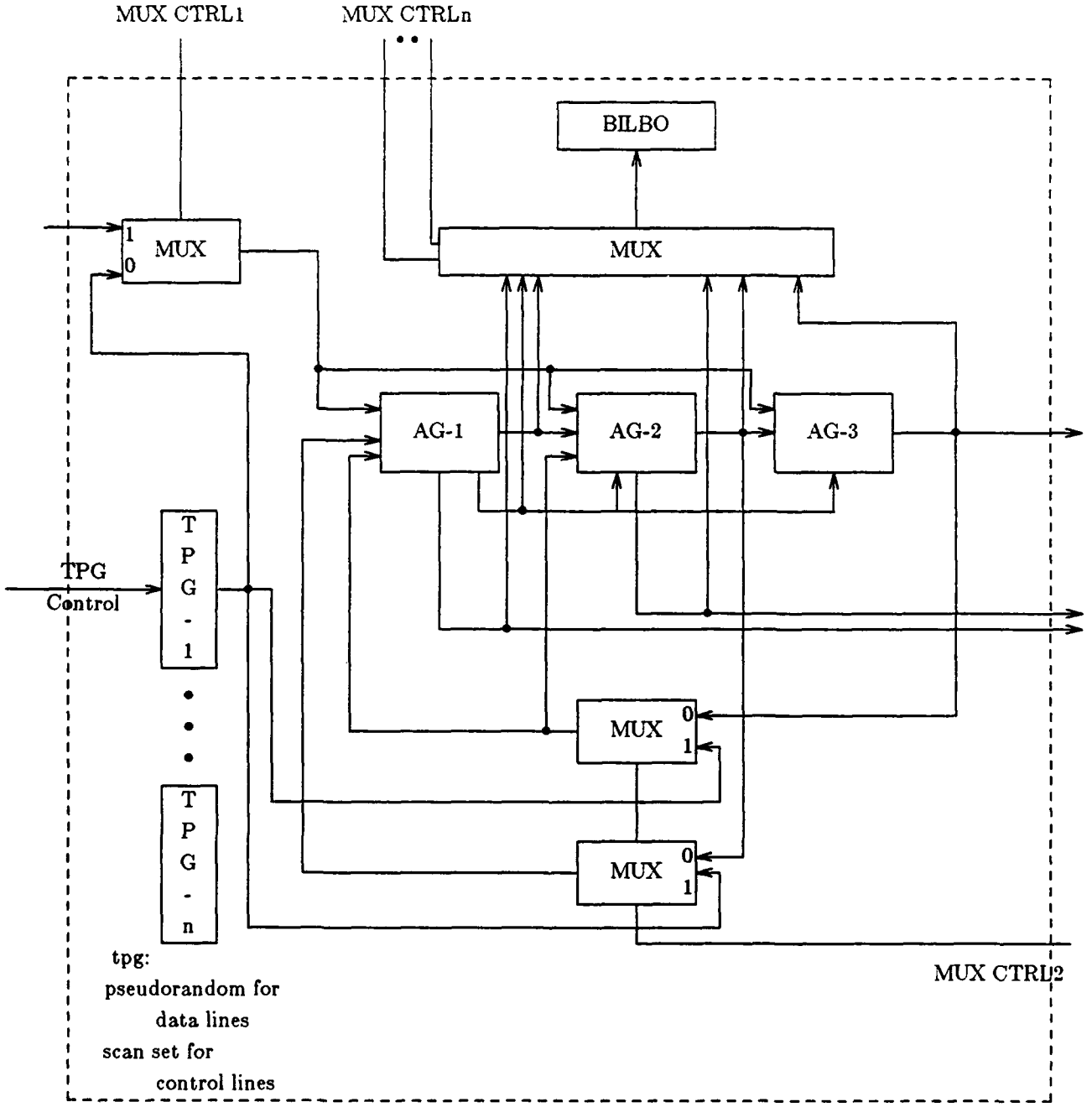


Figure 3-17. BIT Technique 2-a: Test Point Monitoring — Data Compression

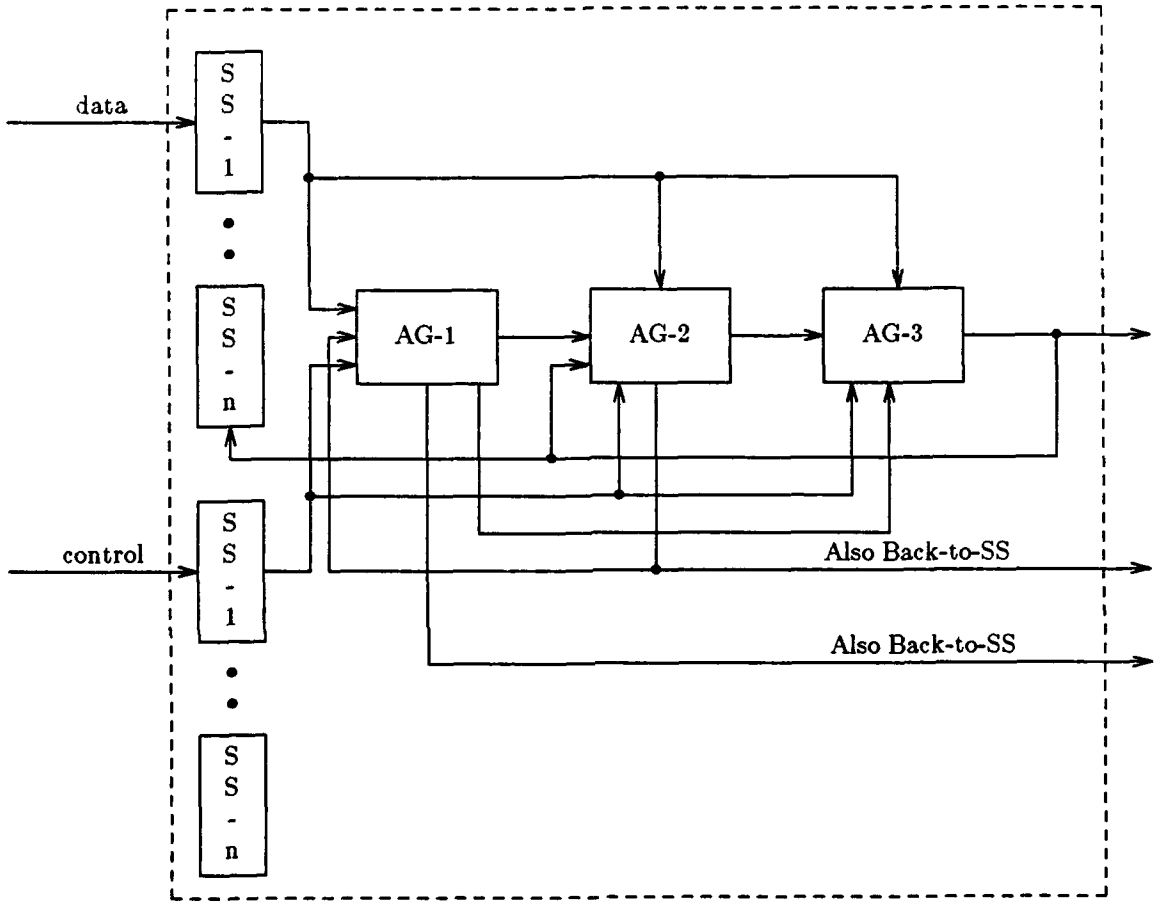


Figure 3-18. BIT Technique 3-a: Board-Level Boundary Scan

set registers used to intercept incoming data lines so there is overhead associated with high number of board outputs compared to data input lines.

This technique does not generate test patterns for the circuit, so predetermined test patterns would either have to be stored for the board or generated off-board and applied serially through the scan-set registers.

This technique supports deterministic testing.

Scan-Set is abbreviated **scan_set** in the tool menus. This is an off-line testing technique. Figure 3-19 shows how this technique is applied to the example board shown in Figure 3-15. **scan_set** adds

a. scan-set registers for intercepting incoming control lines

b. scan-set registers for intercepting incoming data lines

c. scan-set registers for intercepting internal data lines which feed another AG

d. scan-set registers for intercepting internal control lines which feed another AG

All lines go from a scan-set module to the AG both in normal and "test" modes. No line will (except clocks) enter an AG if it has not been fed through a scan-set module. All lines leaving an AG are fed back to the scan-set module which provided the input to the AG so there is some overhead associated with providing for the larger of those counts for each AG. All feedback and fanout paths are broken to further aid fault isolation to an AG and accompanying scan-set modules. All AG outputs which leave the board are also fed back to the AG input registers. In all cases, data and control lines are kept separate.

Careful control of the test program set is needed as the user will want to serially test each AG while the other AGs are in normal mode.

This technique does not generate test patterns for the circuit, so predetermined test patterns would either have to be stored for the board or generated off-board and applied serially through the scan-set registers.

Test Pattern Generation with Data Compression is abbreviated **gen_sa** in the tool menus. This is an off-line testing technique. Figure 3-20 shows how this technique is applied to the example board shown in Figure 3-15. **gen_sa** can add

a. testing switches (TSWITCH) for intercepting incoming control lines

b. TSWITCHes for intercepting incoming data lines

c. TSWITCHes for intercepting internal data lines which feed another AG

d. TSWITCHes for intercepting internal control lines which feed another AG

e. TSWITCHes for intercepting data lines leaving the board

f. TSWITCHes for intercepting control lines leaving the board

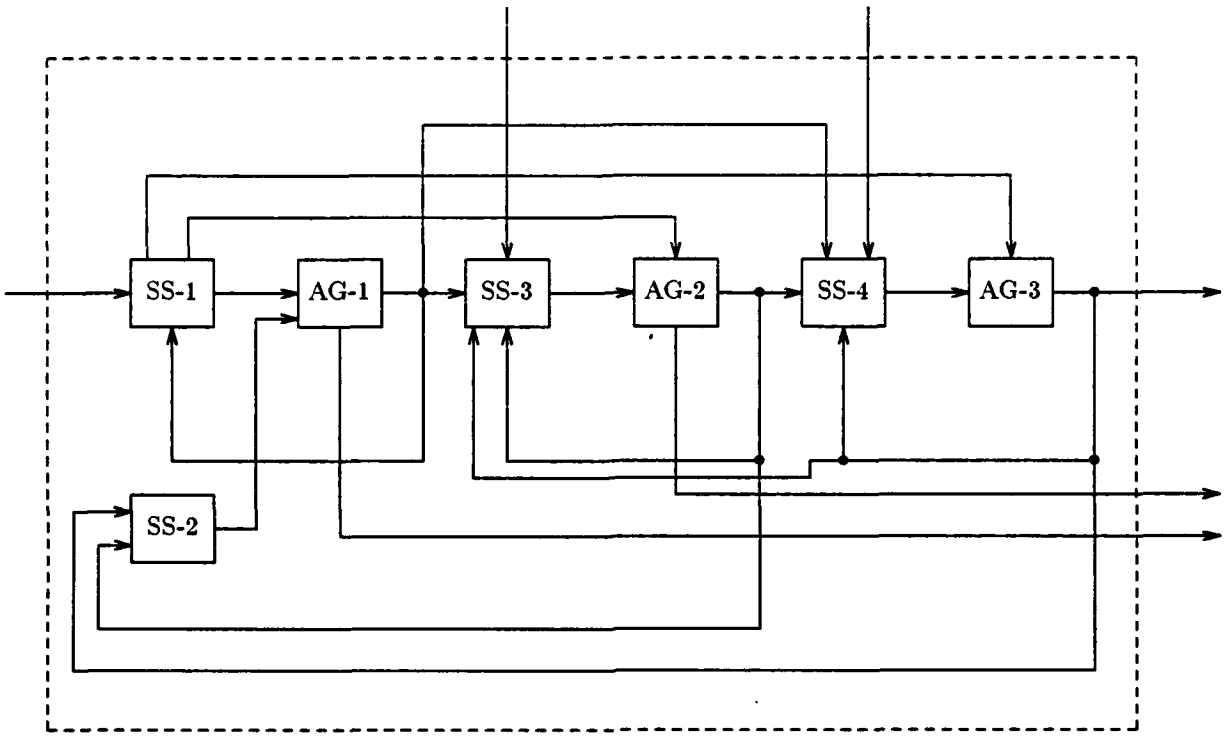


Figure 3-19. Bit Technique 4-a: Scan-set Technique Using Scan-set BIT Modules Distributed over the Board.

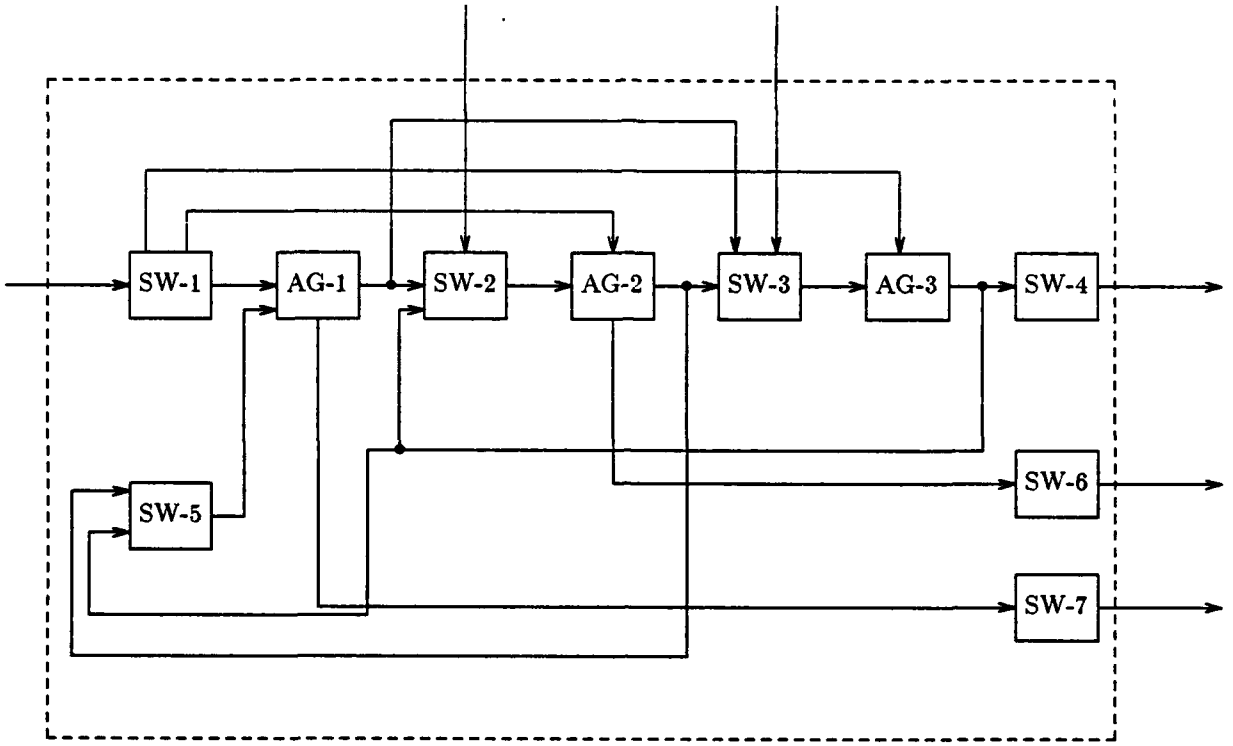


Figure 3-20. Bit Technique 5-a: Test Pattern Generation and Response Compression Using "Testing Switch" Modules Distributed over the Board.

The TSWITCHes which precede an AG generates the input vectors for the AG. The TSWITCHes which follow an AG analyzes the output vectors from the AG. All lines go from a TSWITCH module to the AG both in normal and "test" modes. No line (except clocks) will enter an AG if it has not been first fed through a TSWITCH module. All lines leaving an AG are fed forward to the TSWITCH module which provides the input to the next AG(s). All feedback and fanout paths are broken to further aid fault isolation to an AG and accompanying TSWITCH modules and this is done with just one TSWITCH delay. All AG outputs which leave the board are also fed to a TSWITCH module. In all cases, data and control lines are kept separate.

Careful control of the test program set is needed as the user will want to test each AG while other AGs are in normal mode. A single TSWITCH is capable of simultaneously generating patterns for one AG while compressing patterns received from another.

This technique can support both pseudorandom and deterministic test pattern generation on both the data and control lines.

Parity Check/Generate is abbreviated **parity** in the tool menus. This is an on-line, or concurrent testing technique, meaning that testing can be accomplished during operation of the system. Figure 3-21 shows how this technique is applied to the example board shown in Figure 3-15.

This technique is supported differently from the five techniques described above. No estimation of overhead costs is performed for this technique because from strict topological information available to TEA, TEA does not make decisions about where modules are needed to generate or check the parity of a device. The user is directed to texts describing coding theory and practical application of parity.

TEA allows the straight-forward addition of the "parity" library module to a schematic by requesting the user to only specify the lines to which the module should be attached. TEA will automatically reroute the arcs and add the nodes to the graph. The user will then have to make the graph "pretty" again.

Test patterns are functional input patterns, as this is an on-line technique.

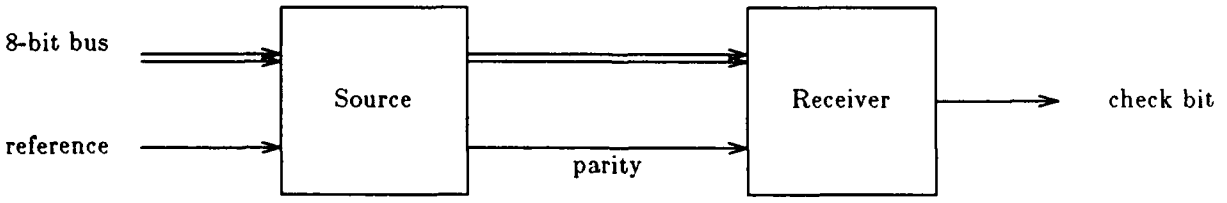


Figure 3-21. BIT Technique 6: Parity Generation/Checking.

On-line Comparison of Duplicated Modules is abbreviated **compare** in the tool menus, except on the top-level TEA menu, where **compare** refers to the System Summary tool (Paragraph 2.7.5). This is an on-line, or concurrent testing technique, meaning that testing can be accomplished during operation of the system. Figure 3-22 shows how this technique is applied to the example board shown in Figure 3-15.

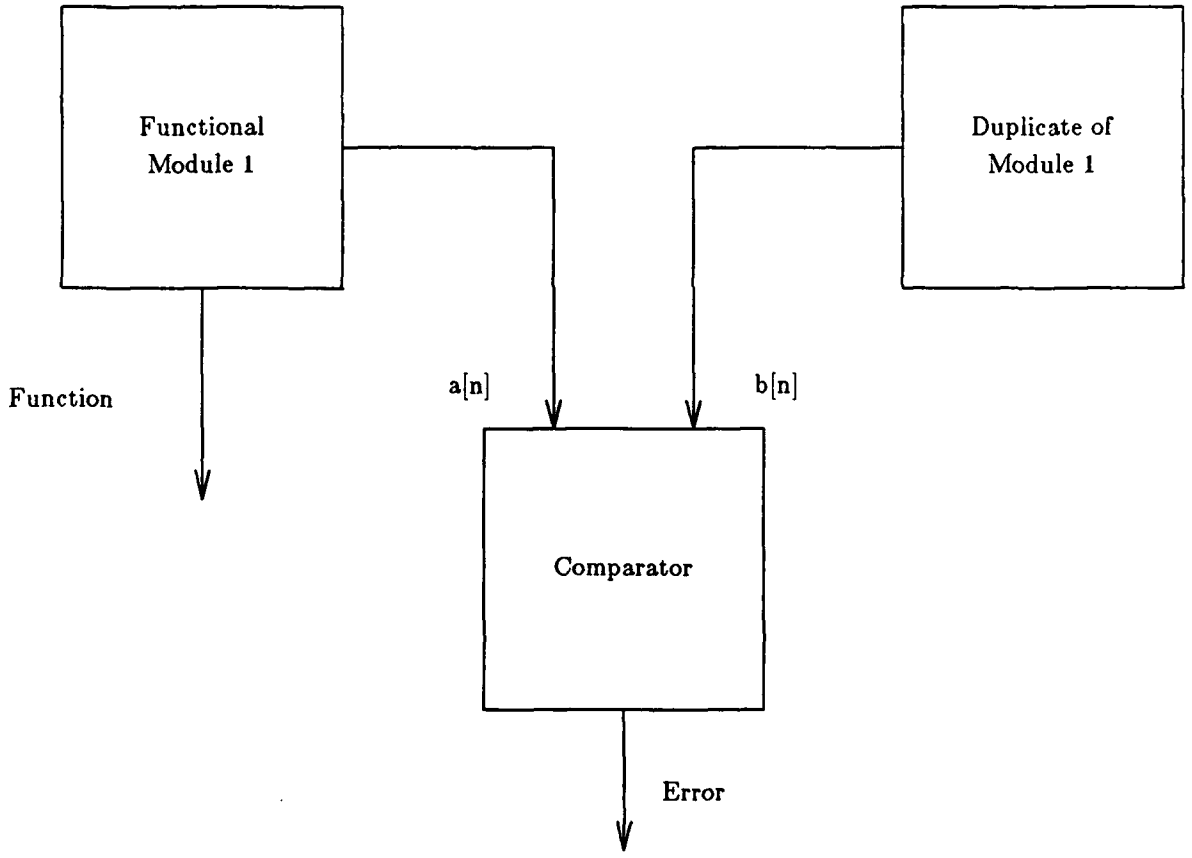


Figure 3-22. BIT Technique 7: Selective Duplication.

This technique's support is identical to the technique identified as **parity**. No estimation of overhead costs is performed for this technique because from strict topological information available to TEA, TEA does not make decisions about where duplication of modules is needed to provide fault detection (and perhaps other fault tolerance) capabilities. The user is directed to texts describing the use of module duplication as a form of on-line fault detection. Duplication and comparison will not aid in fault isolation because if an error is indicated at the output of the comparator, the user still doesn't know if the error occurred in the "functional" module or the "duplicated" module.

TEA allows the straight-forward addition of the "compare" library module to a schematic by requesting the user to only specify the lines to which the module should be attached. TEA will automatically reroute the arcs and add the nodes to the graph. The user will then have to make the graph "pretty" again.

Test patterns are functional input patterns, as this is an on-line technique.

3.4.3. TEA BIT Support Modules. Appendix C describes the details of each of the BIT support modules provided with TEA in the form of application notes. The group of modules forms a library or data base. With this library, the user receives

a. an application note describing the function of the module along with an example implementation and notes on using the module in a testing situation

b. an ADAS template to use when adding nodes representing BIT support modules into a graph

c. VHDL, version 7.2, and IEEE 1076 structural and/or behavioral descriptions of the module

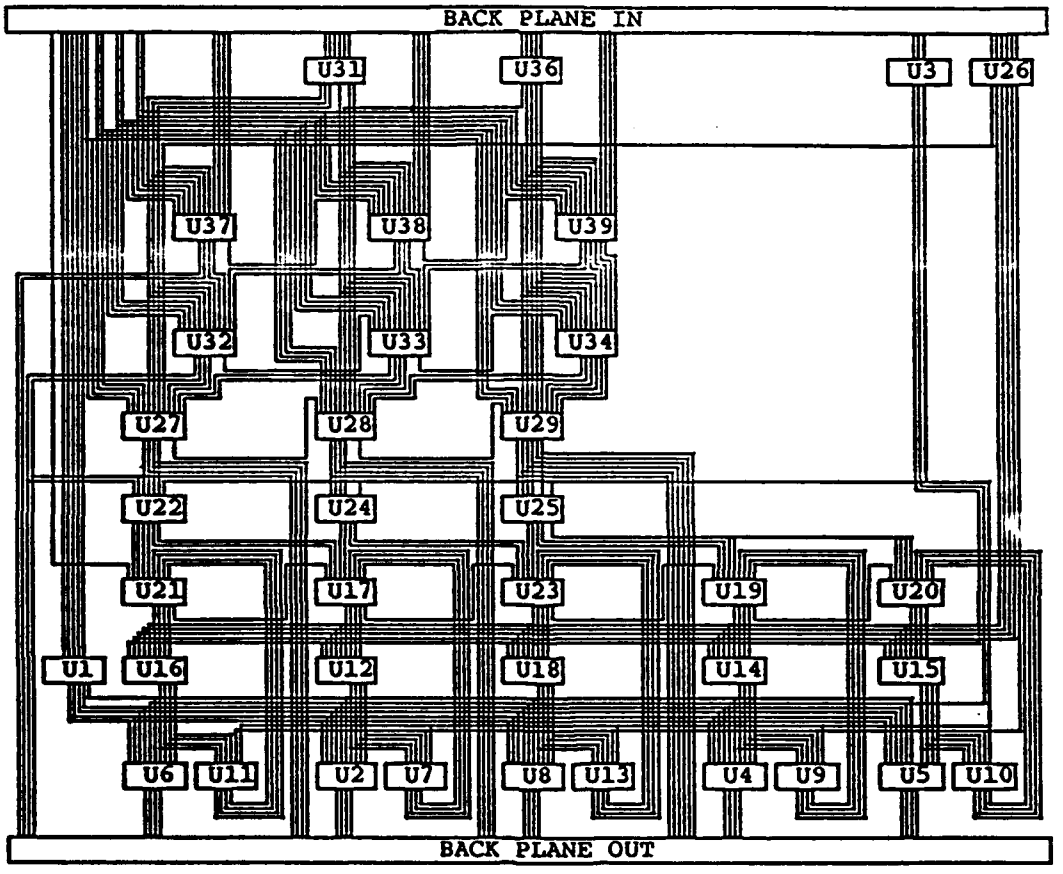
d. a gate level description of the model entered in CADAT (product of HHB-Systems of Mahwah, NJ) format

The BIT Overhead Summary (Paragraph 2.7.3) and BIT Placement Recommendation (Paragraph 2.7.4) tools use the library to pick out modules to support the BIT technique chosen by the user.

In many cases, a second version of each BIT module (referred to as version 2) is supplied. This second version has been built to interface to the maintenance node delivered, without additional functionality. The version 2 modules will not automatically be added to a system by the TEA tools.

4. EXAMPLE USE OF TEA

The following is an example of the use of the TEA tools. Figure 4-1 is the ADAS representation of the system under consideration.



prod_20.hwg

Figure 4-1. ADAS Graph of Doppler Filter Card of the Radar Signal Processor of Firefinder

Output from the tools is in regular type. User responses are shown in bold typeface. Commentary will be shown in italics.

TERMINAL SESSION

$ tea prod_20.hwg -d vsii

The user has entered the graph name (prod_20.hwg) and graphics display on the command line. This refers to the DEC VAXstation II.

```
*******************************************************
*                                                     *
*  ADAS:  Architecture Design and Assessment System   *
*  Copyright 1985, 1986, 1987 by  Research Triangle   *
*  Institute.                                         *
*                                                     *
*       Tea (version PROTO.TYPE)                      *
*                                                     *
*     Proprietary Software. All rights reserved.      *
*     No part of this software may be sold            *
*     or distributed in any form or by any means      *
*     without the prior written permission of         *
*     the Research Triangle Institute.                *
*                                                     *
*******************************************************
```

```
Getting database ...
```

The user's hardware graph file is loaded into the data base.

```
=== menu ===
quit            brt             log_file        hardcopy
save            bit_cost        window          script
edit            placebit        environ         macro
move            compare         stats           reload
dft             ag_name         subgraf         help
============
command% dft
```

This is the TEA top-level menu. The user has selected the Design for Testability Guideline Checker from the menu.

```
=== menu ===
  %help
  analyze
  identify
  explain
============
DFT: Select a DFT option to be executed: analyze
```

The first menu the user encounters is the dft action menu. Here the user has selected to analyze his graph for untestable or hard-to-test structures.

```
=== menu ===
  %help
  standard
  user_rules
============
DFT: Select the type of rules to use: standard
```

The user wants to select from the guidelines provided with TEA.

```
=== menu ===
%done           g1_02           g1_08           g1_14
%help           g1_03           g1_09           g1_15
%all            g1_04           g1_10           g1_16
%all_g1         g1_05           g1_11           g2_01
%all_g2         g1_06           g1_12
g1_01           g1_07           g1_13
============
Enter '*' to view additional menu items
DFT: Select the guideline(s) to be executed: g1_02 %done
```

The only guideline selected is g1_02: Make preset/clear a primary input or primary input controllable. Note that the user must select the %done option to exit this menu.

```
=== menu ===
  %done
  %help
  %all
  radar
============
DFT: Indicate the subsystem(s) of interest: radar %done
```

```
=== menu ===
  %done
  %help
  %all
  doppler
============
DFT: Indicate the board(s) of interest: doppler %done
```

The doppler board of the radar subsystem has been selected for analysis.

```
**** Running prolog code ****
 [compiling project:[tea.jrc.demo]graph.pl...]
 [graph.pl compiled 334.350 sec 77,072 bytes]
 [compiling project:[tea.jrc.demo]control.pl...]
 [control.pl compiled 0.980 sec 496 bytes]
Run analyze rules
running g1_02

*** g1_02 VIOLATION --> r1 U22 port=7 clear
*** g1_02 VIOLATION --> r1 U24 port=7 clear
*** g1_02 VIOLATION --> r1 U25 port=7 clear
```

Exiting the board menu starts the execution of dft. Dft is implemented in prolog, so the graph data base must be translated into prolog for comparison to the guidelines. graph.pl is the prolog representation of the graph data base. control.pl is the prolog representation of the input the user has given to dft (e.g., board name, guidelines).

```
Enter <cr> or YES to accept dft_01.rpt as the default filename
Or enter desired output filename : g1_02
```

The user has selected to name his output report g1_02 rather than accept the default name selected by the tool.

```
=== menu ===
quit              brt             log_file        hardcopy
save              bit_cost        window          script
edit              placebit        environ         macro
move              compare         stats           reload
dft               ag_name         subgraf         help
============
command% log_file
```

Once the tool completes, process control is returned to the top-level menu. The user now selects the report access utility.

```
=== menu ===
  %help
  view
  print
============
Log_file: View or print report? view


Enter directory path to search.
<cr> or YES searches current dir: yes


=== menu ===
  %help
  g1_02.rpt
============
Log_file: Select a report: g1_02.rpt
```

The user has selected to view the file called g1_02.rpt in his current directory. This is the output of the mail to dft that was just completed.

```
------------------------------------------------------------
Test Engineer's Assistant (TEA) Output Report

ADAS/TEA Version: PROTO.TYPE
Research Triangle Institute
P.O. Box 12194
Research Triangle Park, NC 27709

Date/Time:  Wed Sep 21 06:16:41 1988

User: JRC

Graph:
      Current View Graph:           Doppler_Filter_Board
      Current View Filename:        PROJECT:[TEA.JRC.DEMO]PROD_20.HWG;1

Output Report Filename = PROJECT:[TEA.JRC.DEMO]G1_02.RPT;1
------------------------------------------------------------
DESIGN FOR TESTABILITY GUIDELINE CHECKER
(dft analyze)


Where:
      Subsystem-
            radar

      Board-
            doppler

      Guidelines Selected-
            g1_02


G1_02

G1_02_REC - To aid test pattern generation, the
PRESET or CLEAR input of flip_flops, counters and
shift_registers should be directly controllable.
Port 'clear' (port=7) of node 'U22'
is a PRESET/CLEAR line that is not primary input
controllable. Make this port (port=7) primary input
controllable.
```

```
G1_02_REC - To aid test pattern generation, the
PRESET or CLEAR input of flip_flops, counters and
shift_registers should be directly controllable.
Port 'clear' (port=7) of node 'U24'
is a PRESET/CLEAR line that is not primary input
controllable. Make this port (port=7) primary input
controllable.


G1_02_REC - To aid test pattern generation, the
PRESET or CLEAR input of flip_flops, counters and
shift_registers should be directly controllable.
Port 'clear' (port=7) of node 'U25'
is a PRESET/CLEAR line that is not primary input
controllable. Make this port (port=7) primary input
controllable.


Flip_flops, Counters, and Shift_registers

U10  U11  U12  U13  U14  U15  U16  U18  U2
U22  U24  U25  U4   U5   U6   U7   U8   U9

There were 3 violations found

dft analyze Completed
-------------------------------------------------------
```

The output shows all the registers, flip flops, counters found in the graph. The node_names of the nodes are shown as a list. In addition to this list, the violations and recommendations are shown.

```
=== menu ===
quit              brt              log_file         hardcopy
save              bit_cost         window           script
edit              placebit         environ          macro
move              compare          stats            reload
dft               ag_name          subgraf          help
============
command% dft


=== menu ===
  %help
  analyze
  identify
  explain
============
DFT: Select a DFT option to be executed: identify
```

The user has entered dft again, this time selecting the identify function.

```
=== menu ===
  %done
  %help
  %all
  radar
============
DFT: Indicate the subsystem(s) of interest: radar %done


=== menu ===
  %done
  %help
  %all
  doppler
============
DFT: Indicate the board(s) of interest: doppler %done


=== menu ===
%help              fanout>5          node              register
attribute          flipflop          one_shot          scan_reg
bit_mod            loop              primaries         test_point
counter            memory            processor
============
DFT: Select an item to be identified: fanout>5
```

The user has instructed the tool to look for occurrances of fanout > 5 on the doppler board of the radar subsystem.

```
Enter <cr> or YES to accept dft_01.rpt as the default filename
Or enter desired output filename : doppler_fanout
```

The results of this action will be itemized in doppler_fanout.rpt.

```
=== menu ==
quit               brt               log_file          hardcopy
save               bit_cost          window            script
edit               placebit          environ           macro
move               compare           stats             reload
dft                ag_name           subgraf           help
============
command% dft


=== menu ===
  %help
  analyze
  identify
  explain
============
DFT: Select a DFT option to be executed: explain
```

```
=== menu ===
  %help
  g1
  g2
============
DFT: Select the appropriate guideline group: g1

=== menu ===
%help              g1_05              g1_10              g1_15
g1_01              g1_06              g1_11              g1_16
g1_02              g1_07              g1_12
g1_03              g1_08              g1_13
g1_04              g1_09              g1_14
============
DFT: Select the specific guideline to be explained: g1_02
```

The help file information concerning g1_02 will be accessed through use of the explain function of dft.

```
=== menu ===
  %help
  yes
  no
============
DFT: Store the explanation to a file? no
```

The results are displayed on the screen.

```
TOOLS

  DFT_Checker

    Rules

      G1_02


        G1_02 Rule: Make Preset/Clear a  primary  input  or  primary
        input controllable.

        Why?
             Because this provides direct control of the initializa-
        tion  function  to  the tester and can be used to initialize
        several circuits at once with a single pulse during test.

        How?
             (a) Avoid preset and clear being tied together  through
        a resistor to VDD.  Although the tester can access the point
        (a primary input) to initialize this component, the  outcome
        will  be  indeterminate  when  the  tester releases the test
        point.

             (b) Do one of these.
```

(i) Make the clear input a primary input or primary input controllable. The tester can access this input directly or through other controlling circuitry.

(ii) Make the clear line an arbitrary signal gated with a primary input which is connected with VDD through a resistor. The tester accesses the primary input and clears the circuit with override.

(iii) Make the clear line an arbitrary signal gated with a primary input which is connected with VDD through a resistor and VSS through a capacitor. The circuit clears with tester override and also on power-up.

Syntax

inport_id: preset, pre, clear, clr
Tarc_type: data, control, cntrl, ctrl, clock, clk

Tdft_io: primary_input, pi, primary_output, po, test_point, test_point_output, test_pt_output, tp, tpo

Tlogic_type: combinational, comb_logic, comb

Tnode_type: flip_flop, flipflop, ff, counter, ctr, shift_register, sr, shiftregister

Topic? **<CR>**

The user types a carriage return to exit the help files.

```
=== menu ===
quit              brt              log_file         hardcopy
save              bit_cost         window           script
edit              placebit         environ          macro
move              compare          stats            reload
dft               ag_name          subgraf          help
============
command% brt
```

Now the user has selected to enter the BIT Recommendation Tool.

```
=== menu ===
  %help
  radar
============
BRT: Indicate the subsystem of interest: radar
```

```
=== menu ===
  %help
  doppler
============
BRT: Indicate the board of interest: doppler
BRT: Enter desired AG size: 3

=== menu ===
  %help
  yes
  no
============
BRT:  Alter cost function to account for testability? yes
```

The possible groups of size 3 and less of the doppler board will be considered for their testability. A modified cost function will be used to find a minimum cost solution. A testing recommendation will be assigned to each node and to each ambiguity group.

```
Enter <cr> or YES to accept brt_01.rpt as the default filename
Or enter desired output filename : brt_doppler_size3

Finding ambiguity groups . . .

Finding optimal configuration of ags . . .
```

The results are stored in an output file.

```
=== menu ===
quit              brt               log_file          hardcopy
save              bit_cost          window            script
edit              placebit          environ           macro
move              compare           stats             reload
dft               ag_name           subgraf           help
============
command% bit_cost

=== menu ===
  %help
  radar
============
Bit_cost: Indicate the subsystem of interest: radar

=== menu ===
  %help
  doppler
============
Bit_cost: Indicate the board of interest: doppler
```

The costs of implementing BIT on the doppler board with ambiguity groups as assigned by brt will be estimated by the BIT Overhead Summary tool.

```
=== menu ===
%done              %all               tp_sa              scan_set
%help              det_tp             boundary           gen_sa
============
Bit_cost: Select BIT technique(s): tp_sa %done


Enter <cr> or YES to accept bitcost_01.rpt as the default filename
Or enter desired output filename : bitcost_tp_sa


Enter <cr> or YES to accept bitcost_01.dwg as the default filename
Or enter desired output filename : bitcost_tp_sa
```

The results of running bit_cost on the doppler with the Test Point Monitoring with Data Compression technique will be written to the files called bitcost_tp_sa.rpt and bitcost_tp_sa.dwg.

```
=== menu ===
quit               brt                log_file           hardcopy
save               bit_cost           window             script
edit               placebit           environ            macro
move               compare            stats              reload
dft                ag_name            subgraf            help
============
command% placebit


=== menu ===
%help              tp_sa              scan_set           parity
det_tp             boundary           gen_sa             compare
============
Placebit: Select a BIT technique: tp_sa
```

Confident that tp_sa is the BIT technique of choice, the user has entered the BIT Placement Recommendation tool and selected tp_sa.

```
=== menu ===
  %help
  yes
  no
============
Placebit: Do you want TEA to attempt an example implementation? yes
```

This indicates that the user wants more than general instructions about the technique he has chosen.

```
=== menu ===
  %help
  yes
  no
============
Placebit: Add nodes to supply constant 0/1 values? yes
```

When the graph is updated, all input data lines of the BIT modules will have an input value attached.

```
=== menu ===
  %help
  radar
============
Placebit: Indicate the subsystem of interest: radar


=== menu ===
  %help
  doppler
============
Placebit: Indicate the board of interest: doppler


== menu ===
  %done
  %help
  %all
  autoplace
  wirelist
============
Placebit: Place and/or supply manual wiring list? wirelist %done
```

The user has not chosen to have his graph automatically updated, rather he has selected to get a wirelist so that manual implementation of the technique could be achieved.

```
Enter <cr> or YES to accept placebit_01.rpt as the default filename
Or enter desired output filename : placebit_doppler_tp_sa
```

General wiring information and error reports are given in the .rpt output file.

```
Enter <cr> or YES to accept placebit_01.dwg as the default filename
Or enter desired output filename : placebit_doppler_tp_sa
```

Specific wiring information is given in the .dwg output file.

```
Some changes to the graphs were recommended.
```

The tool is indicating that the wiring list is not empty.

```
=== menu ===
quit              brt               log_file          hardcopy
save              bit_cost          window            script
edit              placebit          environ           macro
move              compare           stats             reload
dft               ag_name           subgraf           help
============
command% compare
```

The user wants some statistics concerning his graph(s), so he has entered the System Summary tool.

```
Enter directory path to search.
<cr> or YES searches current dir: yes


=== menu ===
%help               system_2.hwg
%self               system_3.hwg
system_1.hwg        system_4.hwg
============
Compare: Select the comparison graph: %self
```

The list of hardware graph files in the current directory is presented. The user has chosen to only get output concerning his "current view" graph and about no others.

```
Enter <cr> or YES to accept compare_01.rpt as the default filename
Or enter desired output filename : compare_self
```

A chart of statistics about the graph is shown in the .rpt output file. This includes number of BIT modules, number of test points, and mean time to test.

```
Enter <cr> or YES to accept compare_01.dwg as the default filename
Or enter desired output filename : compare_self
```

A breakdown of the ambiguity groups and board inputs/outputs are listed in the .dwg output file.

```
Some errors were detected in the graphs,
check the report for details.
```

The tool indicates that some error messages will appear in the .rpt file.

```
=== menu ===
quit                brt                 log_file            hardcopy
save                bit_cost            window              script
edit                placebit            environ             macro
move                compare             stats               reload
dft                 ag_name             subgraf             help
============
command% quit nosave
```

Exiting the tool is done by selecting quit from the top-level menu. No changes made to the graph (There should have been none anyway, in this case.) during the session will be saved since the user has selected to nosave his data base.

5. NOTES

5.1. Glossary.

aid to test pattern application. Some design for testability guidelines are in the data base to aid test pattern application. These guidelines generally ease the controllability of clocks and significant control functions of a system so that automatic test equipment can be used to externally control the system during test mode.

aid to test pattern generation. Some design for testability guidelines are in the data base to aid test pattern generation. These guidelines generally increase controllability and observability of nodes buried within a system.

ambiguity group (AG). the smallest group of items to which a fault can be isolated

analog node. a node that has not been identified as digital. An analog node has its **Tdevice_type** set to a, analog. A node with no **Tdevice_type** value is digital.

built-in test (BIT). a particular design for testability technique that requires that some technique for detecting faults be provided in addition to the function

built-in test techniques. TEA has a data base of seven particular BIT techniques. These include

a. continuous test point monitoring, can support both deterministically and pseudorandomly generated test patterns
b. test point monitoring with data compression, supports deterministic test patterns for control lines and pseudorandomly generated test patterns for data lines
c. board level boundary scan, supports deterministic test pattern application to both control and data lines
d. scan-set, supports deterministic test pattern application to both control and data lines
e. test pattern generation with signature analysis, supports pseudorandomly generated control and data line inputs
f. parity generation/checking
g. on-line comparison of duplicated modules

The first five are off-line techniques, implying that the system is not operational at the time of testing and the test patterns supported are noted. The last two are on-line, or concurrent to operation, techniques which require very special consideration.

controllability. the ability to control values on nets. The controllability of a line is the probability that a randomly applied input vector will set the line to a given value.

controllable loop. a closed circuit of controllable nodes. Controllable nodes have inputs which are primary inputs or primary input controllable. Controllable loops cannot contain entirely combinational logic.

design for testability (DFT). One of the approaches used to reduce testing costs is to recognize testing difficulties when designing the system, subsystem, board, or chip and incorporate certain features to facilitate fault detection and fault isolation into the design. Design styles that result in testable designs are collectively grouped under the name "design for testability." Testability is desirable because it reduces the cost of testing and it improves the quality of the test.

fanout node.

a. a node is a fanout node if its attribute **hw_module** is set to one of fanout, copy, null
b. a node is a fanout node if its attribute **Tnode_type** is set to one of fanout, copy, null

Fanout nodes are not recommended. ADAS buses should be used to show connections from one source to multiple destinations. In general, TEA will give unpredictable results if fanout nodes are used.

large node. identified by **Thw_module** (number of gates) and **Tstates** (number of states).

$$\text{Size} = (\textbf{gates})\ 2^{\textbf{states}}$$

If Size $> 2^{16}$=65536, a node is large.

mean time to test. sum of the mean times given to test the ambiguity groups

observability. the ability to observe values on nets. The observability of a line is the probability that a randomly applied input vector will sensitize one or more paths from that line to a primary output.

output observable (OO).

A node is Primary Output Observable if all of its outports are primary output observable. The following is the order in which outport X is checked to see if it is Primary Output Observable:

a. The outport is checked for successors. If there are no successors, outport X is not OO. If one or more successors exists, it keeps on checking.

b. The outgoing arc is checked to see if the **Tdft_io** is set to PO, PI or TPO. If so, outport X is OO. If not, it keeps on checking.

c. The sink node is checked to see if it belongs to another board. If so, outport X is OO. If not, it keeps on checking.

d. The sink node is checked to see if it has outports. If the sink node has no outports, outport X is OO. If the sink node does have outports, it keeps on checking.

e. The sink node is checked to see if it has been checked before. If it has, this is a loop and outport X is not OO. If it has not been checked before, it goes back to step a, using the sink node for outport X.

f. The outgoing arc (that goes out from outport X) is checked to see if the **Tarc_type** is set to clock. If it is not set to clock, (i.e., data or control) it keeps on checking. If **Tarc_type** = clock, outport X is not OO.

g. If one of the following is true, then it goes to step f. Otherwise, outport X is not OO. The **Tlogic_type** for the sink node is set to combinational

   1. The **Tnode_type** for the sink node is set to scan latch
   2. The **Tnode_type** for the sink node is set to bus (or the sink is a bus)
   3. The **Tnode_type** for the sink node is set to fanout
   4. The **Tnode_type** for the sink node is set to flip flop
   5. The **Tnode_type** for the sink node is set to counter
   6. The **Tnode_type** for the sink node is set to shift register

h. A node must be PIC in order to be OO.

primary input (PI) or primary output (PO).

a. an arc with **Tdft_io** labeled as PI/PO

b. an arc which crosses from one board to another

primary input controllable (PIC). A node is Primary Input Controllable if all of its inports are PIC. The following is the order in which node X is checked to see if it is Primary Input Controllable:

a. A node port is NOT PIC if the inport has no sources.

b. The incoming arc is checked to see if the **Tdft_io** is set to PO, PI or TPO. If so, inport X is PIC. If not, it keeps on checking.

c. The source node is checked to see if it comes from another board. If so, inport X is PIC. If not, it keeps on checking.

d. The source node is checked to see if it has inports. If it has no inports, inport X is PIC. If the source node does have inports, it keeps on checking.

e. The source node is checked to see if it has been checked before. If it has, this is a loop and inport X is not PIC. If it has not been checked before, it goes back to step a, using the source node for inport X.

f. The incoming arc (that comes into inport X) is checked to see if the **Tarc_type** is set to clock. If it is not set to clock, (ie, data or control) it keeps on checking. If **Tarc_type** = clock, inport X is not PIC.

g. If one of the following is true, then it goes to step f. Otherwise, inport X is not PIC.

   1. The **Tlogic_type** for the source node is set to combinational
   2. The **Tnode_type** for the source node is set to scan latch
   3. The **Tnode_type** for the source node is set to bus (or the source is a bus)
   4. The **Tnode_type** for the source node is set to fan out

sequential node. a node that has not been identified as combinational. A combinational node has its **Tlogic_type** set to comb, comb_logic, combinational. A node with no **Tlogic_type** value is sequential.

test patterns. the set of input stimuli used to verify the operation of a system. These patterns can be generated deterministically or pseudorandomly. Some systems are better tested with vectors generated one way over the other. Typically, only patterns applied to data inputs are generated pseudorandomly.

test point (TPO). an arc with **Tdft_io** labeled as test_point, test_point_output, test_pt_output, tp, tpo

testability. a design characteristic which allows the unit's status (operable, inoperable, or degraded) and the fault locations within the unit to be confidently determined in a timely fashion

5.2. Index.

5.3. List of Acronyms.

Software Design Document
Acronyms for TEA System

ACO	Administrative Contracting Officer
ACST	Advanced CAD/E ior Systems Testability
ADAS	Architecture Design and Assessment System
ADS	Automated Data System
ADSD	Automated Data System Document
AG	Ambiguity Group
AJD	Analysis and Justification Document
AVI	ADAS/VHDL Interface
ATE	Automatic Test Equipment
BIT	Built-in Test
CA	Configuration Audit
CAD	Computer-aided Design
CDSR	Center for Digital Systems Research
CO	Contracting Officer (or Commanding Officer)
COM	Computer Operation Manual
COTR	Contracting Officer's Technical Representative
CDR	Critical Design Review
CDRL	Contract Data Requirements List
CDSR	Center for Digital System Research
CFSR	Contract Funds Status Report
CIDS	Critical Item Development Specification
CM	Configuration Management
CMP	Configuration Management Plan
CPFF	Cost Plus Fixed Fee
CRISD	Computer Resources Integrated Support Manual
CSC	Computer Software Component
CSCI	Computer Software Configuration Item
CSDM	Computer Software Diagnostic Manual
CSOM	Computer System Operator's Manual
CWBS	Contract Work Breakdown Structure
DAC	Days After Contract Signed
DAL/ID	Data Accession List/Internal Data
DAO	Days After Option
DAR	Defense Acquisition Regulations
DBS	Data Base Specification
DFT	Design for Testability
DID	Data Item Descriptions

DoD	Department of Defense
DoDISS	Department of Defense Index of Specifications and Standards
DRD	Data Requirements Document
EDR	Evaluation and Demonstration Report
ETM	Element Test and Maintenance
FCA	Functional Configuration Audit
FD	Functional Description
FER	Funds Expenditure Report
FQR	Formal Qualification Review
GFS	Government-Furnished Software
HWCI	Hardware Configuration Item
INC	Incrementally Funded Contract
IV & V	Independent Verification & Validation
JLC	Joint Logistics Commanders
JTAG	Joint Test Action Group
LLCSC	Low Level Computer Software Component
LRU	Line Replaceable Unit
MCCS	Mission Critical Computer System
MN	Maintenance Node
OM	Operator's Manual
PCA	Physical Configuration Audit
PCO	Procuring Contracting Officer
PDR	Preliminary Design Review
PI	VHSIC Interoperability
PIDS	Prime Item Development Specification
PMM	Program Maintenance Manual
PP/APC	Project Planning/Actual Project Chart
PS	Program Specification
PSR	Project Status Report
RTI	Research Triangle Institute
SDD	Software Design Document
SDDD	Software Detailed Design Document
SDF	Software Development File
SDL	Software Development Library
SDP	Software Development Plan
SDR	System Design Review
SOM	System Operation Manual
SOW	Statement Of Work
SQEP	Software Quality Evaluation Plan
SRR	System Requirements Review
SSSRD	System Specification and Software Requirements Document
SSA	Software Support Agency
SSPM	Software Standards and Procedures Manuals
SSR	Software Specification Review

SSS	System Subsystem Specification
STD	Software Test Description
STP	Software Test Plan
STPR	Software Test Procedure
STR	Software Test Report
SSUM	Software System User's Manual
SWCI	Software Configuration Item
TAR	Test Analysis Report
TEA	Test Engineer's Assistant
TEAS	Test Engineer's Assistant System
TCO	Termination Contracting Officer
TCU	Test Control Unit
TIP	Technology Insertion Plan
TISSS	Tester Independent Support Software System
TLCSC	Top Level Computer Software Component
TM	Test and Maintenance
TP	Test Plan
TRR	Test Readiness Review
UDF	Unit Development Folder
UM	User's Manual
USAF	United States Air Force
USN	United States Navy
VDD	Version Description Document
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VSC	VHSIC Silicon Compiler
WBS	Work Breakdown Structure
ag_name	TEA Ambiguity Group Names
bit_cost	TEA BIT Overhead Summary
brt	TEA BIT Recommendation
compare	TEA System Summary
dft	TEA Design for Testability Guideline Checker
help	TEA On-Line User Support System
log_file	TEA Report Generation/Access
placebit	TEA BIT Placement Recommendation

Appendix A

Quick DFT Guideline Reference

G1-01 Guideline: Use flip-flops, counters, and shift registers with a Preset/Clear capability.

G1-02 Guideline: Make Preset/Clear a primary input or primary input controllable.

G1-03 Guideline: Make Dselect of multiplexors a primary input or primary input controllable.

G1-04 Guideline: Make Tristate Control a primary input or primary input controllable.

G1-05 Guideline: Make Enable/Hold of microprocessors a primary input or primary input controllable.

G1-06 Guideline: Make Chip Select (CS), Address Latch Enable (ALE), Read, and Write of memories primary inputs or primary input controllable.

G1-07 Guideline: Make Control Access of any bus structure a primary input or primary input controllable.

G1-08 Guideline: Make buried (i.e., not primary input or primary input controllable) control lines of the types Preset/Clear; Dselect of multiplexors; Tristate Control; Enable/Hold of microprocessors; CS, ALE, Read, and Write of memories; and Control Access of bus structures primary outputs.

G1-09 Guideline: Make the output of all logic redundant nodes a primary output.

G1-10 Guideline: Make the trunk section of high fanout nodes or junctions a primary output.

G1-11 Guideline: Terminate all unused device inputs and tristatable outputs.

G1-12 Guideline: Keep analog and digital circuits physically apart.

G1-13 Guideline: Make analog lines used as input to digital data acquisition circuits a primary output.

G1-14 Guideline: Avoid asynchronous logic.

G1-15 Guideline: Avoid uncontrollable feedback.

G1-16 Guideline: Avoid one-shots as delay elements.

G2-01 Guideline: Make all on-board clocks a primary input or primary input controllable.

G2-02 Guideline: Avoid fanout greater than 5.

G2-03 Guideline: Break up long counter chains greater than 8 bits with logic that contains primary inputs or primary input controllable lines.

G2-04 Guideline: Avoid mixed logic families on the same board unless all I/O is compatible in logic levels and rise and fall times.

G2-05 Guideline: Limit chip fanout at test points to one less than the specified maximum.

G2-06 Guideline: Provide the refresh circuitry for DRAMs on the same board as the RAM.

Appendix B

Alternative BIT Technique Implementations Currently not Supported by TEA

TEA supports one implementation of each of seven board level built-in test techniques. designed to achieve fault detection and/or fault isolation to a particular set of chips. This appendix is designed to show a few alternative implementations of some of these techniques. These alternatives ARE NOT currently supported by TEA.

A single example is used throughout this Appendix to illustrate the TEA techniques. Figure 3-15 shows a simple system consisting of three undetermined size (number of chips) ambiguity groups (AG) which are highly interconnected. Each AG is connected to both of the others. Each AG accepts input from the board edge and delivers output to the board edge. This example does not show any special AGs. but any of the AGs could be considered special. Simple one-line interconnections are shown for the purposes of illustration.

Continuous Test Point Monitoring

Figures B-1 through B-5 show alternatives of this technique as applied to the example board shown in Figure 3-15.

The implementation shown in Figure B-1 allows the outputs of test points to be read into and out of a scan-set register so that the results can be serialized for analysis.

Figure B-2 shows an implementation that allows for self generation of input stimulus. Test points are monitored externally.

A combination of the previous two options are shown in Figure B-3. Input stimulus is generated on board and test point output is latched.

An alternative to the last option allows for latching both the input stimulus and output responses. A maintenance node is shown as a reminder that to interface to a subsystem test strategy all inputs and outputs should be handled through a maintenance node. This is shown in Figure B-4.

This continuous test point monitoring alternative implementation allows for both pseudorandom stimulus for data lines and deterministic stimulus for control lines and also for latching of the test point output data. Figure B-5 shows this most complete option.

Test Point Monitoring with Data Compression

Figure B-6 shows an alternative of this technique as applied to the example board shown in Figure 3-15. Fanout as well as feedback paths are broken with appropriate test pattern generation capability so that faults are better isolated to ambiguity groups.

Test Pattern Generation with Data Compression

Figures B-7 through B-10 show alternatives of this technique as applied to the example board shown in Figure 3-15.

The implementation shown in Figure B-7 uses built-in logic block observers (BILBOs) instead of testing-switches (TSWITCHes) to provide the input and compress the output.

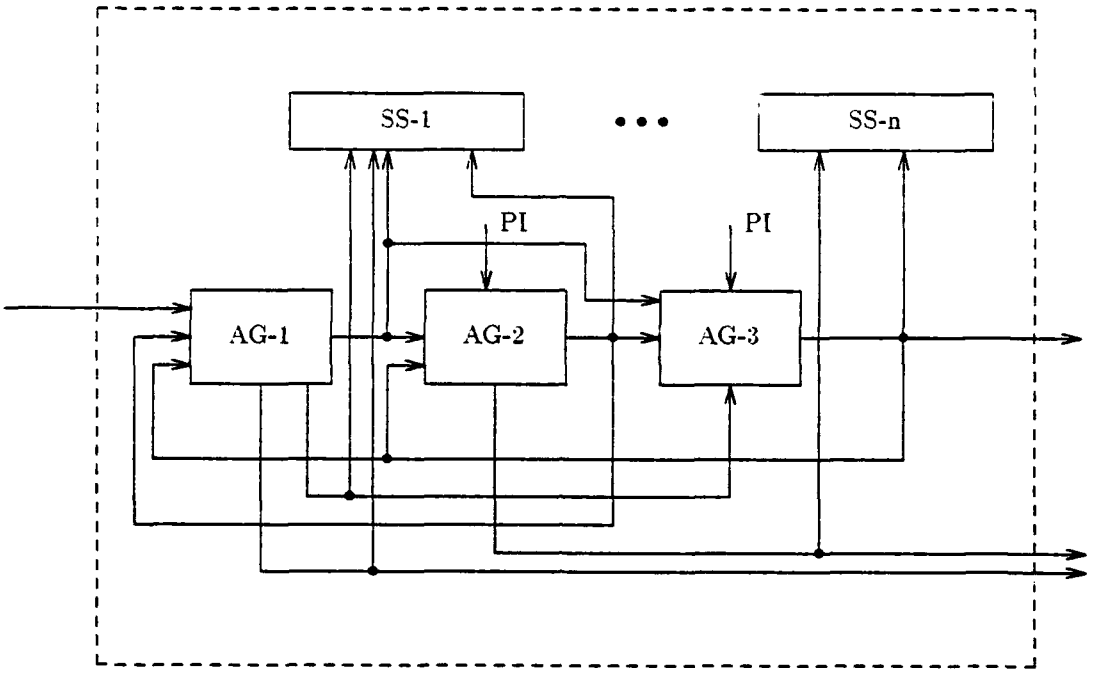


Figure B-1. BIT technique 1-b: Test point monitoring - continuous (cycle by cycle)

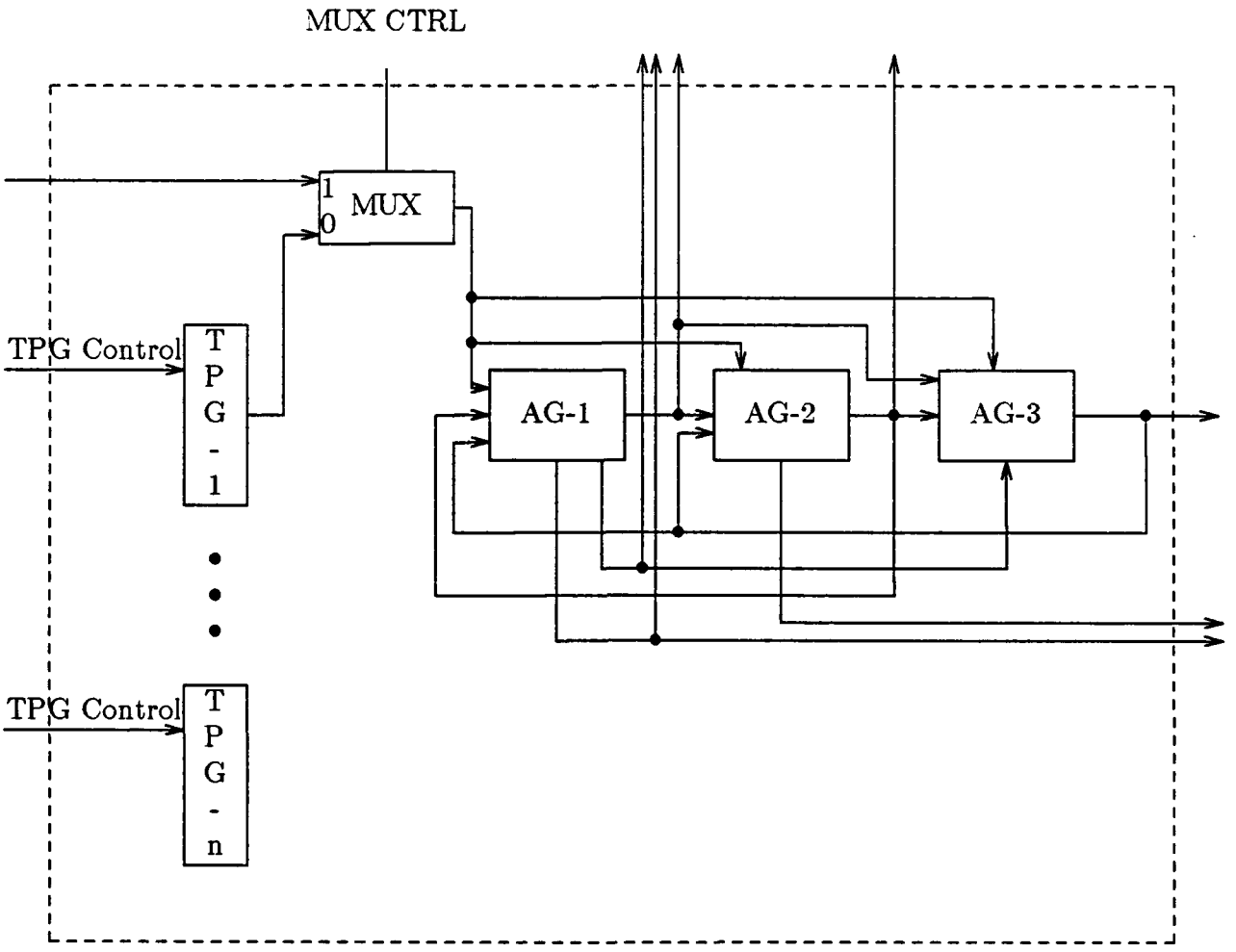


Figure B-2. BIT technique 1-c: Test point monitoring - continuous (cycle by cycle)

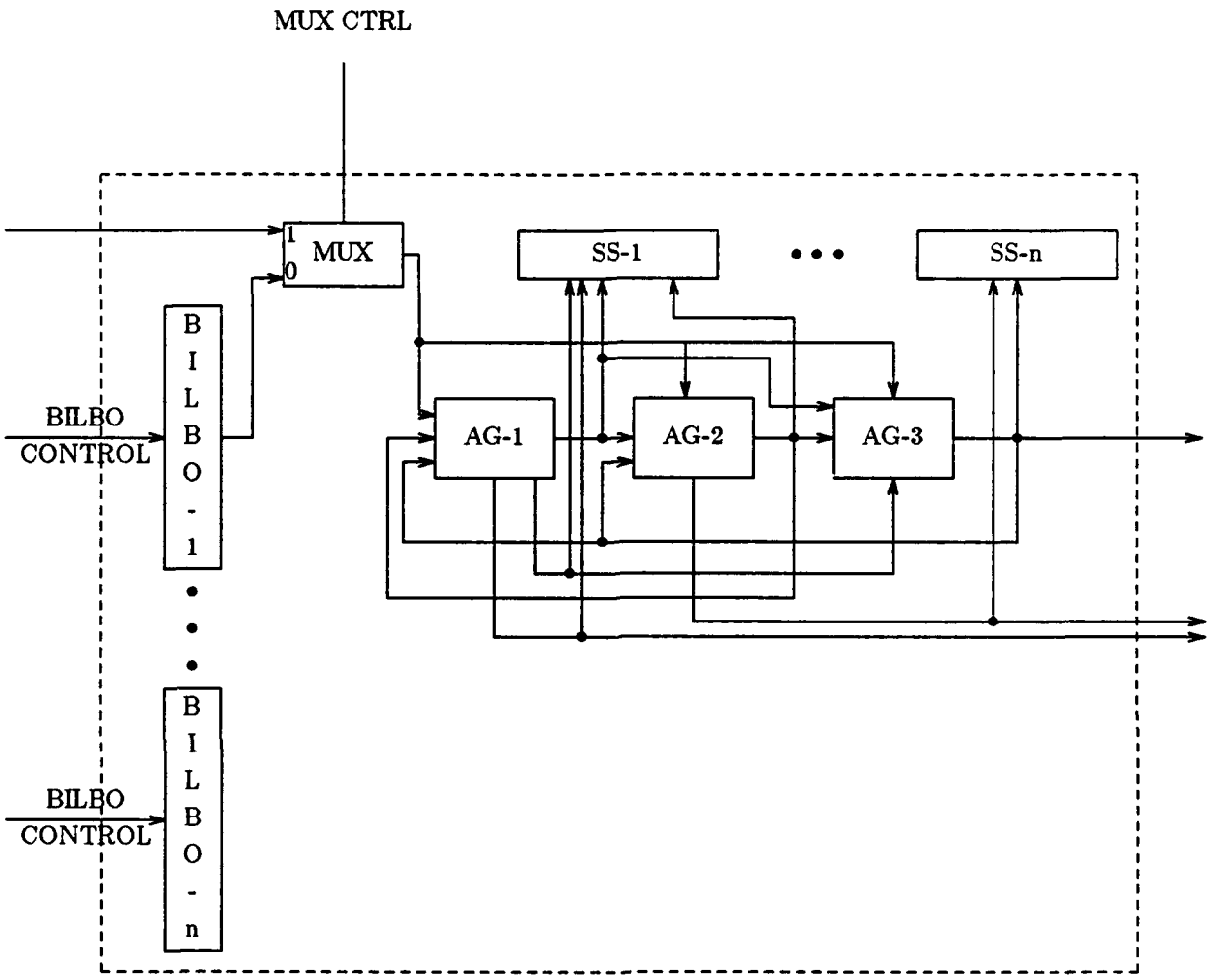


Figure B-3. BIT technique 1-d: Test point monitoring - continuous (cycle by cycle)

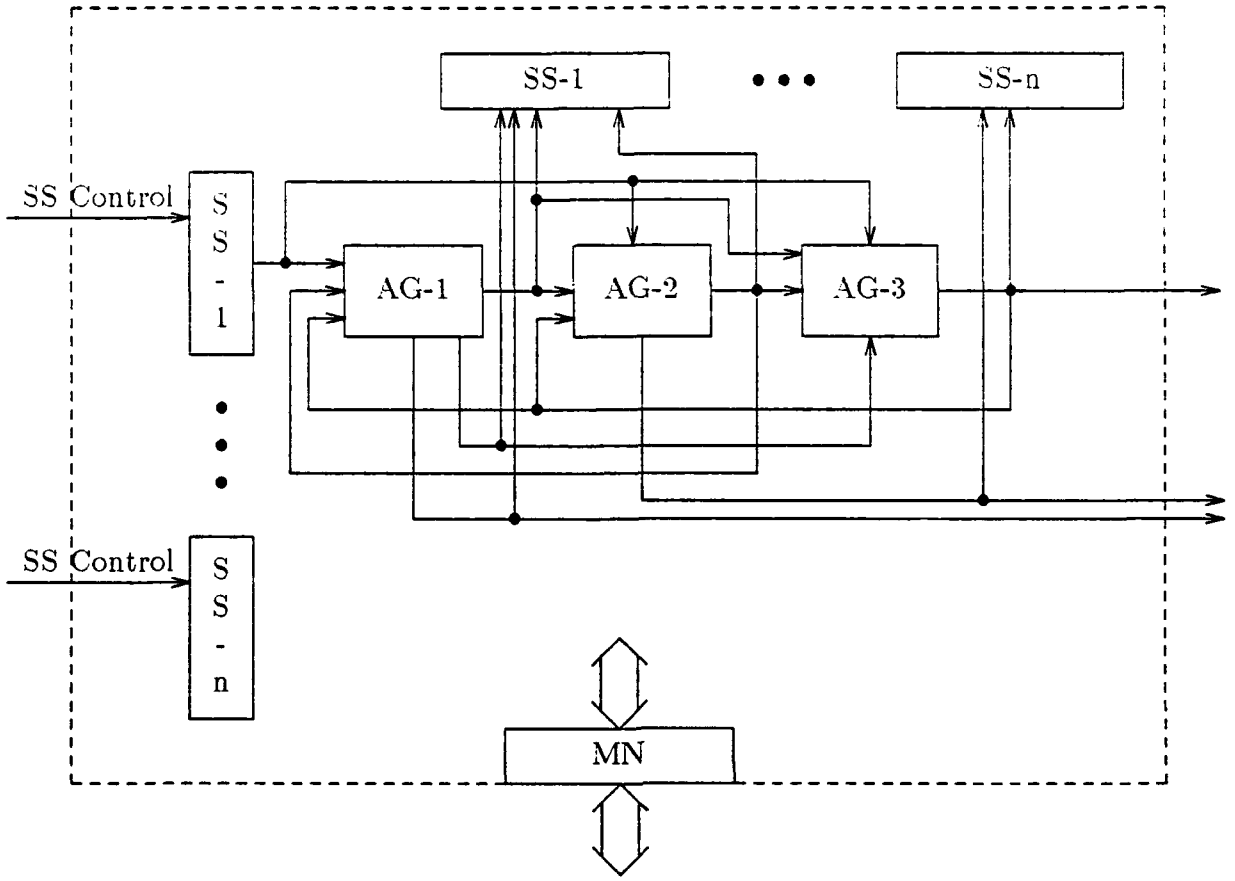


Figure B-4. BIT technique 1-e: Test point monitoring - continuous (cycle by cycle)

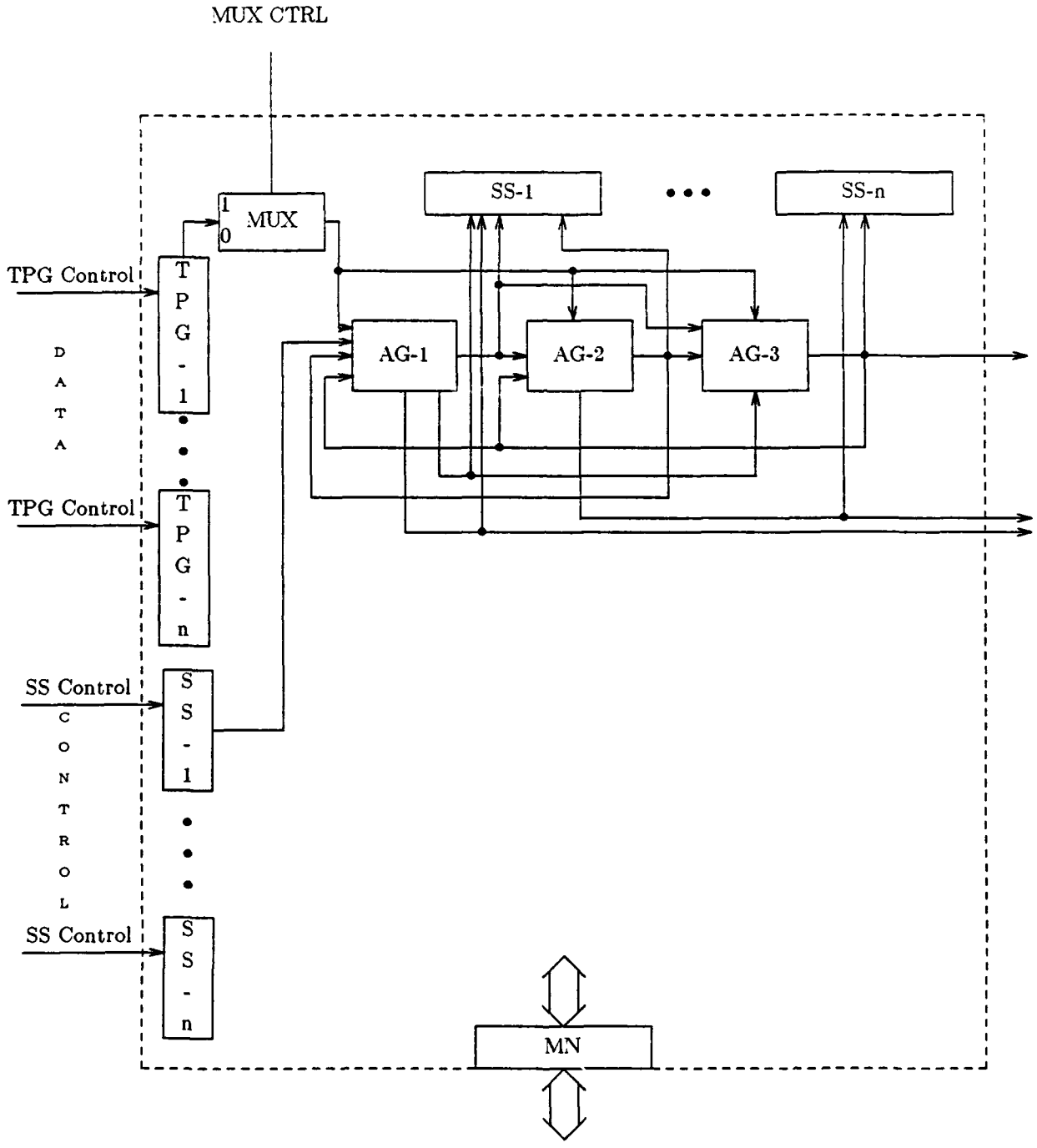


Figure B-5. BIT technique 1-f: Test point monitoring - continuous (cycle by cycle)

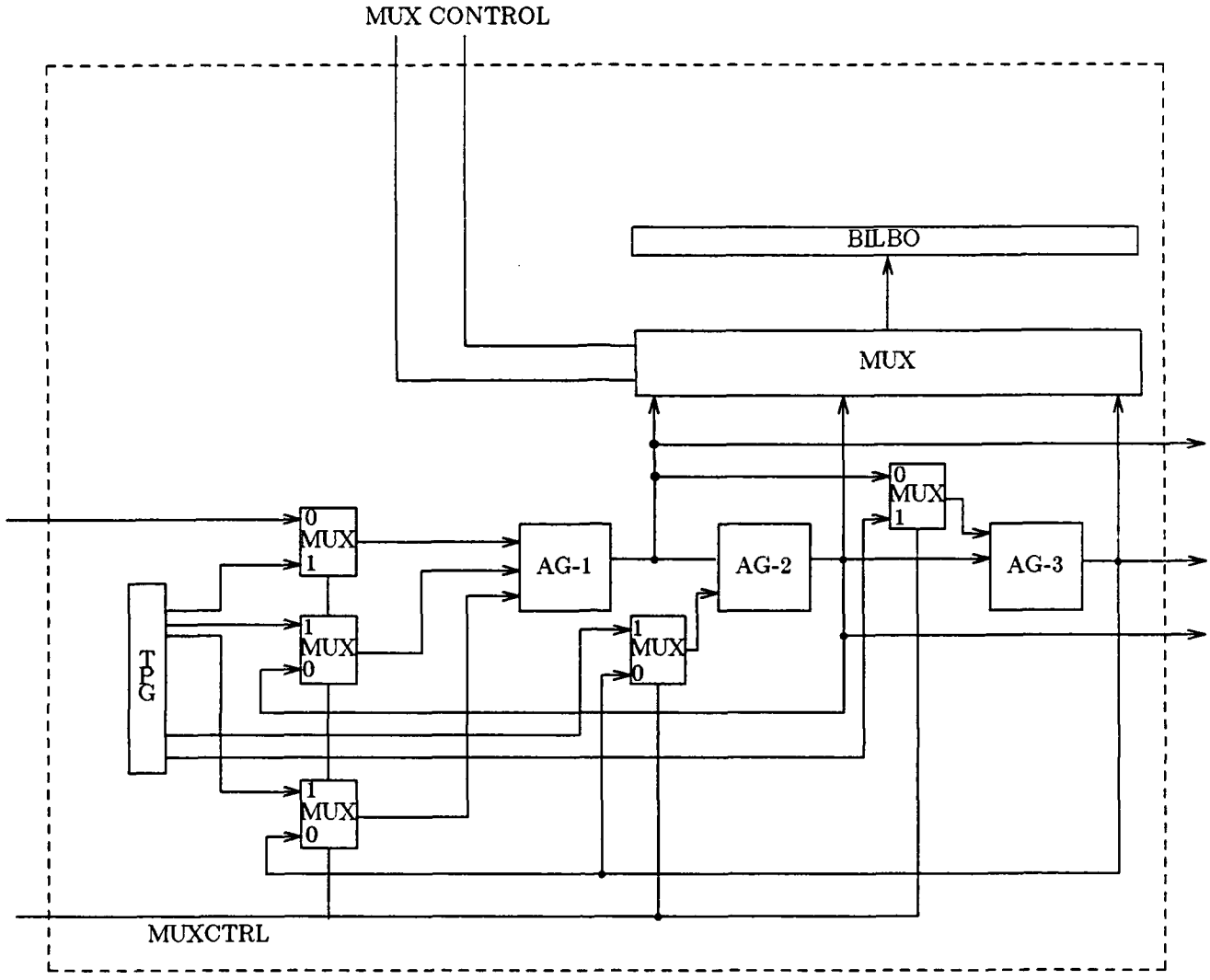


Figure B-6. BIT technique 2-b: Test point monitoring - data compression

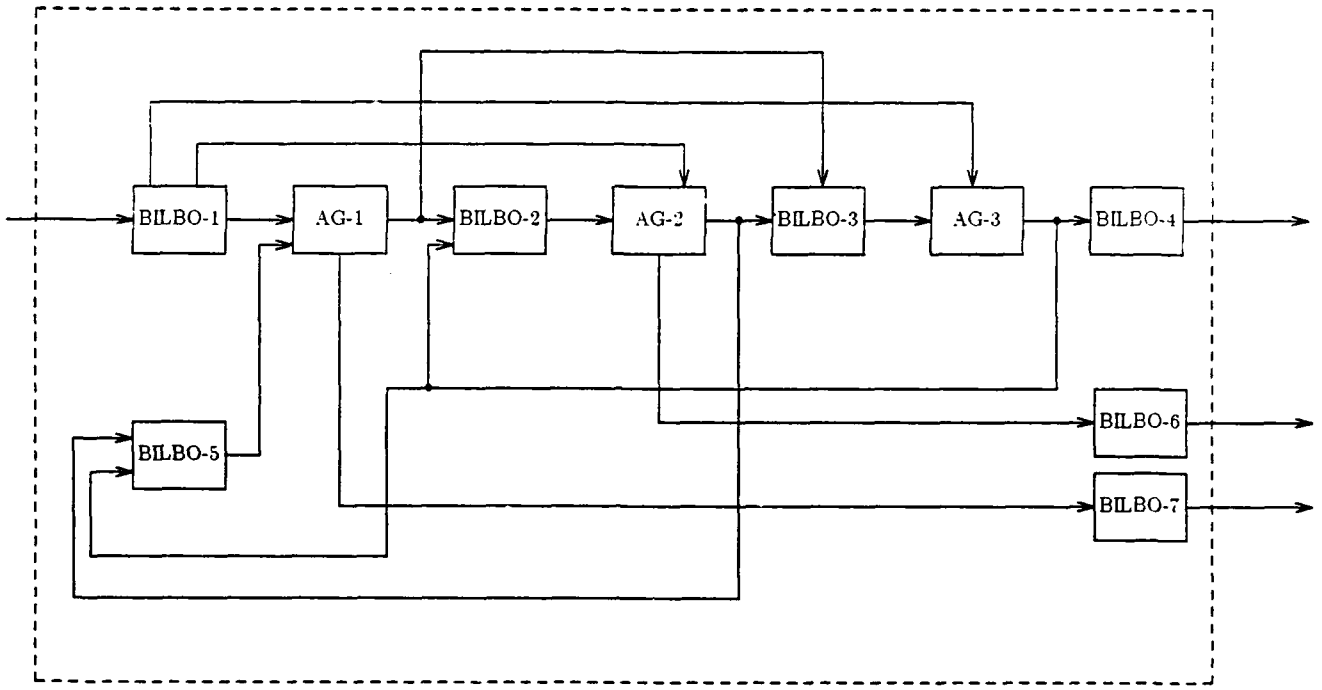


Figure B-7. BIT technique 5-b: Test pattern generation and response compression using "BILBO" modules distributed over the board

Figure B-8 shows a higher overhead alternative. This technique keeps lines from different sources isolated from each other for better fault isolation capability. Data and control isolation should also be maintained. A maintenance node is shown as a reminder that every board should have this monitoring and interface capability.

Only data lines are intercepted by BILBOs in the implementation of test pattern generation with data compression shown in Figure B-9. This results in lower overhead than what is experienced in the implemented version.

Only data lines are intercepted by TSWITCHes in the implementation of test pattern generation with data compression shown in Figure B-10. This results in lower overhead than what is experienced in the implemented version.

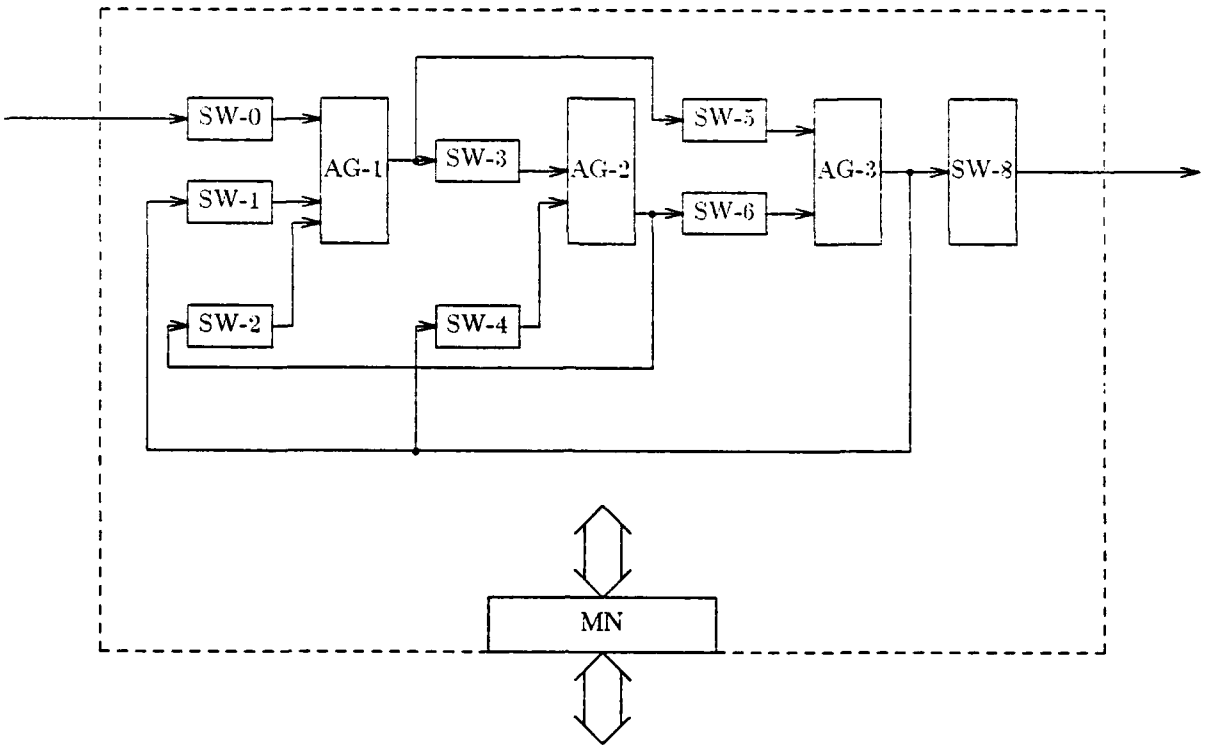


Figure B-8. BIT technique 5-c: Test pattern generation and response compression using "Testing Switch" modules - control and data are kept separate

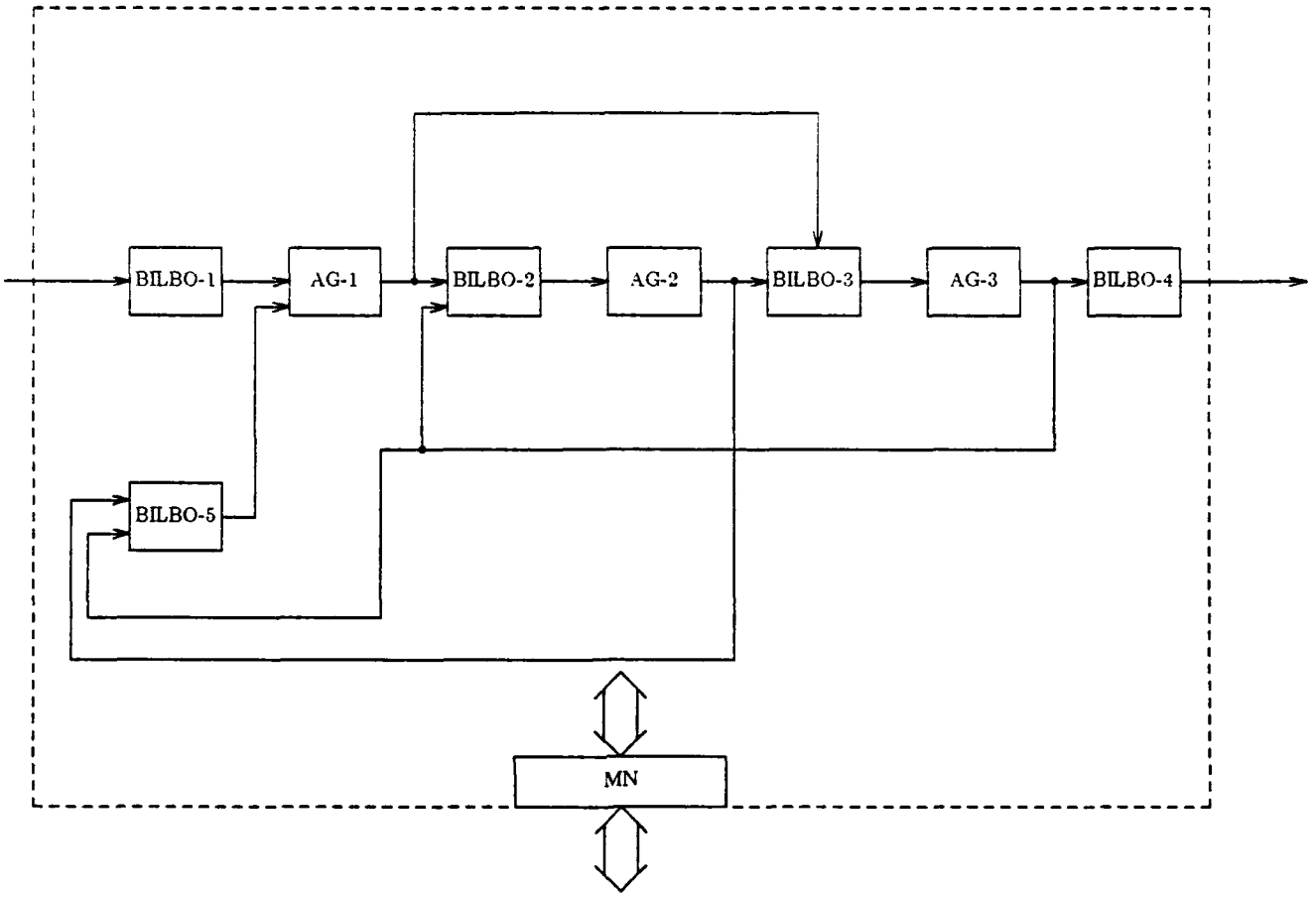


Figure B-9. BIT technique 5-d: Test pattern and response compression using "BILBO" modules distributed over the board - no BILBO for control lines

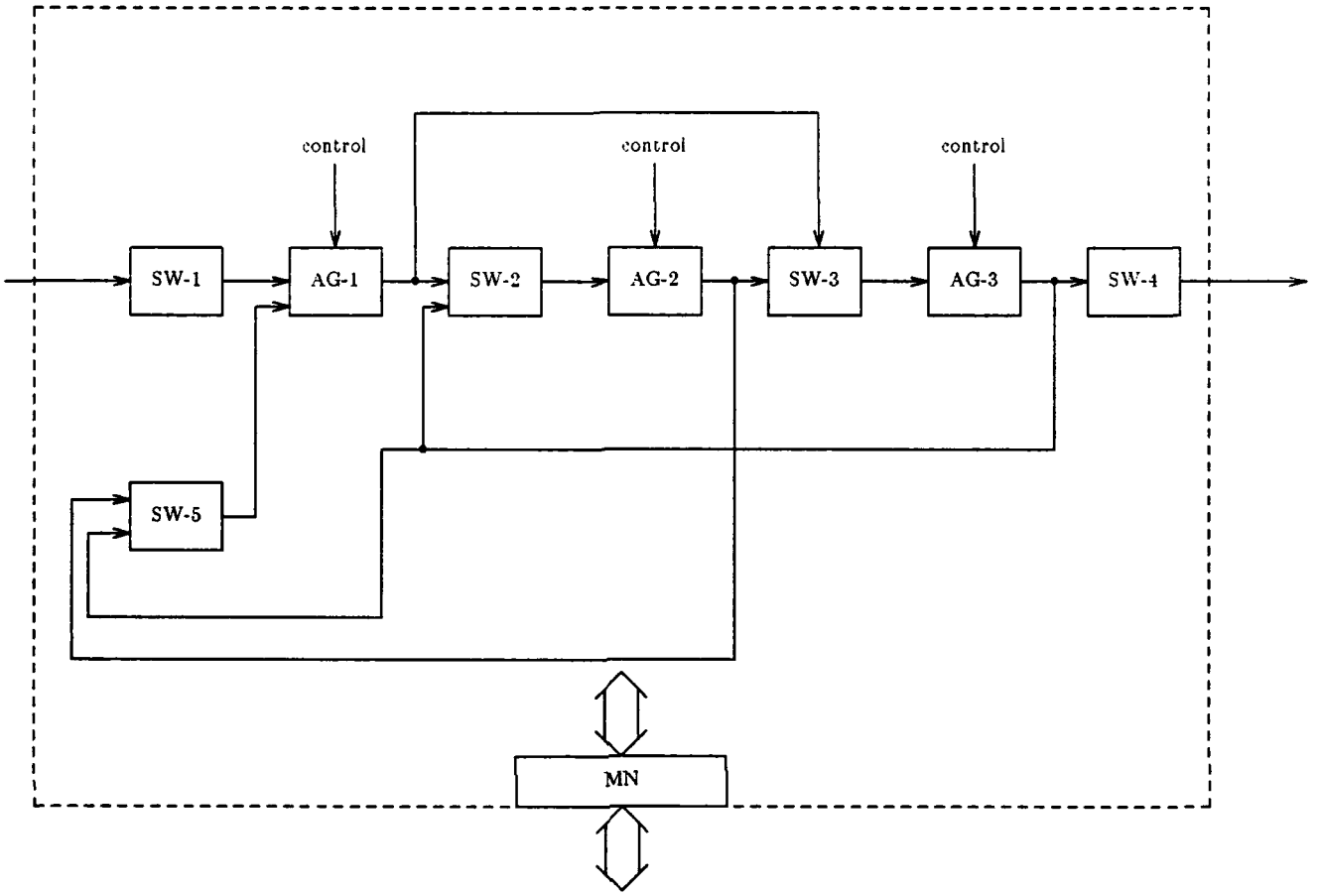


Figure B-10. BIT technique 5d: Test pattern generation and response compression using "Test Switch" modules distributed over the board - no "Testing Switch" for control lines

Appendix C
TEA BIT Module Specifications

TEA provides a library of built-in test support modules. Included in this library. the user receives

a. an application note describing the function of the module along with an example implementation and notes on using the module in a testing situation

b. an ADAS template to use when adding nodes representing BIT support modules into a graph

c. a VHDL, version 7.2, structural and/or behavioral description of the module

d. a gate level description of the model entered in CADAT (product of HHB-Systems of Mahwah, NJ) format

The BIT Overhead Summary (Paragraph 2.7.3) and BIT Placement Recommendation (Paragraph 2.7.4) tools use the library to pick out modules to support the BIT technique chosen by the user.

List of Modules Included in the TEA BIT Module Library
All VHDL models can be find in subdirectories off [..src]

BIT Modules with CADAT descriptions in [...cadat.version1.bit]
Structural models listed before behavioral models

Name	ADAS Template Name	VHDL Model Name	CADAT File Name	Notes
Built-in Logic Block Observer (BILBO)	BILBO	bilbo_ent bilbo_arch bilbo_beh	bilbo.ckt bilbo.dsl bilbo1.dsl	supports test pattern generation and signature analysis
Pseudorandom Test Pattern Generator (External Exclusive-or Implementation)	TPG	tpg_ext_ent tpg_ext_str tpg_ext_beh	tpg1.ckt tpg1.dsl	supports TEA BIT techniques
Pseudorandom Test Pattern Generator (Internal Exclusive-or Implementation)	IEO_TPG	tpg_int_ent tpg_int_str tpg_int_beh	tpg2.ckt tpg2.dsl	not used automatically by any TEA BIT technique
Scan-set	SCAN	ss_bit_ent ss_bit_arch ss_bit_beh	scanset.ckt scanset.dsl scanseta.dsl	supports several TEA BIT techniques
Testing-Switch	TSWITCH	test_switch_ent test_switch_str test_switch_beh	switch.ckt switch1.dsl switch2.dsl	supports **gen_sa** BIT technique

BIT Module with CADAT description in [...cadat.version2.maint_node]

No behavioral model available

Name	ADAS Template Name	VHDL Model Name	CADAT File Name	Notes
Maintenance Node	Maintenance_Node	main_node_ent main_node_str	mn.ckt mn.dsl	supports subsystem to board testing communications (mainly for controlling the BIT modules on the board)

Version 2 BIT Modules with CADAT descriptions in [...cadat.version2.bit]

Only structural models are available

Name	ADAS Template Name	VHDL Model Name	CADAT File Name	Notes
BILBO	BILBOv2	bilbo_ent_ver2 bilbo_arch_ver2	bilbo.ckt bilbo.dsl	interfaces to example maintenance node
Pseudorandom Test Pattern Generator (External Exclusive-or Implementation)	TPGv2	tpg_ext_ent_ver2 tpg_ext_arch_ver2	tpg1.ckt tpg1.dsl	interfaces to example maintenance node
Pseudorandom Test Pattern Generator (Internal Exclusive-or Implementation)	IEO_TPGv2	tpg_int_ent_ver2 tpg_int_arch_ver2	tpg2.ckt tpg2.dsl	interfaces to example maintenance node
Scan-set	SCANv2	ss_bit_ent_ver2 ss_bit_arch_ver2	scanset.ckt scanset.dsl	interfaces to example maintenance node

Version 2 BIT Module with CADAT descriptions in [...cadat.version2.switch]

Only structural model is available

Name	ADAS Template Name	VHDL Model Name	CADAT File Name	Notes
Testing-Switch	TSWITCHv2	test_switch_ent_ver2 test_switch_arch_ver2	mn.ckt mn.dsl	interfaces to example maintenance node

BIT Modules with VHDL descriptions in [...src.special]

No CADAT files available

Name	ADAS Template Name	VHDL Model Name	CADAT File Name	Notes
8-bit Equal Comparator	COMPARE	comp.entity comp.parts	NA	supports **compare** BIT technique
9-bit Parity Generator Checker	PARITY	parity.entity parity.parts	NA	supports **parity** BIT technique
MUX2	MUX2	mux2_by_16.entity mux2_by_16.behave	NA	2 sets of 16 lines input
MUX4	MUX4	mux4_by_16.entity mux4_by_16.behave	NA	4 sets of 16 lines input
MUX2x8	MUX2x8	mux2_by_8.entity mux2_by_8.behave	NA	2 sets of 8 lines input
MUX4x8	MUX4x8	mux4_by_8.entity mux4_by_8.behave	NA	4 sets of 8 lines input
Ones	ONES	ones.entity ones.behave	NA	16 lines of constant-valued high output
Zeros	ZEROS	zeros.entity zeros.behave	NA	16 lines of constant-valued low output

NA: Not Available

Appendix D

ADAS-related Information

PART I

ADAS Data Base File Format

This part defines a new ADAS data base file format which supports changes for this contract and other enhancements being made concurrently as part of ADAS development.

Each ADAS graph is stored in a graph data base file which contains all the information about the graph and its components. Eight different types of graph components exist: nodes, arcs, buses, partitions, junctions, inports, outports, and biports. Buses, junctions, and biports exist only in hardware graphs.

A template data base file is associated with each graph data base file and contains templates defining the characteristics of graph components present in the corresponding graph data base file. These templates are used in adding new instances of graph components to a graph.

A filename extension identifies an ADAS data base file as either

a. a hardware graph data base file (".hwg"),

b. a hardware template data base file (".hwt"),

c. a software graph data base file (".swg"), or

d. a software template data base file (".swt").

Each variation of the ADAS data base file has the same format. Information in the files is interpreted differently depending on the variety. ADAS graph and template data base files are ASCII text files. The ADAS data base routines interpret these files as consisting of records terminated by carriage returns (<CR>). No special character is used to delimit fields within these records except in cases where delimiters are explicitly mentioned. A record field may belong to one of the following types:

a. string — an ASCII string of arbitrary length;

b. string$ — an alphanumeric string of arbitrary length, terminated by a dollar sign ("$"): e.g., "newnode$";

c. decimal — an integer represented by an arbitrarily long string of decimal digits: e.g., "101";

d. B26 — a two-digit base 26 integer represented by using lower case letters of the alphabet (a = 0, b = 1, ..., z = 25). Fields of type B26 can represent integers in the range [0..675] ([aa..zz]). For example, the value "dx" for a field of type B26 represents the integer value 101:

$$d\ (= 3) * 26 + x\ (= 23) = 101.$$

An ADAS data base file consists of:

a. a header section,

b. the graph attribute list,

c. a node section containing twc or more records for each node in the graph,

d. a bus section containing two or more records for each bus in the graph,

e. an arc section containing two records for each arc in the graph, and

f. a partition section containing two records for each partition in the graph.

The header section contains eight records:

a. a record with the following fields delimited by spaces:

1. the keyword "adasdb_new,"

2. the number of nodes in the graph (decimal),

3. the number of arcs in the graph (decimal),

4. the number of buses in the graph (decimal),

5. the number of partitions in the graph (decimal);

b. the template search path for the graph (string);

c. the graph attribute definition list;

d. the node attribute definition list;

e. the bus attribute definition list;

f. the arc attribute definition list;

g. the partition attribute definition list;

h. the inport attribute definition list;

i. the outport attribute definition list;

j. the biport attribute definition list;

k. a record containing only the string "%end%," marking the end of the header.

Each attribute definition list has the following fields:

a. a one-letter code identifying the type of the object:

1. "G" — graph,

2. "N" — node,

3. "B" — bus,

4. "A" — arc,

5. "P" — partition,

6. "I" — inport,

7. "O" — outport,

8. "D" — biport;

b. the total number of attributes present for objects of that type (B26);

c. the ADAS internal data base name (attribute ID) for each attribute of that object type (string$).

An attribute definition list defines the set of ADAS attributes for its object type written to the data base file. This list may or may not contain the full set of ADAS attributes. An attribute list for a given object has entries only for the attributes appearing in the attribute definition list for its type.

Each attribute list is a record consisting of the following fields:

a. a one-letter code identifying the object type ("G," "N," "B," "A," "P," "I," "O," or "D"),

b. two fields for each attribute appearing in the attribute definition list for that object type:

1. a one-letter modifiability code:

a) "M" — modifiable,

b) "P" — modifiable only by ADAS programs (not by the user with the **edit** command),

c) "N" — not modifiable,

d) "U" — unused status;

2. the value of the attribute (string$).

For each node in the graph there are two or more records:

a. one record containing:

1. the node's (x, y) location on the graph grid (B26, B26),

2. the number of inports on the node (B26),

3. the number of outports on the node (B26),

4. the number of biports on the node (B26),

5. the node's name (string$);

b. a node attribute list;

c. an inport attribute list for each of the node's inports;

d. an outport attribute list for each of the node's outports;

e. a biport attribute list for each of the node's biports.

For each bus in the graph there are two or more records:

a. a record containing

1. the number of junctions on the bus (B26),
2. the bus name (string$);

b. a bus attribute list;

c. a record for each junction on the bus containing the junction's location (B26, B26).

For each arc in the graph are two records:

a. a record containing:

1. the arc's name (string$),
2. the (x, y) coordinates of its source (B26, B26),
3. the number of its source port or connection (B26),
4. the type of its source port (B26),
5. the (x, y) coordinates of its sink (B26, B26),
6. the number of its sink port or connection (B26),
7. the type of its sink port (B26);

b. an arc attribute list.

For each partition in the graph are two records:

a. the partition's name (string$), and

b. a partition attribute list.

PART II

ADAS Data Base Routines

FOREWORD

This part has been adapted from a UNIX manual page entry for the ADAS data base routines. Its intended audience is programmers writing software that uses the ADAS data base. The routines described herein constitute the interface to the ADAS data base for all ADAS related tools. Programs using these routines must include the file "adasdb.h".

CONTENTS

GRAPH COMPONENTS

The ADAS system uses directed graphs to model hardware and software systems. These graphs consist of collections of nodes (representing hardware processors or software modules) that are connected by arcs (representing data flow paths between nodes). Hardware graphs can also include buses which may be attached to nodes via arcs.

Node ports provide places on a node where arcs may attach and are referred to by numbers. If a node has N input ports, they are numbered 0 through N-1. Software graph nodes can have inports and outports, while hardware graph nodes can also have biports. Only one arc may be attached to a particular port. Bus junctions provide the connection points for arcs on buses. Any number of arcs may be attached to each junction.

Each element in a graph has a number of *attributes* that characterize it. When an element is created, the initial values for these attributes are taken from a *template* used to generate the element.

HIERARCHICAL STRUCTURE

ADAS graphs are hierarchical; they are organized in a tree structure consisting of one or more levels. The single graph at the top of this tree structure is called the root.

An individual node on one level may be expanded into an entire subgraph on the next level. In keeping with tree structure terminology, a node that has a subgraph is an *internal* node; otherwise, it is a *leaf* node. An internal node is the *parent* of all nodes in its subgraph.

Special nodes called *graph ports* represent the connections of a subgraph with nodes connected to its parent node. There is a one-to-one mapping between graph ports of a subgraph and node ports of the subgraph's parent node.

PURPOSE OF THE ADASDB LIBRARY

The *ADASDB* routines provide application programs with the ability to manipulate the elements of a graph, expand nodes into subgraphs, and traverse graph hierarchies.

The *ADASDB* routines embody the principles of information hiding. Application programs manipulate graphs and graph elements indirectly through the *ADASDB* routines and are not required to know the makeup of various internal data structures and file formats used to represent graphs.

ADASDB supports two different logical views of graphs. Applications programs may look at a graph in terms of its connectivity, or in terms of a grid with nodes at the grid intersections.

Finally, *ADASDB* provides attribute independence. Applications programs can only deal with relevant graph component attributes and ignore the rest. Also, adding new attributes is done transparently and requires only recompilation of the *ADASDB* library, followed by relinking of the applications programs.

GRAPH DATA BASES

ADAS graphs are defined by graph data bases. These data bases are normally stored as disk files. When an application program wants to access a graph, it invokes an *ADASDB* routine that reads the data base from the disk file and creates an internal (in-memory) version. All subsequent operations are performed on this internal version of the data base. Since this internal data base

exists only while the program executes, it must be written back to the disk file if modifications made to it are to be preserved.

VIEWING THE GRAPH HIERARCHY

The root and each subgraph of a graph hierarchy are kept on disk as separate data base files. This separation is maintained internally; when a node is expanded, its subgraph is kept as a separate data base. These data bases are arranged in a hierarchy that mirrors the subgraph hierarchy. A set of *ADASDB* routines is provided for traversing this data base hierarchy depth-first in an orderly manner.

Since the data bases in a graph hierarchy are kept separately, they can be manipulated individually, even when a graph is partially or fully expanded. This approach is useful when editing graphs where the connections between subgraphs at the same level are not needed.

An expanded graph can also be viewed as a hierarchy with a single graph at each level, formed by joining all the subgraphs at that level. This approach is useful in graph analysis when it is desirable to view all subgraphs at a particular level as a single graph.

THE CURRENT GRAPH AND CURRENT DATA BASE

Because there can be multiple internal data bases within a given graph hierarchy and because many *ADASDB* routines are designed to operate on a particular data base (e.g., most of the element retrieval routines), *ADASDB* maintains a ***current data base***. Data base-specific operations are performed on this current data base. The *DBPushDb* and *DBPopDb* routines change the current data base, allowing the data base hierarchy to be traversed.

ADASDB also can simultaneously handle multiple root graphs and multiple graph hierarchies. The *DBGraphInit* routine initializes new root graphs, and the *DBUseGraph* routine provides a way of specifying which graph to work on currently.

If multiple graph hierarchies are simultaneously in the memory, each hierarchy keeps track of its own current data base. When switching to a different hierarchy, the current data base becomes that of the current hierarchy.

ERROR HANDLING

If an *ADASDB* routine encounters an error condition, it returns immediately without performing its intended action. In addition, it sets a global error flag *dberrno* with an error code that indicates the nature of the error. The meaning of an error code in the context of a particular routine is described in that routine's definition. A list of error codes is provided at the end of this document. All error codes are negative integers.

Once *dberrno* is set, it is not reset unless another error occurs subsequently.

Aside from setting *dberrno*, *ADASDB* routines have other ways of indicating errors, depending on what kind of routine they are.

If a routine is a *procedure* (i.e., does not return a value), then it generally returns an integer completion code. A negative code indicates an error. A zero code indicates normal completion.

On the other hand, if a routine is a *function* (i.e., does return a value), then it generally indicates an error by returning a value that is out of its normal range. For instance, a routine that

normally returns a character string will return a null pointer (NULL) on error.

DEBUGGING FACILITIES

The *ADASDB* routines provide several debugging facilities for aiding installation and maintenance.

When the *ADASDB* subroutine library is compiled with the DEBUG option (this should be done in the *ADASDB* makefile), there is a trace facility and an assertion facility.

The trace facility prints useful information about what the data base routines are doing. Tracing is turned on by setting the external integer ***dbtrace*** to 1 and is disabled by setting *dbtrace* to 0. Tracing may be turned on or off at any point during a program's execution.

The assertion facility is not program controlled. It checks certain conditions within the data base routines that are assumed to be true. If an assumption is not met, the program is forced to exit. printing out the name and line number of the *ADASDB* source file where the problem occurred. This message should be reported to the *ADASDB* maintenance person who can then track down the problem.

Lint, the C program verifier, should be used when installing the *ADASDB* routines in a program to insure that procedure parameters are correct in type and number. A lint library for the *ADASDB* has been created and can be used as described above in the synopsis section.

CONVENTIONS USED IN THIS DOCUMENT

All upper-case keywords in this document are symbolic constants. These constants are defined in the header file ***adasdb.h***.

Italicized words are either new or important terms, or the names of *ADASDB* routines or their parameters.

DEFINITION OF ADASDB ROUTINES

The remainder of this document consists of brief definitions of the routines that comprise the *ADASDB* library.

INITIALIZING AND TERMINATING GRAPHS

int DBGraphInit(dbfile)
char *dbfile;

DBGraphInit initializes a graph, returning a descriptor for the new graph. This descriptor is a small non-negative integer like a UNIX file descriptor. The root data base of the new graph is the graph's default current data base. The root data base is initialized. If ***dbfile*** exists, it is read into the root data base.

Returns graph descriptor (small non-negative integer) on normal completion, (1) XINIT if there are no more unused graph descriptors, (2) XACCESS if it couldn't open external data base, (3) XFORMAT ifit couldn't read external data base, (4) XEOF if a premature end-of-file is encountered while reading external data base, and (5) XMEMORY if ran out of memory.

NOTE: this routine MUST be called before any data base operations are performed on this graph. See also *DBUseGraph()*.

DBUseGraph(gd)
int gd;

DBUseGraph makes graph hierarchy denoted by the graph descriptor *gd* the current graph. The descriptor *gd* must be one returned by the routine *DBGraphInit*.

Assumes *gd* is a valid graph descriptor.

The previous graph descriptor is returned.

DBTermGraph()

DBTermGraph deletes all nodes and arcs in the root of the current graph and makes the graph available for reinitialization by *DBGraphInit*. Since *DBTermGraph* frees all memory occupied by the terminated graph, graphs that are no longer being used should be terminated to reduce a program's memory requirements.

No return code.

int DBIsGraphInit(gd)
int gd;

DBIsGraphInit returns a nonzero value when the graph with descriptor *gd* is initialized and otherwise, a zero.

int DBGetCurrGraph()

DBGetCurrGraph returns the graph descriptor of the current graph.

boolean DBIsHWGraph()

DBIsHWGraph returns TRUE if the current graph is a hardware graph, and FALSE otherwise.

UPDATING THE EXTERNAL DATA BASE

DBWriteDb(dbfile)
char *dbfile;

DBWriteDb writes the contents of the current data base to the file *dbfile*. This is the only routine that actually modifies an external data base.

If *dbfile* already exists, it will be overwritten.

Returns XACCESS if *dbfile* cannot be opened for writing.

HANDLING SUBGRAPHS

The following routines manipulate subgraphs. A node is expanded if its class is INTERNAL (see *DBGetNodeClass()*) and it currently has an internal subgraph data base associated with it.

DBSubgraphInit(n, dbfile, newdb)
nodetype n;
char *dbfile;
boolean *newdb;

DBSubgraphInit creates a new subgraph for internal node *n*. Memory is allocated for the new data base, then a new subgraph is created by adding one graph in/out port for each in/out port of node *n*. Node *n* is made the "parent" of all the nodes in the new subgraph (see *DBGetParentNode()*). *DBSubgraphInit* sets *newdb* to TRUE if *dbfile* did not already exist, and to FALSE otherwise.

If data base file *dbfile* exists, it is read into the internal subgraph data base.

Returns XACCESS if *dbfile* exists but couldn't be opened, XFORMAT if *dbfile* exists but couldn't be read, XEOF if premature EOF encountered while reading *dbfile*, XMEMORY if we ran out of memory.

Assumes *n* is in the current data base. Assumes class of *n* is INTERNAL. Assumes *n* is not already expanded. Assumes we are not trying to expand past graph level MAXDEPTH.

DBFreeSubgraph(n)
nodetype n;

DBFreeSubgraph frees the space occupied by the internal subgraph data base of node *n*. This routine has no effect on the corresponding external data bases.

Assumes *n* is in the current data base, is class INTERNAL, and is currently expanded. Assumes no nodes in the subgraph are currently expanded.

No return code.

boolean DBIsExpanded(n)
nodetype n;

DBIsExpanded returns TRUE if the subgraph of node *n* is currently expanded, and false otherwise.

Assumes node *n* is internal.

TRAVERSING THE DATA BASE HIERARCHY

The following routines change the current data base. A stack of data bases allows the graph hierarchy to be traversed depth-first in an orderly manner.

DBPushDb(n)
nodetype n;

DBPushDb pushes the current data base onto the data base stack and makes the subgraph data base of node *n* the new current data base.

Assumes *n* is an internal node in the current data base, and it is currently expanded.

No return code.

DBPopDb()

DBPopDb pops the top data base from the data base stack and makes it the new current data base.

Assumes current data base is a subgraph (i.e., graph level is > 0).

No return code.

int DBGetCurrGraphLevel()

DBGetCurrGraphLevel returns the level within the graph hierarchy of the current data base, i.e., the current data base stack level.

char *DBGetCDBFileName()

DBGetCDBFileName returns the name of the ADAS data base file associated with the current data base.

ACCESSING GRAPH ELEMENTS THROUGH CONNECTIVITY

Graph connectivity can be viewed in one of two ways. The normal (default) view looks at node, arc, and bus connections within a single data base. The alternate view looks at the connections that exist between subgraphs at the lowest level of the graph hierarchy only when the graph is flattened (see *DBViewGraph()*). This alternate view makes all subgraphs at the lowest level appear to be a single graph and allows the inter-data base connections to be traversed.

int DBViewGraph(view)
int view;

DBViewGraph sets the view of the graph hierarchy that will be used for connectivity purposes. If the view is NORMAL, the graph hierarchy is viewed as a collection of subgraphs that may be separately manipulated. This is the default view.

If the view is FLAT, the entire hierarchy is (conceptually) flattened into a single graph consisting of all leaf nodes in the hierarchy.

Returns previous view.

Only connectivity routines, e.g., *DBGetNextNode*, *DBGetOutputArc*, *DBGetSinkNode*, are directly affected by the graph view. When the flat view is used, the *DBSetFirstNode*/*DBGetNextNode* routines may be used to step through all nodes in the flattened graph, and the *DBGetSource*/*DBSinkNode* and *DBGetInput*/*DBOutputArc* routines automatically follow inter-subgraph and inter-level connections.

After *DBViewGraph* is called initially, subsequent calls are quite cheap in terms of the amount of work the routine has to do.

NOTE: Because of the way *DBViewGraph* is implemented, changes to graph connectivity should never be made after *DBViewGraph* is called. Changes include adding/deleting nodes or arcs, expanding/unexpanding subgraphs, etc.

NOTE: *DBViewGraph* does not actually expand subgraphs, it only flattens subgraphs that are already expanded. Expansion must be done explicitly by the user program before *DBViewGraph* is called for the first time. If a node's class is INTERNAL, but the node is not expanded, it is considered a leaf node for connectivity purposes and is included in the flattened graph.

DBGetSourceNode(a, np, portp)
arctype a;
nodetype *np;
int *portp;

DBGetSourceNode sets the pointer *np* to point to the source node for arc *a*. The integer pointed to by *portp* is set to the outport where *a* is attached. If the graph is flattened (see *DBViewGraph()*), inter-data base connectivity is used; otherwise, normal connectivity is used.

No return code.

Although calls to *DBGetSourceNode* are still supported, this routine has been superceded by *DBGetSource*. New code should use *DBGetSource* instead of *DBGetSourceNode*.

DBGetSource(a, bp, junction, np, conn, ioflag)
arctype a;
bustype *bp;
int *junction;
nodetype *np;
int *conn;
int *ioflag;

DBGetSource returns the source of arc *a*. If the source is a bus, *DBGetSource* sets the pointer *bp* to point to the source bus and sets *junction* to the junction index; *conn* to the connection number of the arc; and *np* to NULL. If the source is a node, *np* is set to point to the source node; *conn* is set to the source port number; *ioflag* is set to the type of the port to which the arc is connected (OUT or BI); and *bp* is set to NULL.

Note, there is no notion of "source" or "sink" for the connection of an arc to a junction or node biport. A bus is considered the source of an arc only if the other end of the arc is attached to a node inport. If an arc is attached to biports on both ends, the assignment of source and sink depends on which node is chosen as the source at the time the arc is created. *DBGetSource* and *DBGetSink* are guaranteed not to return the same node.

DBGetSinkNode(a, np, portp)
arctype a;
nodetype *np;
int *portp;

DBGetSinkNode sets the node pointed to by *np* to the sink node for arc *a*. The integer pointed to by *portp* is set to the inport where the *a* is attached.

If the graph is flattened (see *DBViewGraph()*), inter-data base connectivity is used; otherwise, normal connectivity is used.

No return code.

Although calls to *DBGetSink* are still supported, this routine has been superceded by *DBGetSink*. New code should use *DBGetSink* instead of *DBGetSink*.

DBGetSink(a, bp, junction, np, conn, ioflag)
arctype a;
bustype *bp;
int *junction;
nodetype *np;
int *conn;
int *ioflag;

DBGetSink returns the sink of arc *a*. If the sink is a bus, it sets the pointer *bp* to point to the bus, sets *junction* to the junction index, *conn* to the connection number, and *np* to NULL. If the sink is a node, it sets the pointer *np* to the node, *conn* to the port index, *ioflag* to the port type, and *bp* to NULL.

Note no notion of "source" or "sink" exists for the connection of an arc to a junction or node biport. A bus is considered the sink of an arc if the other end of the arc is attached to a node outport or biport. If an arc is attached to biports on both ends, the assignment of source and sink depends on which node was specified as the source when the arc was created. *DBGetSource* and *DBGetSink* are guaranteed not to return the same node.

DBGetInputArc(n, port, ap)
nodetype n;
int port;
arctype *ap;

DBGetInputArc sets the arc pointed to by *ap* to the input arc associated with inport *port* on node *n*.

If the graph is flattened (see *DBViewGraph()*), inter-data base connectivity is used; otherwise, normal connectivity is used.

Assumes *port* is a valid port number for node *n*; assumes an arc is connected at *port*.

No return code.

Although calls to *DBGetInputArc* are still supported, this routine has been superceded by *DBGetNodeArc*. New code should use *DBGetNodeArc* instead of *DBGetInputArc*.

DBGetOutputArc(n, port, ap)
nodetype n;
int port;
arctype *ap;

DBGetOutputArc sets the arc pointed to by *ap* to the output arc associated with outport *port* on node *n*.

If the graph is flattened (see *DBViewGraph()*), inter-data base connectivity is used: otherwise, normal connectivity is used.

Assumes *port* is a valid port number for node *n*; assumes there is an arc connected at *port*.

No return code.

Although calls to *DBGetOutputArc* are still supported, this routine has been superceded by *DBGetNodeArc*. New code should use *DBGetNodeArc* instead of *DBGetOutputArc*.

DBGetNodeArc(n, port, ioflag, ap)
nodetype n;
int port;
int ioflag;
arctype *ap;

DBGetNodeArc sets the pointer *ap* to point to the arc associated with port number *port* of the port type specified by *ioflag* (inport, outport, or biport) on node *n*.

DBGetBusArc(b, junction, conn, ap)
bustype b;
int junction;
int conn;
arctype *ap;

DBGetBusArc sets the pointer *ap* to point to the arc connected at connection number *conn* on junction *junction* of bus *b*.

DBGetParentNode(n, parent)
nodetype n;
nodetype *parent;

DBGetParentNode sets the node pointed to by *parent* to the parent node of node *n*, or NULL if *n* has no parent, which only happens if *n* is in the root data base.

No return code.

DBC BusParentNode(b, parent)
busty, b;
nodetype *parent;

DBGetParentNode sets the node pointed to by *parent* to the parent node of bus *b*, or NULL if *b* has no parent, which only happens if *b* is in the root data base.

No return code.

DBGetParentArc(a, parent)
arctype a, *parent;

DBGetParentArc sets the arc pointed to by *parent* to the parent arc of *a*, or NULL if *a* has no parent, which only happens if *a* is in the root data base.

No return code.

ACCESSING GRAPH ELEMENTS IN THE CURRENT DATA BASE

DBGetNodeByLoc(x, y, np)
int x, y;
nodetype *np;

DBGetNodeByLoc sets the node pointed to by *np* to the node at graph location (x,y) in the current data base.

Assumes location is in bounds.

Returns XNOTFOUND if there is no node at location (x,y).

DBGetNodeByName(name, np)
char *name;
nodetype *np;

DBGetNodeByName sets the pointer *np* to point to node *name* in the current data base. Returns XNOTFOUND if there is no such node in the current data base.

DBGetArcByName(name, ap)
char *name;
arctype *ap;

DBGetArcByName sets the arc pointer *ap* to point to arc *name* in the current data base. Returns XNOTFOUND if there is no such arc in the current data base.

DBGetBusByName(name, bp)
char *name;
bustype *bp;

DBGetBusByName sets the pointer *bp* to point to bus *name* in the current data base. Returns XNOTFOUND if there is no such bus in the current data base.

DBGetPartitionByName(name, pp)
char *name;
parttype pp;

DBGetPartitionByName sets the pointer *pp* to point to the partition named *name* in the current data base. Returns XNOTFOUND if there is no such partition in the current data base.

DBGetJunctionByLoc(x, y, bp, jp)
int x:
int y;
bustype *bp;
int *jp

DBGetJunctionByLoc sets the pointer *bp* to point to the bus and the integer pointed to by *jp* to the index of the junction at coordinates (*x*, *y*).

DBSetFirstNode()

DBSetFirstNode resets the node list of the current data base (or flattened graph) to an arbitrary first node. See the explanation of *DBGetNextNode* for more detail.

No return code.

DBGetNextNode(np)
nodetype *np;

DBGetNextNode, when used in conjunction with *DBSetFirstNode* and *DBMoreNodes*, permits sequential access to all nodes in the current data base. A call to *DBSetFirstNode* resets a node list maintained in the current data base. Each subsequent call to *DBGetNextNode* sets the node pointed to by *np* to an arbitrary next node in the current data base until all nodes have been retrieved. *DBMoreNodes* returns TRUE while there are still nodes in the list.

If the graph view is FLAT (see *DBViewGraph*), instead of using the node list of the current data base, *DBSetFirstNode*, *DBMoreNodes*, and *DBGetNextNode* operate on the node list for the entire flattened graph.

Assumes there are more nodes in the list (see *DBMoreNodes*).

No return code.

boolean DBMoreNodes()

DBMoreNodes returns TRUE while there are still nodes in the node list of the current (or flattened) data base. See *DBGetNextNode*.

DBSetFirstArc()

DBSetFirstBus resets the arc list of the current data base to an arbitrary first arc. See *DBGetNextArc*.

DBGetNextArc(ap)
arctype *ap;

DBGetNextArc, when used in conjunction with *DBSetFirstArc* and *DBMoreArcs*, permits sequential access to all arcs in the current data base. A call to *DBSetFirstArc* resets an arc list

maintained in the current data base. Each subsequent call to *DBGetNextArc* sets the arc pointed to by *ap* to an arbitrary next arc in the current data base until all arcs have been rerrieved. *DBMoreArcs* returns TRUE while there are still arcs in the list.

boolean DBMoreArcs()

DBMoreArcs returns TRUE while there are still arcs in the arc list of the current data base. See *DBGetNextArc*.

DBSetFirstBus()

DBSetFirstBus resets the bus list of the current data base to an arbitrary first bus. See *DBGetNextBus*.

DBGetNextBus(bp)
bustype *bp;

DBGetNextBus, when used in conjunction with *DBSetFirstBus* and *DBMoreBuses*, permits sequential access to all buses in the current data base. A call to *DBSetFirstBus* resets a bus list maintained in the current data base. Each subsequent call to *DBGetNextBus* sets the bus pointed to by *bp* to an arbitrary next bus in the current data base until all buses have been retrieved. *DBMoreBuses* returns TRUE while there are still buses in the list.

boolean DBMoreBuses()

DBMoreBuses returns TRUE while there are still buses in the bus list of the current data base. See *DBGetNextBus*.

DBSetFirstPartition()

DBSetFirstPartition resets the partition list of the current data base to an arbitrary first partition. See *DBGetNextPartition*.

DBGetNextPartition(pp)
parttype *pp;

DBGetNextPartition, when used in conjunction with *DBSetFirstPartition* and *DBMorePartitions*, permits sequential access to all partitions in the current data base. A call to *DBSetFirstPartition* resets a partition list maintained in the current data base. Each subsequent call to *DBGetNextPartition* sets the partition, pointed to by *pp*, to an arbitrary next partition in the current data base until all partitions have been retrieved. *DBMorePartitions* returns TRUE while there are still partitions in the list.

boolean DBMorePartitions()

DBMoreArcs returns TRUE while there are still partitions in the partition list of the current data base. See *DBGetNextPartition*.

DBGetGraphPort(port, ioflag, np)
int port;
int ioflag;
nodetype *np;

DBGetGraphPort sets the node pointed to by *np* to the graph inport or outport *port* of the current data base. It assumes *ioflag* is either IN (for graph inport) or OUT (for graph outport).

Returns XNOTFOUND if no graph port *port* is found in the current data base.

DBGetGraphParent(np)
nodetype *np;

DBGetGraphParent sets the node pointed to by *np* to the parent node of the current subgraph.

It assumes the current data base is a subgraph (not the root graph).

No return code.

ADDING AND DELETING GRAPH ELEMENTS

When a graph element is added to the internal data base, it is initialized using a previously defined template. Initialization fills in the initial attributes of the graph element using the template attributes. These templates are found by scanning the files in the template search path (see *DBSetTSPath()*).

Graph element names must begin with a letter, contain letters and/or digits, and be no longer than ATTLEN characters. Routines in this section that operate on node ports take a parameter called *ioflag*. This flag must have a value of IN, OUT, or BI, indicating whether the operation is to be performed on an inport, outport, or biport.

DBAddNode(x, y, template, name, np)
int x, y;
char *template;
char *name;
nodetype *np;

DBAddNode adds a new node to the current data base at location (x,y). Inports and outports are allocated to the node and attributes are initialized using the specifications in *template*. *Name* is the name given to the node. If *name* is NULL, *template* is used as the node name, with digits appended as needed to make the name unique.

Sets the pointer *np* to point to the new node.

Returns (1) XLOCATION if location $(x.y)$ is already occupied by another node or is out of bounds.

(2) XACCESS if it couldn't access one of the template files in the search path, (3) XFORMAT if it couldn't read one of the template files in the search path, (4) XEOF if it encountered a premature end-of-file while reading a template file, (5) XNOTFOUND if *template* was not found, and (6) XNAME if *name* is already being used by another node or does not have the proper format.

DBAddInport(n)
nodetype n;

DBAddInport adds a new inport to node *n*. The index of the new inport will be one greater than the highest index of an inport currently on node *n*, or will be zero if there are no inports.

DBAddOutport(n)
nodetype n;

DBAddOutport adds a new outport to node *n*. The index of the new outport will either be one greater than the highest index of an outport currently on node *n* or zero if there are no outports.

DBAddBiport(n)
nodetype n;

DBAddBiport adds a new biport to node *n*. The index of the new biport will either be one greater than the highest index of an biport currently on node *n* or zero if there are no biports.

DBReplicatePort(n, count, ioflag)
nodetype n;
int count;
int ioflag;

DBReplicatePort copies the single port of type *ioflag* on node *n* so that *n* has *count* ports of that type. It is used to create node templates from the master node template.

DBAddArc(src, srcport, srcflag, sink, sinkport, sinkflag, bus, template, name, ap)
nodetype src;
int srcport;
int srcflag;
nodetype sink;
int sinkport;
int sinkflag;
bustype bus;
char *template;
char *name;
arctype *ap;

DBAddArc adds a new arc to the data base, connecting it to its source and sink. If *src* is not NULL, it is the source node of the arc; otherwise, *bus* is the arc's source. If *sink* is not NULL, it is the sink of the arc; otherwise, *bus* is the arc's sink. Obviously, either the source or sink, but not both, may be a bus. The source and sink ports to which the new arc is connected are given by *srcport* and *sinkport*. If the source or sink is a bus, *srcport* or *sinkport* gives the index of the

junction to which the arc is attached. The values of *srcflag* and *sinkflag* indicate the types of the source and sink ports. *Srcflag* can be OUT for an outport, or BI for a biport. *Sinkflag* can be IN for an inport, or BI for a biport. The arc template named *template* is used to initialize the new arc. *Name* is the name given to the arc. If *name* is null, *template* is used as the arc name, with digits appended as needed to make the name unique.

Sets the pointer *ap* to point to the new arc.

Returns (1) XPORT if arcs are already attached at *srcport* or *sinkport*, (2) XACCESS if it couldn't access one of the template files in the search path, (3) XFORMAT if it couldn't read one of the template files in the search path, (4) XEOF if it encountered premature end-of-file while reading a template file, (5) XNOTFOUND if *template* does not exist, and (6) XNAME if *name* is already being used by another arc or does not have the proper format.

DBAddBus(template, name, bp)
char *template;
char *name;
bustype *bp;

DBAddBus adds a new bus to the current data base. Attributes are initialized using the specifications in the bus template named *template*. *Name* is the name given to the bus. If *name* is NULL, *template* is used as the bus name, with digits appended as needed to make the name unique. The new bus is created without junctions. The pointer *bp* is set to point to the new bus.

DBAddPartition(name, color, nlist, ncount, pp)
char * name;
char * color;
char * nlist;
char * ncount;
parttype *pp;

DBAddPartition adds a partition with name *name* to the current data base. The new partition's *partition_color*, *partition_node_count*, and *partition_node_list* attributes are set to the values *color*, *nlist*, and *ncount*. Other attributes are initialized using the specifications in the partition template from the master template file. The pointer *pp* is set to point to the new partition.

DBAddJunction(b, junction, x, y)
bustype b;
int junction;
int x, y;

DBAddJunction adds a new junction to bus *b*. The junction is added at location *(x, y)* with the index *junction*, which specifies its position in the ordered list of the bus's junctions.

DBAddNode2(tmpl, x, y, np)
nodetype tmpl;
int x, y;
nodetype *np;

DBAddNode2 adds a new node to the current data base at location (*x,y*). Existing node *tmpl* is the node template that will determine the initial characteristics of the new node, including node attributes and the number of inports and outports. A unique name is generated for the new node by appending one or two digits onto the name of node *tmpl*. The node pointed to by *np* is set to the new node.

Returns XMEMORY if the memory runs out while trying to allocate space for this node. Returns XNAME if the new name generated exceeds the maximum length.

Assumes location (x,y) is in bounds and not currently occupied by another node.

DBAddArc2(tmpl, src, srcport, srcflag, sink, sinkport, sinkflag, bus, ap)
arctype tmpl;
nodetype src;
int srcport;
int srcflag;
nodetype sink;
int sinkport;
int sinkflag;
bustype bus;
arctype *ap;

DBAddArc2 adds a new arc to the current data base. *src* and *srcport* will be the source node and source port of the new arc; *sink* and *sinkport* will be the sink node and port. If *src* is NULL, *bus* is the source of the arc and *srcport* is the junction on the bus where the arc is attached. If *sink* is NULL, *bus* is the arc sink and *sinkport* is the junction. Existing arc *tmpl* is used to initialize the attributes of the new arc. A unique name is generated by appending one or two digits onto the arc name of *tmpl*. The arc pointed to by *ap* is set to the new arc.

Returns XMEMORY if the memory runs out while trying to allocate space for a new arc structure.

Assumes source and sink nodes are in current data base; assumes the nodes have the indicated ports and arcs are not already attached there.

DBAddBus2(tmpl, bp)
bustype tmpl;
bustype *bp;

DBAddBus2 adds a new bus to the current data base. Attributes are initialized using the specifications in the bus template *tmpl*. The name of the new bus is created by appending digits to the template name. The new bus is created without having junctions. The pointer *bp* is set to point to the new bus.

DBDeleteNode(n)
nodetype n;

DBDeleteNode removes the node *n* and its associated arcs from the current data base.

No return code.

DBDeleteInport(n, inport)
nodetype n;
int inport;

DBDeleteInport deletes the inport with index *inport* from node *n*. The inport is removed from position *inport*, and inports with higher indices have their indices decremented by one.

DBDeleteOutport(n, outport)
nodetype n;
int outport;

DBDeleteOutport deletes the outport with index *outport* from node *n*. The outport is removed from position *outport*, and outports with higher indices have their indices decremented by one.

DBDeleteBiport(n, biport)
nodetype n;
int biport;

DBDeleteBiport deletes the biport with index *biport* from node *n*. The biport is removed from position *biport*, and biports with higher indices have their indices decremented by one.

DBDeleteArc(a)
arctype a;

DBDeleteArc disconnects arc *a* from the source and sink ports where it is attached, then removes *a* from the data base.

No return code.

DBDeleteBus(b)
bustype b;

DBDeleteBus removes the bus *b* and any arcs attached to its junctions from the current data base.

DBDeletePartition(p)
parttype p;

DBDeletePartition deletes partition *p* from the current data base.

DBDeleteJunction(bus, junction)
bustype bus;
int junction;

DBDeleteJunction deletes junction *junction* of bus *bus*. Junctions in the bus having indices higher than *junction* will have their indices decremented by one.

int DBAddGPorts(tgd, n, tmpflag)
int tgd:
nodetype n;
boolean tmpflag;

DBAddGPorts adds graph port nodes corresponding to the ports of node *n* to the current subgraph. If templates are needed for these new nodes, they are added to the template data base with graph descriptor *tgd*, and *tmpflag* is set to TRUE.

GETTING AND SETTING GRAPH ELEMENT NAMES

Graph elements names are not handled like other attributes. The routines in this section are used to determine and modify graph element names.

char* DBGetNodeName(n)
nodetype n;

DBGetNodeName returns the name of node *n*; it also returns a pointer to a static buffer that is overwritten each time the routine is called, so the user should make a copy of the string, should it need to be saved.

DBRenameNode(n, name)
nodetype n;
char *name;

DBRenameNode changes the name of node *n* to *name*.

Returns XNAME if *name* is not valid or already being used by another node.

char* DBGetArcName(a)
arctype a;

DBGetArcName returns the name of arc *a* and returns a pointer to a static buffer that is overwritten each time the routine is called, so the user should make a copy of the string if it needs to be saved.

DBRenameArc(a, name)
arctype a;
char *name;

DBRenameArc changes the name of arc *a* to *name*.

Returns XNAME if *name* is not valid or already being used by another arc.

char* DBGetBusName(b)
bustype b;

DBGetBusName returns the name of bus *b*; returns a pointer to a static buffer that is overwritten each time the routine is called, so the user should make a copy of the string if it needs to be saved.

DBRenameBus(b, name)
bustype b;
char *name;

DBRenameBus changes the name of bus *b* to *name*.

char* DBGetPartitionName(p)
parttype p:

DBGetPartitionName returns the name of partition *p*. It returns a pointer to a static buffer that is overwritten each time the routine is called, so the user should make a copy of the string if it needs to be saved.

DBRenamePartition(p, name)
parttype *p;
char *name;

DBRenamePartition changes the name of partition *p* to *name*.

GETTING AND SETTING GRAPH ELEMENT FLAG VALUES

Every graph element is equipped with a general purpose flag and a utility bit flag. The general purpose flag is an integer. The utility flag provides eight independent boolean flags, numbered 0 through 7. Both the general flag and the utility flags are provided for use by application programs for their own purposes. These flags are not used in any way by the data base routines. The following routines are used to get and set values for these flags.

int DBGetNodeFlag(n)
nodetype n;

DBGetNodeFlag returns the value of node *n*'s general purpose flag.

DBSetNodeFlag(n, val)
nodetype n;
int val;

DBSetNodeFlag sets the general purpose flag of node *n* to *val*.

boolean DBGetUtilFlag(n, flag)
nodetype n;
int flag;

DBGetUtilFlag returns the value of node *n*'s utility bit flag *flag*.

DBSetUtilFlag(n, flag, val)
nodetype n;
int flag;
boolean val;

DBSetUtilFlag sets the value of node *n*'s utility bit flag *flag* to *val*.

int DBGetArcFlag(a)
arctype a;

DBGetArcFlag returns the value of arc *a*'s general purpose flag.

DBSetArcFlag(a, val)
arctype a;
int val;

DBSetArcFlag sets flag of arc *a* to *val*.

boleanDBGetArcUtilFlag(a, flag)
arctype a;
int flag;

DBGetArcUtilFlag returns the value of arc *a*'s utility bit flag *flag*.

DBSetArcUtilFlag(a, flag, val)
arctype a;
int flag;
boolean val;

DBSetArcUtilFlag sets the value of arc *a*'s utility bit flag *flag* to *val*.

int DBGetBusFlag(b)
bustype b;

DBGetBusFlag returns the value of bus *b*'s general purpose flag.

DBSetBusFlag(b, val)
bustype b;
int val;

DBSetBusFlag sets the general purpose flag of bus *b* to *val*.

boolean DBGetBusUtilFlag(b, flag)
bustype b;
int flag;

DBGetBusUtilFlag returns the value of bus *b*'s utility bit flag *flag*.

DBSetBusUtilFlag(b, flag, val)
bustype b;
int flag;
boolean val;

DBSetBusUtilFlag sets the value of bus *b*'s utility bit flag *flag* to *val.*

int DBGetPartitionFlag(p)
parttype p;

DBGetPartitionFlag returns the value of partition *p*'s general purpose flag.

DBSetPartitionFlag(p, val)
parttype p;
int val;

DBSetPartitionFlag sets the general purpose flag of partition *p* to *val.*

boolean DBGetPartUtilFlag(p, flag)
parttype p;
int flag;

DBGetPartUtilFlag returns the value of partition *p*'s utility bit flag *flag.*

DBSetPartUtilFlag(p, flag, val)
parttype p;
int flag;
boolean val;

DBSetPartUtilFlag sets the value of partition *p*'s utility bit flag *flag* to *val.*

int DBGetPortFlag(n, port, ioflag)
nodetype n;
int port;
int ioflag;

DBGetPortFlag returns the value of the general purpose flag for port *port* of type *ioflag* on node *n.*

DBSetPortFlag(n, port, ioflag, val)
nodetype n;
int port;
int ioflag;
int val;

DBSetPortFlag sets the value of the general purpose flag for port *port* of type *ioflag* on node *n* to *val.*

boolean DBGetPortUtilFlag(n, port, ioflag, flag)
nodetype n;
int port:
int ioflag;
int flag;

DBGetPortUtilFlag returns the value of utility bit flag *flag* for port *port* of type *ioflag* on node *n*.

DBSetPortUtilFlag(n, port, ioflag, flag, val)
nodetype n;
int port;
int ioflag;
int flag;
boolean val;

DBSetPortUtilFlag sets the value of utility bit flag *flag* for port *port* of type *ioflag* on node *n* to *val*.

MISCELLANEOUS OPERATIONS ON GRAPH ELEMENTS

Routines in this section that operate on node ports and graph ports take a parameter called *ioflag*. This flag must have a value of either IN, OUT, or BI, indicating whether the operation is to be performed on an inport, outport, or biport.

DBMoveNode(n, x, y)
nodetype n;
int x, y;

DBMoveNode moves the node *n* to the new location (*x*,*y*) in the current data base.

Assumes location is in bounds and not occupied by another node. It assumes node *n* is in the current data base.

DBGetNodeLoc(n, xp, yp)
nodetype n;
int *xp, *yp;

DBGetNodeLoc copies the (x,y) coordinates of node *n* within the current data base into the integers pointed to by *xp* and *yp*.

int DBGetPortCount(n, ioflag)
nodetype n;
int ioflag;

DBGetPortCount returns the number of inports, outports, or biports associated with node *n*.

boolean DBPortHasArc(n, port, ioflag)
nodetype n;
int port;
int ioflag;

DBPortHasArc returns TRUE if there is an arc attached at inport, outport, or biport *port* on node *n*, and FALSE if not.

Assumes *port* is a valid port number for node *n*.

int DBGetGraphPortCount(ioflag)
int ioflag;

DBGetGraphPortCount returns the number of graph inports, outports, or biports in the current data base.

int DBSetGraphPort(n, port)
nodetype n;
int port;

DBSetGraphPort sets the graph port number of node *n* to *port*. Note that this number is only meaningful if *n* is a graph port, i.e., its class is either "graph_inport," "graph_outport," or "graph_biport."

Returns XRANGE if *port* is less than zero or greater than MAXPORTS.

int DBGetGPortNum(n)
nodetype n;

DBGetGPortNUM returns the graph port number of node *n*. Note that this number is only meaningful if *n* is a graph port, i.e., its class is either "graph_inport," "graph_outport," or "graph_biport."

boolean DBIsInBounds(x, y)
int x, y;

DBIsInBounds returns TRUE if location (*x*, *y*) is in the bounds of the graph grid; otherwise, FALSE. A location is in bounds if the x and y coordinates are greater than 0 and less than GRAPHMAX.

boolean DBIsOccupied(x, y)
int x, y;

IsOccupied returns TRUE if there is a node at location (*x*, *y*) in the current data base; otherwise, FALSE.

boolean DBIsValidName(name)
char *name;

DBIsValidName returns TRUE if *name* is a valid name; otherwise, FALSE. A valid name starts with an alphabetic character and all of the remaining characters are alphanumeric.

boolean DBIsUniqName(name, naflag)
char *name;
int naflag;

DBIsUniqName returns TRUE if *name* is not the name of an element of type naflag already in the current data base; otherwise, FALSE.

DBSetNodeClass(n, class)
nodetype n;
int class;

DBSetNodeClass sets the class of node *n* to *class*. This routine should be used with discretion, since changing the class of existing nodes may introduce inconsistencies in the data base.

DBClass must be one of INTERNAL, LEAF, GRAPH_INPORT, or GRAPH_OUTPORT.

int DBGetNodeClass(n)
nodetype n;

DBGetNodeClass returns the class of node *n*.

DBMoveJunction(b, j, x, y)
bustype b;
int j;
int x, y;

DBMoveJunction moves junction *j* of bus *b* to the new location *(x, y)* in the current data base.

DBGetJunctionLoc(b, j, xp, yp)
bustype b;
int j;
int *xp, *yp;

DBGetJunctionLoc copies the (x, y) coordinates of junction *j* of bus *b* into the integers pointed to by *xp* and *yp*.

DBGetJunctionCount(b)
bustype b;

DBGetJunctionCount returns the number of junctions associated with bus *b*.

int DBGetConnectionCount(b, j)
bustype b;
int j;

DBGetConnectionCount returns the number of arcs associated with junction *j* of bus *b*.

DBGetOpenLoc(xp, yp)
int *xp;
int *yp;

DBGetOpenLoc returns the coordinates (*xp*, *yp*) of an unoccupied location in the current graph. An unoccupied location is where no node or bus junction resides.

boolean DBNodeChange()

DBNodeChange returns TRUE if the list of node names for the current data base has changed since the last call to *DBNodeChange*, and FALSE otherwise. The list of node names changes when a node is added, deleted, or renamed, or when a new data base becomes current.

boolean DBBusChange()

DBBusChange returns TRUE if the list of bus names for the current data base has changed since the last call to *DBBusChange*, and FALSE otherwise. The list of bus names changes when a bus is added, deleted, or renamed, or when a new data base becomes current.

DETERMINING ATTRIBUTE VALUES

Because *ADASDB* prohibits direct manipulation of graph element structures, the following routines must be used to access to graph element attributes. All attribute values are maintained as character strings which must be interpreted according to the attribute's data type (see *DBGetAttType*).

An attribute for a graph element is specified by giving the offset of that attribute in the element's attribute list. These integer offsets are constants that are defined in the *ADAS* include file <attributes.h>.

Because attributes are specified as offsets into an attribute list, all attributes in a list can be enumerated, without having to know about each individual attribute, by simply stepping through the list one attribute at a time.

Each routine in this section returns a pointer to a buffer that is static to the routine. This buffer is overwritten each time the routine is called, so the user should make a copy of the string if it needs to be saved.

char *DBGetGraphAtt(att)
int att;

DBGetGraphAtt retrieves the value of attribute *att* of the current graph.

char *DBGetNodeAtt(n, att)
nodetype n;
int att;

DBGetNodeAtt retrieves the value of attribute *att* of node *n*.

char* DBGetArcAtt(a, att)
arctype a;
int att;

DBGetArcAtt retrieves the value of attribute *att* of arc *a*.

char *DBGetPortAtt(n, port, ioflag, att)
nodetype n;
int port
int ioflag
int att;

DBGetPortAtt retrieves the value of attribute *att* of inport or outport *port* on node *n*. It assumes *ioflag* is either IN (for inport) or OUT (for outport).

char *DBGetBusAtt(b, att)
bustype b;
int att;

DBGetBusAtt returns a pointer to the string representation of the value of attribute *att* of bus *b*.

char *DBGetPartitionAtt(p, att)
parttype p;
int att;

DBGetPartitionAtt returns a pointer to the string representation of the value of attribute *att* of partition *p*.

SETTING ATTRIBUTE VALUES

Because *ADASDB* prohibits direct manipulation of graph element structures, graph element attributes must by modified using the following routines.

Each routine checks the modification status of the attribute and if it is modifiable, assigns the indicated value to that attribute. These routines return XSTATUS if the attribute's status is not MODIFY; XFORMAT if the attribute value has a bad format (in terms of its data type. see *DBGetAttDataType*); and XRANGE if the attribute value is out of range.

DBSetGraphAtt(att, attval)
int att;
char *attval;

DBSetGraphAtt assigns *attval* to the attribute *att* of the current graph.

DBSetNodeAtt(n, att, attval)
nodetype n;
int att;
char *attval;

DBSetNodeAtt assigns *attval* to the attribute *att* of node *n*.

DBSetArcAtt(a, att, attval)
arctype a;
int att;
char *attval;

DBSetArcAtt assigns *attval* to the attribute *att* of arc *a*.

DBSetPortAtt(n, port, ioflag, att, attval)
nodetype n;
int port;
int ioflag;
int att;
char *attval;

DBSetPortAtt assigns *attval* to the attribute *att* of the inport or outport *port* of node *n*. It assumes *ioflag* is either IN (for inport) or OUT (for outport).

DBSetBusAtt(b, att, attval)
bustype b;
int att;
char *attval;

DBSetBusAtt assigns value *attval* to attribute *att* of bus *b*.

DBSetPartitionAtt(p, att, attval)
parttype p;
int att;
char *attval;

DBSetPartitionAtt assigns value *attval* to attribute *att* of partition *p*.

DETERMINING ATTRIBUTE STATUSES

A graph, node, arc, or port attribute may have one of four modification statuses: (1) MODIFY (attribute is modifiable), (2) NOMODIFY (attribute is not modifiable), (3) PROGMODIFY (attribute is modifiable by an ADAS program only), or (4) NOTUSED (attribute is not being used). An element's attribute statuses are determined by its template.

int DBGetGraphAttStat(att)
int att;

DBGetGraphAttStat returns the modification status of attribute *att* of the current graph.

int DBGetNodeAttStat(n, att)
nodetype n:
int att:

DBGetNodeAttStat returns the modification status of attribute *att* of node *n*.

int DBGetArcAttStat(a, att)
arctype a;
int att:

DBGetArcAttStat returns the modification status of attribute *att* of arc *n*.

int DBGetPortAttStat(n, port, ioflag, att)
nodetype n;
int port;
int ioflag;
int att;

DBGetPortAttStat returns the modification status of attribute *att* of inport or outport *port* on node *n*. It assumes *ioflag* is either IN (for inport) or OUT (for outport).

int DBGetBusAttStat(b, att)
bustype b;
int att;

DBGetBusAttStat returns the modification status of attribute *att* of bus *b*.

int DBGetPartitionAttStat(p, att)
parttype p;
int att;

DBGetPartitionAttStat returns the modification status of attribute *att* of partition *p*.

SETTING ATTRIBUTE STATUSES

The following routines change an attribute's status. These routines are meant for template editing and should *not* be used for other applications since changing attribute statuses can have unpredictable side-effects.

Status must be either MODIFY (the attribute is modifiable), NOMODIFY (the attribute is not modifiable), PROGMODIFY (the attribute is modifiable by an ADAS program only), or NOTUSED (the attribute is not needed).

DBSetGraphAttStat(att, status)
int att;
int status;

DBSetGraphAttStat sets the modification status of attribute *att* of the current graph to *status*.

DBSetNodeAttStat(n, att, status)
nodetype n;
int att;
int status;

DBSetNodeAttStat sets the modification status of attribute *att* of node *n* to *status*.

DBSetArcAttStat(a, att, status)
arctype a;
int att;
int status;

DBSetArcAttStat sets the modification status of attribute *att* of arc *a* to *status*.

DBSetPortAttStat(n, port, ioflag, att, status)
nodetype n;
int port;
int ioflag;
int att;
int status;

DBSetPortAttStat sets the modification status of attribute *att* of inport or outport *port* on node *n* to *status*. It assumes *ioflag* is either IN (for inport) or OUT (for outport).

int DBSetBusAttStat(b, att, status)
bustype b;
int att;
int status;

DBSetBusAttStat sets the modification status of attribute *att* of bus *b* to *status*.

int DBSetPartitionAttStat(p, att, status)
parttype p;
int att;
int status;

DBSetPartitionAttStat sets the modification status of attribute *att* of partition *p* to *status*.

DETERMINING ATTRIBUTE DATA TYPES

The attributes associated with each kind of graph element have data types. Since attributes are stored as character strings, an attribute's data type must be known to interpret the string. Attribute data types are bound in an *attribute template* that is defined in the *ADASDB* library. An attribute's data type can be one of INTEGER, LABEL, IDENTIFIER, COORDINATE, TEXT, FILENAME, or FLOAT. Once an attribute's data type is known, standard UNIX functions (e.g., *atoi*, *atof*, etc.) may be used to interpret that attribute.

int DBGetGraphAttType(att)
int att;

DBGetGraphAttType returns the data type of the graph attribute *att*.

int DBGetNodeAttType(att)
int att;

DBGetNodeAttType returns the data type of the node attribute *att*.

int DBGetArcAttType(att)
int att;

DBGetArcAttType returns the data type of the arc attribute *att*.

int DBGetPortAttType(ioflag, att)
int ioflag;
int att;

DBGetPortAttType returns the data type of the port attribute *att*. It assumes *ioflag* is either IN (for inport) or OUT (for outport).

int DBGetBusAttType(att)
int att;

DBGetBusAttType returns the data type of the bus attribute *att*.

int DBGetPartitionAttType(att)
int att;

DBGetPartitionAttType returns the data type of the partition attribute *att*.

DETERMINING ATTRIBUTE EXISTENCE

boolean DBGraphAttExists(att)
int att;

DBGraphAttExists returns TRUE if the graph attribute *att* exists in the current data base, and FALSE otherwise.

boolean DBNodeAttExists(att)
int att;

DBNodeAttExists returns TRUE if the node attribute *att* exists in the current data base, and FALSE otherwise.

boolean DBArcAttExists(att)
int att;

DBArcAttExists returns TRUE if the arc attribute *att* exists in the current data base, and FALSE otherwise.

boolean DBBusAttExists(att)
int att;

DBBusAttExists returns TRUE if the bus attribute *att* exists in the current data base, and FALSE otherwise.

boolean DBPartitionAttExists(att)
int att;

DBPartitionAttExists returns TRUE if the partition attribute *att* exists in the current data base, and FALSE otherwise.

boolean DBPortAttExists(p, att)
int p;
int att;

DBPortAttExists returns TRUE if the port attribute *att* of port type *p* (IN, OUT, or BI) exists in the current data base, and FALSE otherwise.

DETERMINING ATTRIBUTE MODIFIABILITY

boolean DBGraphAttIsMod(att)
int att;

DBGraphAttIsMod returns TRUE if graph attribute *att* is modifiable, and FALSE otherwise.

boolean DBNodeAttIsMod(att)
int att;

DBNodeAttIsMod returns TRUE if node attribute *att* is modifiable, and FALSE otherwise.

boolean DBArcAttIsMod(att)
int att;

DBNodeAttIsMod returns TRUE if node attribute *att* is modifiable, and FALSE otherwise.

boolean DBBusAttIsMod(att)
int att;

DBBusAttIsMod returns TRUE if bus attribute *att* is modifiable, and FALSE otherwise.

boolean DBPartitionAttIsMod(att)
int att;

DBPartitionAttIsMod returns TRUE if partition attribute *att* is modifiable, and FALSE otherwise.

boolean DBPortAttIsMod(p, att)
int p;
int att;

DBPortAttIsMod returns TRUE if attribute *att* of port type *p* (IN, OUT, or BI) is modifiable, and FALSE otherwise.

DETERMINING ATTRIBUTE REQUIRED STATUS

boolean DBGraphAttIsReq(att)
int att;

DBGraphAttIsReq returns TRUE if graph attribute *att* is marked as required, and FALSE otherwise.

boolean DBNodeAttIsReq(att)
int att;

DBNodeAttIsReq returns TRUE if node attribute *att* is marked as required, and FALSE otherwise.

boolean DBArcAttIsReq(att)
int att:

DBArcAttIsReq returns TRUE if arc attribute *att* is marked as required, and FALSE otherwise.

boolean DBBusAttIsReq(att)
int att;

DBBusAttIsReq returns TRUE if bus attribute *att* is marked as required, and FALSE otherwise.

boolean DBPartitionAttIsReq(att)
int att:

DBPartitionAttIsReq returns TRUE if partition attribute *att* is marked as required, and FALSE otherwise.

boolean DBPortAttIsReq(p, att)
int p;
int att;

DBPortAttIsReq returns TRUE if attribute *att* of port type *p* is marked as required, and FALSE otherwise.

DETERMINING ATTRIBUTE SAVE STATUS

boolean DBGraphAttIsSave(att)
int att;

DBGraphAttIsSave returns TRUE if graph attribute *att* is savable, FALSE otherwise.

boolean DBNodeAttIsSave(att)
int att;

DBNodeAttIsSave returns TRUE if node attribute *att* is savable, FALSE otherwise.

boolean DBArcAttIsSave(att)
int att;

DBArcAttIsSave returns TRUE if arc attribute *att* is savable, FALSE otherwise.

boolean DBBusAttIsSave(att)
int att;

DBBusAttIsSave returns TRUE if bus attribute *att* is savable, FALSE otherwise.

boolean DBPartitionAttIsSave(att)
int att;

DBPartitionAttIsSave returns TRUE if partition attribute *att* is savable. FALSE otherwise.

boolean DBPortAttIsSave(p, att)
int p;
int att;

DBPortAttIsSave returns TRUE if attribute *att* of port type *p* is savable. FALSE otherwise.

SETTING ATTRIBUTE SAVABILITY

DBMakeGraphAttSave(att)
int att;

DBMakeGraphAttSave makes graph attribute *att* savable in the current data base.

DBMakeGraphAttNotSave(att)
int att;

DBMakeGraphAttNotSave makes graph attribute *att* unsavable in the current data base.

DBMakeNodeAttSave(att)
int att;

DBMakeNodeAttSave makes node attribute *att* savable in the current data base.

DBMakeNodeAttNotSave(att)
int att;

DBMakeNodeAttNotSave makes node attribute *att* unsavable in the current data base.

DBMakeArcAttSave(att)
int att;

DBMakeArcAttSave makes arc attribute *att* savable in the current data base.

DBMakeArcAttNotSave(att)
int att;

DBMakeArcAttNotSave makes arc attribute *att* unsavable in the current data base.

DBMakeBusAttSave(att)
int att;

DBMakeBusAttSave makes bus attribute *att* savable in the current data base.

DBMakeBusAttNotSave(att)
int att;

DBMakeBusAttNotSave makes bus attribute *att* unsavable in the current data base.

DBMakePartAttSave(att)
int att;

DBMakePartAttSave makes partition attribute *att* savable in the current data base.

DBMakePartAttNotSave(att)
int att;

DBMakePartAttNotSave makes partition attribute *att* unsavable in the current data base.

DBMakePortAttSave(type, att)
int type;
int att;

DBMakePortAttSave makes attribute *att* of port type *type* savable in the current data base.

DBMakePortAttNotSave(type, att)
int type;
int att;

DBMakePortAttNotSave makes attribute *att* of port type *type* unsavable in the current data base.

DETERMINING ATTRIBUTE NAMES

Each graph, node, arc, and port attribute has a short, descriptive name that can be retrieved with the following routines. These routines all assume *att* is a valid attribute.

char *DBGetGraphAttName(att)
int att;

DBGetGraphAttName returns the name of the graph attribute *att*.

char *DBGetNodeAttName(att)
int att;

DBGetNodeAttName returns the name of the node attribute *att*.

char *DBGetArcAttName(att)
int att;

DBGetArcAttName returns the name of the arc attribute *att*.

char *DBGetPortAttName(att, ioflag)
int ioflag;
int att;

DBGetPortAttName returns the name of the port attribute *att*. It assumes *ioflag* is either IN (for inport) or OUT (for outport).

char *DBGetBusAttName(att)
int att;

DBGetBusAttName returns the name of the bus attribute *att*.

char *DBGetPartitionAttName(att)
int att;

DBGetPartitionAttName returns the name of the partition attribute *att*.

RETRIEVING GRAPH ELEMENT TEMPLATES

Templates are kept in files separate from the graph data bases. A template is retrieved by specifying a template file search path. This search path is a string consisting of the template file names, separated by commas, that are to be searched for templates.

Template files are searched in the order they appear in the path, and the search terminates either when the template is found or all files in the path have been searched.

The template retrieval routines returns XACCESS if a file in the template search path could not be opened for reading, XFORMAT if a template file has a bad format, XEOF if end-of-file was reached prematurely while reading a template file, XMEMORY if there is not enough memory to perform the search, and XNOTFOUND if the template was not found in any of the files in the search path.

DBSetTSPath(path)
char *path;

DBSetTSPath sets the template search path to *path*.

Returns XFORMAT if *path* has an improper format or is longer than MAXPATH characters, or XMEMORY if it couldn't allocate space for the path string.

char *DBGetTSPath(n)
int n;

DBGetTSPath returns the nth component of the template search path, or NULL if there is no nth component.

DBGetNodeTemplate(name, tp)
char *name;
nodetype *tp;

DBGetNodeTemplate retrieves the node template *name* from one of the template files specified in the template search path. It sets the pointer *tp* to point to the template.

DBGetArcTemplate(name, tp)
char *name;
arctype *tp;

DBGetArcTemplate retrieves the node template *name* from one of the template files specified in the template search path. It sets the pointer *tp* to point to the template.

DBGetBusTemplate(name, tp)
char *name;
bustype *tp;

DBGetBusTemplate retrieves the bus template *name* from one of the template files specified in the template search path. It sets the pointer, pointed to by *tp*, to point to the bus template.

DBGetPartitionTemplate(tp)
parttype *tp;

DBGetPartitionTemplate gets the partition template from the master template file. Unlike other graph elements, all partitions are created from a single partition template. The pointer *tp* is set to point to the partition template.

TABLE OF ADASDB ERROR CODES

The following constants represent the possible values of the external error flag *iberrno*. The general meaning of each error code is provided here. For more specific information, see the error code information in the description for the individual routines.

XINIT	initialization error
XACCESS	can't access external data base file
XEOF	premature EOF when reading external data base file
XFORMAT	external data base format is bad, etc.
XMEMORY	ran out of memory (malloc failed)
XRANGE	attribute value is out of range
XLEVEL	graph level too deep or shallow
XLOCATION	grid location out of bounds, etc.
XNAME	bad node or arc name
XNOTFOUND	template, node, etc. not found
XPORT	invalid port
XSTATUS	invalid node class or attribute status
XEXPAND	node subgraph is (not) already expanded

A SAMPLE PROGRAM

This toy program is designed to demonstrate some basic usage of the *ADASDB* routines. It reads in a data base, fiddles with the node utilization attributes, tries to expand a node, then exits.

```
#include <adas.h>
#include <adasdb.h>
#include <attributes.h>

main()
{
   int gd;              * graph descriptor */
   nodetype n;          /* a node in the current data base */
   char *DBGetNodeName(); /* ADASDB routine for getting node name */


    /* Read in the graph data base 'foo.swg'.
     */
   if ((gd = DBGraphInit("foo.swg") < 0)
   {
      switch (dberrno)
      {
      case XINIT:
         printf("too many graphs initialized already\n");
         break;

         /* etc... */
      }

      exit(1);
   }

   DBUseGraph(gd);



    /* Set utilization of all nodes in the current data base.
     * For each node in the current data base,
     * print out old utilization value
     * then set and print new value.
     */
   DBSetFirstNode();
   while (DBMoreNodes())
   {
      DBGetNextNode(&n);

      printf("utilization of node %s was %s,",
        DBGetNodeName(n), DBGetNodeAtt(n, NUTIL));

      if (DBSetNodeAtt(n,NUTIL,"0.667") != 0)
        printf("Oops! can't set node utilization.\n");
```

```
    else printf("but now it is %s\n", DBGetNodeAtt(n, NUTIL));
}



/* Find a node called 'foonode' in the current data base.
 * make sure it is an internal node, then expand it.
 * Change the current data base to the expanded subgraph.
 * then come back.
 */
if (DBGetNodeByName("foonode", &n) != 0)
    printf("can't find node 'foonode'\n");

else if (DBGetNodeClass(n) != INTERNAL)
    printf("node 'foonode' is not an internal node\n");

else if (DBSubgraphInit(n,"foo.swt") != 0)
    printf("can't expand subgraph of node 'foonode'\n");

else
{
    (void)DBPushDb(n);
    (void)DBPopDb();
}


/* Clean up after ourselves before exiting.
 */
DBTermGraph();
exit(0);
}
```

PART III

ADAS Command Interpreter Routines

INTRODUCTION

The ADAS command interpreter (libci) is responsible for building and displaying menus, displaying prompts, and interpreting input for ADAS client programs. A client program gives the command interpreter a top-level command list; the program gives a list of pointers to functions to be executed when the corresponding commands are chosen. For menus other than the top one, client programs build the menus and prompt and pass them to the command interpreter, which returns the user input.

ciExecCmd()

ciExecCmd is the top-level routine called by clients using the ADAS command interpreter. It starts up the command interpreter with the client program's top-level menu and executes commands from that menu.

ciInit(menu, list)
menutype menu[];
functype list[];

ciInit initializes the command interpreter's command menu and command function list for the current program. It does this by setting pointers to the menu, *menu* and the function list, *list*. It also sets the count of the number of commands in the menu.

Ci(canon_form, menu, validate, prompt, help, tokenp)
int canon_form;
menutype *menu;
int (*validate)();
char *prompt, *help;
tokentype *tokenp;

Ci displays the menu pointed to by *menu*, sets *prompt* and *help* as the prompt and help messages, respectively, and gets the next token in the input. The expected canonical form and the validation function for the token are *canon_form* and *validate*. *Ci* sets *tokenp* to point to the new token. It returns ESCAPE if the token is a special token and CANCEL if the command is canceled either by the user or because of too many tries to get an acceptable token.

int ciValidRepeat(token_val,tokenp)
char *token_val;
tokentype *tokenp;

ciValidRepeat returns 0 if token_val is a valid token value representing a number of repetitions for a command (any integer), and -1 if it is not. This routine checks only the syntax of the repetition count, not the range. If the count is valid, *token_val* is copied into the token pointed to by *tokenp*.

CANONICAL FORMS

Each token that the command interpreter processes has associated with it a *canonical form* which

the client program expects it to match. There are nine different canonical forms: INTEGER, REAL, COORDINATE, IDENTIFIER, TEXT, NODE, ARC, PUNCTUATION, and NEWLINE.

int ciConvertToCanon(token_val, token_type, canon_form, tokenp)
char token_val[];
int token_type;
int canon_form;
tokentype *tokenp;

ciConvertToCanon uses the canonical form, *canon_form* and the token type, *token_type* to convert the token value *token_val* into the appropriate form and assigns it to the token pointed to by *tokenp*. It returns 0 if *token_val* was successfully converted and -1 if the conversion could not be accomplished.

boolean ciPickFromGraph()

ciPickFromGraph returns TRUE if the last token converted by *ciConvertToCanon* was the result of a graph pick and FALSE otherwise.

CONTEXT

These routines provide context information in a message included in the command interpreter prompt. This message reflects the command sequence which put the command interpreter into its current state, starting from the main menu.

ciEnableContextString(on)
boolean on;

ciEnableContextString enables the command interpreter's context string mechanism if *on* is TRUE, and disables it otherwise.

int ciContextLevel()

ciContextLevel returns the value of the static variable *context_level*.

ciInitContext()

ciInitContext initializes the value of the static variable *context_level* to 0 to indicate that the main menu is the current menu.

ciSetContextLevel()

ciSetContextLevel increments the static variable *context_level* to record the descent into a submenu of the current menu.

ciMakeContextString(canon_form, tokenp)
int canon_form;
tokentype *tokenp;

ciMakeContextString builds the command context string up to two levels deep for the command specified in the token pointed to by *tokenp* and also having canonical form, *canon_form*.

boolean CommandOK(command)

char *command;

CommandOK determines if *command* is "add," "delete," "edit," "environ," "move," or "simulate," and is, therefore, one of the main menu commands for which context information should be displayed. It returns TRUE if *command* is in this set and FALSE otherwise.

ciPrintContext ()

ciPrintContext displays command context information on the terminal screen.

ConvertToUpperCase(tempstring)
char *tempstring;

ConvertToUpperCase converts *tempstring* to all upper case.

boolean CFormOK(canon_form)
int canon_form;

CFormOK checks that the canonical form *canon_form* is appropriate for use as a context message, i.e., is either IDENTIFIER or TEXT. It returns TRUE if the form is acceptable and FALSE otherwise.

ciContextSave()

ciContextSave saves both the current context string and the context level to be restored later with *ciContextRestore*. This is done when a **help** command is issued, allowing for a return to the current menu.

ciContextRestore()

ciContextRestore restores the context information saved by *ciContextRestore*.

ciSetHelpContext()

ciSetHelpContext sets the context message and level for the **help** command.

ERROR HANDLING

ciSetErrorFlag(value)
boolean value;

ciSetErrorFlag sets the error flag indicating incorrect input to *value*, which should be either TRUE or FALSE.

boolean ciErrorFlag()

ciErrorFlag returns the value of the error flag used, indicating the incorrect input.

HANDLING INPUT

int ciGetTokenDirect(token_buf)
char token_buf[];

ciGetTokenDirect gets a token string directly from input and puts it into *token_buf*. If the string specifies a shell escape, *ciGetTokenDirect* calls *DoShell* with that token and gets the next one. It returns the type of the token.

char *ciExtractToken(stringp)
char **stringp;

ciExtractToken finds the next token in the string pointed to by *stringp* and returns a pointer to it. In this context, a token is a string of one or more non-blank characters. The pointer pointed to by *stringp* is updated to point to the next token in the string. If no token was found, *ciExtractToken* returns NULL.

ciFillQueue()

ciFillQueue gets tokens and fills the token queue. If a token specifies a shell escape, *ciFillQueue* calls *DoShell* to handle it, otherwise it adds the token to the token queue.

int ciGetToken(using_queue, token_val)
int using_queue;
char token_val[];

ciGetToken gets a token string and puts it into *token_val*. If *using_queue* is TRUE, the token is retrieved from the token queue with *ciQueueGet*, otherwise it comes from *ciGetTokenDirect*. It returns (1) the token's lexical type upon normal completion, (2) ESCAPE when the token is the escape token, and (3) CANCEL when the token is the cancel one.

char *ciGetInput()

ciGetInput reads the next line from the script file if one is being used. Otherwise, it polls the keyboard and data tablet for input. If input is received from the tablet, *ciGetInput* converts the coordinates into a token using *ciTransformPick*. It returns a pointer to a static buffer containing the input.

char *ciTransformPick(ndcx, ndcy)
double ndcx, ndcy;

ciTransformPick transforms the normalized device coordinates (*ndcx, ndcy*) into a string and returns a pointer to the string or NULL if the transformation is unsuccessful. If the location is within a menu, the string is the appropriate menu entry. Otherwise, it is the string representation of the location in graph (ADAS grid) coordinates.

boolean ciPickInMenu(ndcx, ndcy, whichmenu)
double ndcx, ndcy;
int whichmenu;

ciPickInMenu returns TRUE if the pick at (*ndcx, ndcy*) is inside the menu specified by *whichmenu*, and FALSE otherwise.

char *ciGetMenuItem(ndcx, ndcy, screen_loc)
double ndcx, ndcy;
int screen_loc;

ciGetMenuItem returns a pointer to a string containing the menu entry located at (*ndcx, ndcy*) and contained in the menu specified by *screen_loc*. The value of *screen_loc* is 0 for the auxiliary menu and 1 for the main menu.

boolean ciInGraph(ndcx, ndcy)
double ndcx, ndcy;

ciInGraph returns TRUE if the location (*ndcx, ndcy*) is inside the graph window.

char *ciGetGraphItem(ndcx, ndcy)
double ndcx, ndcy;

ciGetGraphItem returns a pointer to a string with the string representation of the graph (ADAS grid) coordinates corresponding to NDC location (*ndcx, ndcy*).

MACROS

A macro is simply a user-defined, named sequence of commands and their parameters. Macros can be written so parameter values are given at the time the macro is used by substituting a place holder ('$') for each parameter which will be filled in later.

int ciAddMacro()

ciAddMacro gets a macro definition from input, adds the macro to the command list, and loads it into the macro data structure.

int ciDeleteMacro()

ciDeleteMacro allows the user to select the macro to be deleted from a menu, then it deletes the macro from the command set and from the macro data structure.

ciListMacroDefs()

ciListMacroDefs displays each currently defined macro and its definition on the terminal screen.

ciDoMacro(index)
int index;

DoMacro prepares for the execution of the macro in position *index* in the macro list. If the macro definition has no place holder for parameters, it is pushed onto the token queue. Otherwise, it is pushed onto an auxiliary queue and a flag is set indicating its presence there.

ciGetMacro()

ciGetMacro gets a macro definition from input and puts it into the token queue.

boolean ciIsACommand(token_val)
char *token_val;

ciIsACommand returns TRUE if *token_val* is the name of a command from the command menu and FALSE otherwise.

ciClearMacro(index)
int index;

ciClearMacro resets the individual macro data structure with index *index* to empty.

ciCompactMacroList(index)
int index;

ciCompactMacroList compacts the macro data structure to eliminate macro number *index* by shifting all macros following *index* up one place in the list.

ciCompactMacroMenu(index, menu)
int index;
menutype *menu;

ciCompactMacroMenu compacts the macro menu pointed to by *menu* to eliminate macro number *index* by shifting all macros following *index* up one place in the menu.

boolean ciEmbeddedPlaceHolder(macroindex)
int macroindex;

ciEmbeddedPlaceHolder returns TRUE if there is an embedded parameter placeholder in the definition of macro number *macroindex*, and FALSE otherwise. An embedded placeholder is one that is followed in the macro definition by an argument that is not a placeholder.

MENUS

An ADAS menu consists of a list of strings -- the menu entries -- each with an associated color which is used when the menu item is displayed on the graphics screen. A menu contains an empty string ("") as its last entry, marking the end of the menu.

ciShowMenu()

ciShowMenu erases the existing menu from the graphics display and draws the current page of the current menu on the terminal screen and the graphics display.

ciShowAuxMenu()

ciShowAuxMenu draws the auxiliary menu on the terminal screen.

ciEraseMenu()

ciEraseMenu sets the current menu pointer to NULL.

int ciGetMenuIndex(s, menu)
char *s;
menutype *menu;

ciGetMenuIndex returns the index of the entry *s* in the menu pointed to by *menu*. It returns -1 if *s* is not in *menu*, or -2 if *s* is a prefix of more than one entry.

char *ciGetMenuEntry(index, menu)
int index;
menutype *menu;

ciGetMenuEntry returns a pointer to entry number *index* in *menu*. It assumes *menu* isn't NULL and *index* is the index of a valid entry. *ciGetMenuEntry* returns a static buffer that is overwritten on subsequent calls.

int ciAddMacroToMenu(item, menu)
char *item;
menutype *menu;

ciAddMacroToMenu adds the entry *item* to the end of the menu pointed to by *menu* in the color YELLOW. Actually, the new entry is added in the next-to-last position, just before the "empty" entry that signals the end of the menu.

ciSetMenu(menu)
menutype *menu;

ciSetMenu sets the current menu pointer to the menu pointed to by *menu*. The current page is set to the first page (page 0).

ciUnsetMenu()

ciUnsetMenu unsets the current menu. The current menu is set to a new, blank menu with no entries.

ciPageMenu()

ciPageMenu turns to the next page of the current menu. If the current page is the last page in the menu or the next page is blank, it wraps around to the first page. *ciPageMenu* assumes menu pages are numbered from 0 to MENU_PAGES - 1.

ciDrawMenu(flag)
int flag;

ciDrawMenu displays the items of the current menu in the menu area of the graphics display. If *flag* is 0 the menu is erased; otherwise it is drawn.

ciDrawAuxMenu(flag)
int flag;

ciDrawAuxMenu displays the items of the auxiliary menu in the auxiliary menu area of the graphics display. If *flag* is 0, menu is erased, otherwise it is drawn.

char *ciGetCurrMenuItem(slot, screen_loc)
int slot;
int screen_loc;

ciGetCurrMenuItem returns the contents of slot number *slot* in the menu specified by *screen_loc*. If *screen_loc* is 1, a pointer to the entry in slot *slot* of the current menu is returned. Otherwise, it returns a pointer to a string containing only the first character of auxiliary menu entry number *slot*. This is the special character denoting an auxiliary menu entry.

ciMakeMenu(menu)
menutype **menu;

ciMakeMenu creates a new menu of size MENU_MAX and initializes all its entries to the empty string. The pointer pointed to by *menu* is set to point to the new menu.

ciMenuAdd(menu, item, color)
menutype **menu;
char *item;
int color;

ciMenuAdd adds the entry *item* to the menu pointed to by the pointer pointed to by *menu*. The new entry will be displayed in the color *color*. The new entry is checked to ensure that there is not an existing entry of the same name. If not, the new entry is placed in the first available slot in the menu.

ciFreeMenu(menu)
menutype *menu:

ciFreeMenu frees the memory occupied by *menu*.

ciMenuDel(menu, item)
menutype *menu:
char *item;

ciMenuDel deletes the entry *item* from the menu pointed to by *menu*. All entries coming after *item* in the menu are moved up one slot.

boolean ciInMenu(menu, item)
menutype *menu;
char *item;

ciInMenu returns TRUE if *item* is an entry in the menu pointed to by *menu*, otherwise returns FALSE.

boolean ciInCurrMenu(item)
char *item;

ciInCurrMenu returns TRUE if *item* is an entry in the current menu, otherwise returns FALSE.

int ciMenuSize(menu)
menutype *menu;

ciMenuSize returns the number of items in the menu pointed to by *menu*. It assumes the last item in the menu is followed by a NULL entry.

int ciPageSize()

ciPageSize returns the number of items in the current page of the current menu.

int ciMapSlotV2R(virt)
int virt;

ciMapSlotV2R maps a virtual menu slot *virt* of the menu structure to an actual slot in the menu area of the graphics display and returns the actual slot number.

int ciMapSlotR2V(real_slot)
int real_slot;

ciMapSlotR2V maps a real menu slot *real_slot* of the menu area of the graphics display to a virtual slot in the menu structure and returns the virtual slot number.

SetTopMenu()

SetTopMenu sets a flag that indicates the current menu is the top command menu. This flag is used for handling command repetition.

UnsetTopMenu()

UnsetTopMenu unsets the flag indicating the current menu is the top command menu.

boolean IsTopMenu()

IsTopMenu returns the current value of the top menu flag whose value is modified with *SetTopMenu* and *UnsetTopMenu*.

ciSortMenu(menu, dontsort)
menutype *menu;
int dontsort;

ciSortMenu alphabetically sorts part of the menu pointed to by *menu*, leaving the first *dontsort* items in their original positions.

PROMPT AND HELP MESSAGES

ciSetMessages(prompt, help)
char *prompt;
char *help;

ciSetMessages sets the command interpreter's prompt message to *prompt* and its help message to *help*. Pointers are set to point to the two strings; they aren't copied. Therefore, if one of the strings passed to *ciSetMessages* is changed, the corresponding command interpreter message will change.

ciPrompt()

ciPrompt displays the current command interpreter prompt message on the terminal screen.

ciHelpMsg()

ciHelpMsg displays the current command interpreter help message on the terminal screen.

QUEUE MANAGEMENT

The command interpreter maintains both the input token queue and an auxiliary queue. Each queue is an array of pointers to characters which point to the entries in the queue. The auxiliary queue is used for macros with embedded parameter placeholders. When a macro name is encountered in the input queue, its definition is added to the auxiliary queue. Then, tokens are taken from the auxiliary queue, with each placeholder being replaced by the matching token from the input queue until the auxiliary queue is empty.

ciQueueAdd(token)
char *token;

ciQueueAdd adds the token *token* to the tail of the token queue. It returns 0 upon normal

completion, -1 if the token queue is already full, and -2 on a memory allocation failure. *ciQueueAdd* assumes that the length of new token is less than TOKENLEN and that queue's tail pointer points to an empty slot.

ciAuxQueueAdd(token)
char *token;

ciAuxQueueAdd adds the token *token* to the tail of the auxiliary token queue. It returns 0 upon normal completion and -2 on a memory allocation failure. *ciAuxQueueAdd* assumes that the length of the new token is less than TOKENLEN and that the auxiliary queue's tail pointer points to an empty slot.

ciQueuePush(token)
char *token;

ciQueuePush adds a token to the *head* of the token queue. It returns 0 upon normal completion, -1 if the token queue is already full, and -2 on a memory allocation failure. It assumes that the length of the new token is less than TOKENLEN and that the queue's tail pointer points to an empty slot.

int ciQueueGet(token_val)
char token_val[];

ciQueueGet gets the next token from the input queue or the auxiliary queue into *token_val* and returns the token's lexical type. If the value of the flag *use_aux_queue* (which is set with *ciUseAuxQueue*) is TRUE, it takes the first token from the auxiliary queue. If this token is a place holder and the token at the front of the input queue is a DELIM_CHAR, *ciQueueGet* leaves the auxiliary queue unchanged and gets a token from the input queue instead. *ciQueueGet* assumes that *token_val* is at least TOKENLEN + 1 in length, that the token at the head of the input queue is non-null and is no longer than TOKENLEN, and that the input queue is not empty.

ciQueueFlush()

ciQueueFlush flushes the token queue and, if *use_aux_queue* is TRUE, flushes the auxiliary queue as well. When a queue is flushed, all the tokens in it are removed and its head and tail pointers are reset to zero.

int ciQueueSize()

ciQueueSize returns the number of tokens in the token queue.

boolean ciQueueEmpty()

ciQueueEmpty returns FALSE if the token queue has any tokens or if it is empty and the auxiliary queue has a token other than a place holder. It returns TRUE otherwise.

boolean ciQueueFull()

ciQueueFull returns TRUE if the token queue is full, and FALSE otherwise.

ciUseAuxQueue()

ciUseAuxQueue sets the flag *use_aux_queue*, indicating that the auxiliary queue is to be used.

SCRIPT FILES

The command interpreter can read commands from text files called *script* files, just as it does from the keyboard. Because a command in one script file can read commands from another script file, a stack of script files is maintained.

char *ciReadScript()

ciReadScript gets a line of input from the script file at the top of the stack, and returns a pointer to a static buffer containing the input line, or NULL if end of file was reached. It assumes that the command interpreter is currently in script mode.

ciUseScriptFile(file)
char *file;

ciUseScriptFile opens the file named *file* and pushes the file pointer onto the stack of script files. It also sets the flag indicating that the command interpreter is in script mode. *ciUseScriptFile* returns -1 if it can not open the script file.

boolean ciInScriptMode()

ciInScriptMode returns TRUE if the command interpreter is in script mode, FALSE otherwise.

ciSetScriptFlag(value)
boolean value;

ciSetScriptFlag sets the flag indicating whether the command interpreter is in script mode to *value*, which should be either TRUE or FALSE.

PushItem(fileptr)
FILE *fileptr;

PushItem pushes file pointer *fileptr* onto the dynamically-allocated stack of script file pointers.

PopItem()

PopItem pops the top file pointer from the script file stack and closes the associated file.

ciScriptCleanup()

ciScriptCleanup closes all script files, frees the memory associated with the script file stack, issues an error message, sets the script error flag, and flushes the input queue.

TYPES AND VALIDATION

These routines are used to determine the lexical types of tokens and to validate them. Possible lexical types are INTEGER, FLOAT, COORDINATE, IDENTIFIER, PUNCTUATION, NEWLINE, TEXT, and DELIM_CHAR.

int ciTokenType(token)
char *token;

ciTokenType returns the lexical type of the token *token*, or -1 if its type cannot be determined.

boolean IsIntType(s)
register char *s;

IsIntType returns TRUE if the string *s* has lexical type INTEGER and FALSE otherwise. A valid INTEGER consists of an optional sign [+-] followed by a string of digits [0..9]. It assumes *s* is not NULL and does not handle leading or trailing blanks.

boolean IsRealType(s)
register char *s;

IsRealType returns TRUE if the string *s* has lexical type FLOAT; otherwise, FALSE. A valid FLOAT consists of an optional sign, a string of digits, a decimal point, and another string of digits. Either, but not both, of the strings of digits may be empty. *IsRealType* assumes *s* is not NULL and does not handle leading or trailing blanks.

boolean IsCoordType(s)
register char *s;

IsCoordType returns TRUE if the string *s* has lexical type COORDINATE and FALSE otherwise. A valid COORDINATE consists of a string of digits, a comma, and a second string of digits. *IsCoordType* assumes *s* is not NULL and does not handle leading or trailing blanks.

boolean IsIdentType(s)
register char *s;

IsIdentType returns TRUE if the string *s* has lexical type IDENTIFIER and FALSE otherwise. A valid IDENTIFIER is of the form *[a-zA-Z][a-zA-Z0-9._]**. *IsIdentType* assumes *s* is not NULL and does not handle leading or trailing blanks.

boolean IsPunctType(s)
register char *s;

IsPunctType returns TRUE if the string *s* has lexical type PUNCTUATION and FALSE otherwise. A valid PUNCTUATION string consists of a single punctuation mark.

boolean IsNewLineType(s)
register char *s;

IsNewLineType returns TRUE if the string *s* has lexical type NEWLINE, i.e. consists of a single newline ("\n"), and FALSE otherwise.

boolean IsTextType(s)
register char *s;

IsTextType returns TRUE if the string *s* has lexical type TEXT, i.e., consists only of printable characters or a single newline, and FALSE otherwise.

boolean IsDelimType(s)
register char *s;

IsDelimType returns TRUE if the string *s* has lexical type DELIM_CHAR, i.e., its first character is DELIM_CHAR ("\r"), and FALSE otherwise.

int ciValidateToken(tokenp, validate, canon_form)
tokentype *tokenp;
int (*validate)();
int canon_form;

ciValidateToken attempts to validate the token, *token*, based on the canonical form *canon_form* and using the function *validate*. It returns -1 if the token is not valid and 0 otherwise.

MISCELLANEOUS ROUTINES

DoShell(string)
char *string;

DoShell performs a shell escape from the client program. Under VMS, a subprocess is spawned. Under UNIX, the appropriate shell, as determined by the SHELL environment variable (**/bin/sh** is the default), is invoked. If *string* is not empty, it is passed as a command to the subprocess (VMS) or shell (UNIX).

char *stralloc(s)
char *s;

stralloc allocates a new string of the appropriate length and copies *s* into it. It returns a pointer to the new string, or NULL if there is a memory allocation failure.

coordtype *ci_atoc(s)
char *s;

ci_atoc converts a string representation *s* of a coordinate to coordtype. The string representation must be of the form "x,y". *ci_atoc* returns a pointer to a static coordinate structure containing the converted coordinate if the conversion is successful and NULL if it is not.

boolean ciIsNewline(str)
char *str;

ciIsNewline returns TRUE if the first character of string *str* is the newline ("\n") character and FALSE otherwise.

PART IV

ADAS Editor Routines

INTRODUCTION

The ADAS Editor (libedit) contains the routines that implement the ADAS editor. This editor is used to modify the values of attributes associated with ADAS data base components. The editor does type checking only on the values entered by the user. Any other checking is left to the client program, since different clients may use a given attribute for different purposes.

EDITING GRAPH COMPONENTS

Several global variables are used in the ADAS editor routines. The variables *ned*, *aed*, and *ped* are the editor's current node, arc, and port. Routines accept a parameter that indicates which type of graph element is to be edited (possible values are GRAPH, NODE, ARC, IN, and OUT), then they use the appropriate global variable. The boolean variable *tmpflag* is TRUE if values in the template data base are being edited and FALSE if the graph data base is being used. The boolean *statflag* is TRUE if attribute modification statuses are being edited and FALSE if their values are being edited.

Special editor control characters are used to traverse the attribute list currently being edited. The special edit characters are: newline ("\n": go to next attribute), minus sign ("-": go to previous attribute), plus sign ("+": go to attribute list for next port), tilde ("~": end the editing session), sharp sign ("#": set this string attribute value to the null string), caret ("^": go to the first attribute in the list), and ampersand ("&": display the edit help message). Many routines return a code specifying which special character, if any, has been entered. The valid editor return codes are ED_CURRENT, ED_NEXT, ED_PREVIOUS, ED_FIRST, ED_DONE, ED_NEXT_PORT, and ED_USE_PREV.

Edit()

Edit is the top-level routine for the **edit** command. It accepts the top-level edit menu selection and invokes the appropriate editing procedure.

int EditGraph()

EditGraph allows the user to edit graph attributes for the current graph.

EditNode(n)
nodetype n;

EditNode allows the user to edit the attributes of a node. If *n* is NULL, the node is selected by the user; otherwise, the node *n* is edited. If *n* is not NULL, it is assumed to be a valid node. *EditNode* sets the global variable *ned* to *n*.

EditBus(b)
bustype b;

EditBus allows the user to edit the attributes of a bus. If *b* is NULL, the bus is selected by the user; otherwise, the bus *b* is edited. If *b* is not NULL, it is assumed to be a valid bus. *EditBus* sets the global variable *bed* to *b*.

EditArc(a)
arctype a;

EditArc allows the user to edit the attributes of an arc. If *a* is NULL, the arc is selected by the user; otherwise, the arc *a* is edited. If *a* is not NULL, it is assumed to be a valid arc. *EditArc* sets the global variable *aed* to *a*.

EditNodeTmp()

EditNodeTmp allows the user to edit a node template. The user is prompted for the template name.

EditArcTmp()

EditArcTmp allows the user to edit an arc template. The user is prompted for the arc template name.

EditBusTmp()

EditBusTmp allows the user to edit an bus template. The user is prompted for the bus template name.

EditRepeat(naflag)
int naflag;

EditRepeat allows the user to re-edit the attributes for the selected graph element. When the user exits the editor, he is given the option of re-editing the item. The graph element type is indicated by *naflag*.

DoEdit(naflag)
int naflag;

DoEdit steps through the list of attributes for the graph element type, indicated by *naflag*, until done. The edit is completed when the entire list has been traversed, CANCEL has been received, or the user has entered the ED_DONE character.

int NextPort(n, naflag)
int n;
int naflag;

NextPort advances the editor to the attribute list for the next port when editing a node. The integer *n* is the current (old) attribute number and *naflag* indicates the graph element type being edited. This routine is called when ED_NEXT_PORT is received. *NextPort* returns the index of the next port or the last attribute. It assumes that *naflag* is GRAPH, NODE, or ARC.

EdHelp()

EdHelp displays information concerning editing attribute values on the terminal screen.

boolean SpecialChar(str)
char *str;

SpecialChar returns TRUE if *str* points to a special edit character, and FALSE otherwise.

boolean NodeSpecialChar(str)
char *str;

NodeSpecialChar returns TRUE if *str* is the name of a node or points to a special edit character, and FALSE otherwise.

boolean BusSpecialChar(str)
char *str;

BusSpecialChar returns TRUE if *str* is the name of a bus or points to a special edit character, and FALSE otherwise.

boolean NodeBusSpecialChar(str)
char *str;

NodeBusSpecialChar returns TRUE if *str* is the name of a node or bus or points to a special edit character, and FALSE otherwise.

EDIT SELECT

EditSelect()

EditSelect allows the user to edit a selected attribute for a chosen element type. The user selects the element type from a menu, then specifies whether to edit all items of that type or to select the particular items to edit by name.

EditSelectTmp(napflag, select_all)
int napflag;
int select_all;

EditSelectTmp sets the global template flag *tmpflag* in order to use the template data base, then calls procedure *EditSelectNAP* -- the same routine used for non-template select editing.

EditSelectNAP(napflag, select_all)
int napflag;
int select_all;

EditSelectNAP carries out selective editing for either nodes, arcs, or node ports. The value of *napflag* determines which object type is being edited. Valid values for *napflag* are NODE, ARC, IN, and OUT. If *select_all* is TRUE, the selected attribute is edited for all objects of this type; otherwise, the individual objects are selected by name. *statflag* is checked to see if attribute modification statuses are being edited, and *tmpflag* is checked to determine if the template data base is being used.

EditSelectAll(napflag, att_name)
int napflag;
char *att_name;

EditSelectAll allows the user to edit the attribute with name *att_name* for all instances of the type of graph element type indicated by *napflag*.

EditSelectAtt(napflag, att_name, name)
int napflag;
char *att_name;
char *name;

EditSelectAtt prompts the user for the name *name* of the graph element of type *napflag* for which he wishes to edit the attribute *att_name*.

EditSelectByName(napflag, att_name)
int napflag;
char *att_name;

EditSelectByName displays the full list of graph items of the type indicated by *napflag*. The user selects one item, has a chance to change the value of the attribute *att_name*, and then may select a new item to edit.

EDIT GLOBAL

EditGlobal()

EditGlobal allows the user to select the graph element type for the global edit. Global editing allows the user to set the value of one attribute to a particular value for all existing graph elements derived from a given template; additionally, it optionally saves the new value in the template.

EditGlobalNAP(napflag)
int napflag;

EditGlobalNAP displays the attribute list for the previously selected graph type indicated by *napflag*. The user selects the attribute for the global edit. The global variable *statflag* is used to determine which attributes to display. Some node and arc attributes do not have modifiable statuses.

EditGlobalAtt(napflag, att_name)
int napflag;
char *att_name;

EditGlobalAtt allows the user to do global editing of the attribute *att_name* for the graph element type indicated by *napflag*. First, the user selects the template for the global edit and enters the new attribute value; he is given the option of saving the change in the template itself, then *DoGlobalEdit* is called to set the attribute value for all graph elements derived from the chosen template.

DoGlobalEdit(napflag, name, att_name)
int napflag;
char *name;
char *att_name;

DoGlobalEdit goes through the data base and sets the value of attribute *att_name* for all items of the type indicated by *napflag*, derived from template *name*, to agree with the template's value of that attribute.

CHANGING ATTRIBUTE VALUES

EditAtt(type, att)
int type;
int att;

EditAtt allows the user to edit attribute number *att* of the current element of type *type* (GRAPH, NODE, ARC, IN, or OUT). The current value of the attribute is displayed, and the user is given a chance to enter a new value if the attribute is modifiable. *EditAtt* returns one of the edit return codes.

EdSetAtt(type, att, val)
int type;
int att;
char *val;

EdSetAtt sets the value of attribute number *att* of the current element of type *type* to *val*. *EdSetAtt* assumes that *att* is a valid attribute.

char *EdGetAtt(type, att)
int type;
int att;

EdGetAtt returns a pointer to the string representation of the value of attribute number *att* of element type *type*. If the type is not GRAPH, one or more of the global variables *ned*, *aed*, and *ped* will determine the correct element. *EdGetAtt* assumes that *att* is a valid attribute.

char *EdGetAttName(type, att)
int type;
int att;

EdGetAttName returns the name of attribute *att* of graph element type *type*.

int GetAttIndex(type,attname)
int type;
char *attname;

GetAttIndex returns the index of the attribute of graph element type *type* with name *attname*. It assumes *type* is one of GRAPH, NODE, ARC, IN, or OUT.

boolean EdAttMap(n, naflag, att, type, port)
int n;
int naflag;
int *att;
int *type;
int *port;

EdAttMap maps the attribute number *n* for the graph element type given by *naflag* (GRAPH, NODE, ARC, PARTITION, or BUS) to the appropriate attribute index for that type. The integers pointed to by *att*, *type*, and *port* are set to the adjusted attribute index, the graph element type (GRAPH, NODE, ARC, PARTITION, BUS, IN, OUT, or BI), and the port number.

respectively. The value of *att* is adjusted for special attributes that do not appear in the attribute list, e.g. *node_name*. *EdAttMap* returns TRUE if there are no more attributes to edit, i.e., *n* exceeds the attribute index range for the graph element type.

char *AttPrompt(label, val)
char *label;
char *val;

AttPrompt builds and returns a pointer to the prompt for the attribute with name *label* and current value *val*. The prompt is built in a static buffer which is overwritten by the next call to *AttPrompt*.

boolean DisplayedAtt(type,att)
int type;
int att;

DisplayedAtt determines whether attribute *att* of graph element type *type* affects the appearance of the graphics display, e.g., *node_color*. It returns TRUE if *att* affects the screen and FALSE otherwise.

MODIFICATION STATUSES

A *modification status* is associated with each attribute, in addition to the attribute's value. An attribute's modification status determines the means by which its value can be changed. Valid modification statuses are **M** (MODIFY), **P** (PROGMODIFY), **N** (NOMODIFY), and **U** (NOTUSED). An attribute with status **M** can be modified by the user with the editor. Status **P** indicates that an attribute can be modified only by an ADAS program, not directly by the user. An **N** modification status means the attribute is unmodifiable. An attribute with **U** status is marked as not used.

EditStats()

EditStats allows the user to edit the modification statuses of attributes. *EditStats* operates like the top-level *Edit* routine, after first setting the global flag *statflag* to indicate that statuses are being edited, rather than attribute values.

EditAttStat(type, att)
int type;
int att;

EditAttStat allows the user to edit the attribute modification status of attribute number *att* of the current element of type *type*. The current status is displayed and the user is given a chance to enter a new modification status. *EditAttStat* returns one of the edit return codes.

EdSetAttStat(type, att, s)
int type;
int att;
char *s;

EdSetAttStat sets the modification status of attribute number *att*, of the current element of type *type*, to the status indicated by the character pointed to by *s* ("M", "P", "N", or "U"). *EdSetAttStat* assumes *att* is a valid attribute index.

int EdGetAttStat(type, att)
int type;
int att;

EdGetAttStat returns the modification status of attribute number *att* of the current graph element of type *type*. It assumes *att* is a valid attribute index.

EdSetAttStatInt(type, att, stat)
int type;
int att;
int stat;

EdSetAttStatInt sets the modification status of attribute number *att* of the current element of type *type* to status *stat*. *EdSetAttStatInt* does the same thing as *EdSetAttStat*, except it uses the integer modification status instead of a character code which represents the new status.

EdGetUserModStat(type,att)
int type;
int att;

EdGetUserModStat returns the modification status of attribute *att* for the element of type *type* currently being edited.

EdStatHelp()

EdStatHelp displays information concerning editing attribute modification statuses on the terminal screen.

char *AttStatPrompt(label, stat)
char *label;
int stat;

AttStatPrompt builds and returns a pointer to the prompt for the modification status of the attribute with name *label* and current modification status *stat*. The prompt is built in a static buffer which is overwritten by the next call to *AttStatPrompt*.

boolean IsValidStat(str)
char *str;

IsValidStat returns TRUE if *str* is a valid status name or points to a special edit character, and FALSE otherwise.

ATTRIBUTE SAVE STATUS

Not every ADAS data base contains the full set of attributes for all graph element types. The user can select which attributes to save to a data base file by editing attribute save status. An attribute's save status is either **S**, meaning it will be saved to the data base file, or **N**, meaning it will not. The following editor routines are used for editing attribute save status.

EditSaveStats(allflag)
boolean allflag;

EditSaveStats allows the user to edit the save status for attributes of any object type. It provides

a menu from which the user selects an object type or *allatts*. By choosing *allatts*, the user can set all attributes of an object type to be either savable or unsavable. The *allflag* parameter to *EditSaveStats* is TRUE if it is recursively called to handle an *allatts* selection.

EditSaveAllStats(element, attcount, tmpflag)
int element;
int attcount;
boolean tmpflag;

EditSaveAllStats allows the user to set the save status for all attributes of graph element type *element* to either S or N. The value of *attcount* is the total number of attributes for the element type, and *tmpflag* is TRUE if the desired element type is a template type.

EditRepeatSave(naflag, att_count)
int naflag;
int att_count;

EditRepeatSave allows the user to re-edit the save status of attributes for graph element type *naflag*. When the user exits the editor, he is given the option of re-editing the item. The total number of attributes for the graph element type is given by *att_count*.

EditSaveAtts(eflag, att_count)
int eflag;
int att_count;

EditSaveAtts allows the user to set the save status for the attributes of the element type given by *cflag*. It steps through all *att_count* attributes, and the user can enter a new save status for each.

EditSaveAtt(type, att)
int type;
int att;

EditSaveAtt allows the user to set the attribute save status for attribute *att* of element type *type*.

EditSaveTmplAtts(eflag, att_count)
int eflag;
int att_count;

EditSaveTmplAtts sets the current data base to the template data base in preparation for editing the save status for attributes of element type *eflag*. The number of attributes for the element type is given by *att_count*. *EditSaveTmplAtts* sets up the template data base, then calls *EditRepeatSave* to do the attribute save status editing.

boolean IsValidSaveStat(str)
char *str;

IsValidSaveStat returns TRUE if *str* points to a valid attribute save status or a special edit character, and FALSE otherwise.

EdSaveStatHelp()

EdStatHelp displays information concerning editing attribute save statuses on the terminal screen.

SELECTING GRAPH ELEMENTS FOR EDITING

EdSetNode(node, template, ptype, pstatus)
nodetype *node;
boolean template;
int ptype;
int pstatus;

EdSetNode displays a menu of nodes in the current graph, prompts the user for a node, and sets node to point to the selected node. If *template* is TRUE, a node template is wanted and the menu will contain node template names. The *ptype* and *pstatus* parameters determine which nodes are included. The value of *pstatus* is 0 if only nodes with unoccupied ports are to be added, 1 for only nodes with occupied ports, and 2 for all nodes. The value of *ptype* is interpreted as a bit vector flagging the types of ports to be considered in this decision. The least significant bit is set for inports, the next for outports, and the third for biports. If *pstatus* is set to 3, all nodes with ports of the type(s) indicated by *ptype* will be included, regardless of whether or not the ports are occupied.

EdSetBus(bus, template)
bustype *bus;
boolean template;

EdSetBus displays a menu of the buses in the current graph, prompts the user for a bus, and sets *bus* to point to the selected bus. If *template* is TRUE, a bus template is wanted, so the menu contains bus template names.

EdSetArcTmp(arc)
arctype *arc;

EdSetArcTmp creates a menu of arc names from the current graph and allows the user to select an arc. The variable *arc* is then set to point to this arc. *EdSetArcTmp* returns an editor return code, e.g., ED_DONE.

EdSetPort(port_type, occ, ioflag, port)
int port_type;
int occ;
int *ioflag;
int *port;

EdSetPort prompts the user for a port on *ned*, the node currently being edited. The value of *port_type* is interpreted as a bit vector whose bits are set to indicate which port types should be included in the menu where the user makes his selection. The least significant bit is for inports, the next for outports, and the third for biports. The value of *occ* should be 0 for only the unoccupied ports to be included in the menu, 1 for only occupied ports, and 2 for both. The integer pointed to by *ioflag* is set to the type of port selected (IN, OUT, or BI) and the integer pointed to by *port* is set to the index of the selected port. *EdSetPort* returns an editor return code, e.g., ED_DONE.

EdSetArcNode(node)
nodetype *node;

EdSetArcNode creates a menu of nodes to which arcs are attached and allows the user to select a node. The node pointed to by *node* is set to the selected node.

PART V

ADAS Graphics Interface Routines

INTRODUCTION

The ADAS Graphics Interface (libgraphics) provides a device-independent interface for ADAS programs. It hides the device-dependent details of the graphics system which are isolated in the Graphics Package (libgp).

INITIALIZING AND TERMINATING GRAPHICS

GIInitializeDisplay(disp, libdir, configdir)
char *disp;
char *libdir;
char *configdir;

GIInitializeDisplay initializes the graphics display and the tablet of workstation *disp*, setting the window and viewport to appropriate startup values. The ADAS library directory to be used is *libdir* and the configuration directory is *configdir*.

GITerminateDisplay()

GITerminateDisplay shuts down the graphics display.

COORDINATE SYSTEMS, VIEWPORTS, AND WINDOWS

The ADAS graphics interface operates in terms of two types of coordinate systems. *Normalized Device Coordinates* (NDC) are used to specify screen locations with respect to the entire graphics display. They range from (0.0, 0.0) at the lower left corner of the screen, to (1.0, 1.0) at the upper right corner of the screen. *Viewports* are regions of the display screen defined in the NDC system. A client of the graphics interface might define two viewports: for example, one for displaying ADAS graphs and the other for menus. Associated with each viewport is a *window*. Items drawn in a window appear on the screen in the associated viewport. *World* or *Window* coordinates (WC) are used to refer to locations within a window. When a client defines a window, he specifies the range of coordinates for that window.

GIAskWindow(xmin, ymin, xmax, ymax)
int *xmin, *ymin, *xmax, *ymax;

GIAskWindow gets the coordinates of the lower left (*xmin, ymin*) and upper right (*xmax, ymax*) corners of the **current** window in window coordinates.

GISetWindow(xmin, ymin, xmax, ymax)
int xmin, ymin, xmax, ymax;

GISetWindow sets the current window, with the lower left corner (*xmin, ymin*) and upper right corner at (*xmax, ymax*) given in the window coordinates.

GIGetGraphWindow(xmin, ymin, xmax, ymax)
int *xmin, *ymin, *xmax, *ymax;

GIGetGraphWindow gets the coordinates of the lower left (*xmin, ymin*) and upper right (*xmax, ymax*) corners of the **graph** window in window coordinates.

GISetGraphWindow(xmin, ymin, xmax, ymax)
int xmin, ymin, xmax, ymax;

GISetGraphWindow sets the graph window with the lower left corner (*xmin, ymin*) and upper right corner (*xmax, ymax*) given in window coordinates.

GISetViewport(xmin, ymin, xmax, ymax)
double xmin, ymin, xmax, ymax;

GISetViewport sets the current viewport, with its lower left corner at (*xmin, ymin*) and its upper right corner at (*xmax, ymax*) (NDC).

boolean GIInWindow(x, y)
int x, y;

GIInWindow returns TRUE if the point (*x, y*) is within the current window.

boolean GIInViewport(x, y)
double x, y;

GIInViewport returns TRUE if the point (*x, y*) is within the current viewport.

GIResetGraphWindow()

GIResetGraphWindow sets the graph window to include the whole graph.

GIUseWVP(w)
int w;

GIUseWVP makes the window *w* and its corresponding viewport the current ones.

DRAWING GRAPH ELEMENTS

The following routines are used to draw ADAS graph elements on the graphics display.

GIDrawArc(a, eflag)
arctype a;
int eflag;

GIDrawArc draws the arc *a*. If *eflag* is nonzero, then the arc is drawn in the special ERASE color, meaning it is erased.

GIDrawPartialArc(a, eflag)
arctype a;
int eflag;

GIDrawPartialArc draws all of arc *a* except for the segment from the last joint (or source if there are no joints) to the sink. If *eflag* is nonzero, that part of the arc is erased instead.

GIDrawNode(n, eflag, fillcolor, outline)
nodetype n;
int eflag;
int fillcolor;
int outline;

GIDrawNode draws the node *n*. If *eflag* is nonzero, then the node is drawn in the special ERASE color, i.e., it is erased. If *fillcolor* is nonzero it is used as the node color index and *outline* is used as the outline color index. Otherwise the node color is determined by the current node drawing style.

GIDrawBus(b, eflag)
bustype b;
int eflag;

GIDrawBus draws the bus *b*. If *eflag* is nonzero, then the bus is drawn in the special ERASE color, i.e., it is erased.

GIDrawJunction(b, j, eflag)
bustype b;
int j;
int eflag;

GIDrawJunction draws junction *j* of bus *b*. If *eflag* is nonzero, then the junction is drawn in the special ERASE color, i.e., it is erased.

GIDrawNodePorts(n, eflag)
nodetype n;
int eflag;

GIDrawNodePorts draws the ports for node *n*. If *eflag* is nonzero, the ports are erased instead.

THE CONTEXT WINDOW

The ADAS context window displays the parent graph of the current graph with the parent node highlighted.

GIDrawContext()

GIDrawContext draws the ADAS context window.

boolean GIDrawingContext()

GIDrawingContext returns TRUE if the context window is being drawn, and FALSE otherwise. This routine is provided so that other routines can determine whether they are drawing in the context window.

boolean GIIsContextParent(n)
nodetype n;

GIIsContextParent returns TRUE if node *n* is the parent node of the current graph, and FALSE otherwise.

GISetContextWindow(x1, y1, x2, y2)
int x1, y1, x2, y2;

GISetContextWindow sets the context window corners at (*x1*, *y1*) and (*x2*, *y2*), given in window coordinates. The context window is defined in terms of the graph window coordinates.

DRAWING GRAPH ELEMENT LABELS

The following routines are used to draw labels for ADAS graph elements.

GIDrawNodeLabel(n, eflag)
nodetype n;
int eflag;

GIDrawNodeLabel displays the label for node *n*. If *eflag* is nonzero, the label is erased instead.

GIDrawBusLabel(b, eflag)
bustype b;
int eflag;

GIDrawBusLabel displays the label for bus *b*. If *eflag* is nonzero, the label is erased instead.

GIDrawJunctionLabel(b, j, eflag)
bustype b;
int j;
int eflag;

GIDrawBusLabel displays the label for junction *j* of bus *b*. If *eflag* is nonzero, the label is erased instead.

boolean GILabelsVisible(etype)
int etype;

GILabelsVisible returns TRUE if labels for element type *etype* are currently turned on.

REDRAWING A GRAPH

The following routines are used to redraw an ADAS graph or a subset of the elements of the graph.

GIRedrawScreen()

GIRedrawScreen erases and redraws the whole graphics image.

GIRedrawNodes()

GIRedrawNodes redraws all the nodes in the graph.

GIRedrawArcs()

GIRedrawArcs redraws all the arcs in the graph.

GIRedrawBuses()

GIRedrawBuses redraws all the buses in the graph.

GIRedrawLabels()

GIRedrawLabels redraws all the node labels in the graph. Labels for other graph elements are drawn when the elements themselves are drawn.

GIRedrawPorts()

GIRedrawPorts redraws all the node ports in the graph.

DISPLAYING TEXT

The following routines are used to draw text on the graphics display.

GICenterText(s, x, y, color)
char *s;
int x, y;
int color;

GICenterText draws the string pointed to by *s* in color *color* at the location (*x*, *y*) (WC) in the current window.

GILeftText(s, x, y, color)
char *s;
int x, y;
int color;

GILeftText displays the text string pointed to by *s*, left-justified at location (*x*, *y*) (WC) in color *color*.

GIRightText(s, x, y, color)
char *s;
int x, y;
int color;

GIRightText displays the text string pointed to by *s*, right-justified at location (*x*, *y*) (WC) in color *color*. The text is centered with respect to the given y coordinate.

GISetWCCharWidth(cw)
float cw;

GISetWCCharWidth sets the text character width to *cw*.

GIRotateText(on)
boolean on;

GIRotateText turns on text rotating if *on* is TRUE, and turns it off otherwise. When text rotating is on, text is displayed in an orientation rotated 90 degrees from the default orientation.

GILandscapeMode(on)
boolean on;

GILandscapeMode turns landscape mode on if *on* is TRUE, and turns it off otherwise. When in landscape mode, the graphics coordinate system is rotated 90 degrees counterclockwise from its

default orientation. Landscape mode must be turned on before rotated text is displayed.

BASIC DRAWING OPERATIONS

The following routines are used to do basic drawing operations. They are "lower level" routines than those for drawing ADAS graph elements.

GIErase()

GIErase erases the graphics display by filling it with the special ERASE color.

GIDrawLine(xmin, ymin, xmax, ymax, color)
int xmin, ymin, xmax, ymax;
int color;

GIDrawLine draws a line from (*xmin*, *ymin*) to (*xmax*, *ymax*) (WC) in the color *color*.

GIDrawRect(color, llx, lly, urx, ury, fill)
int color;
int llx, lly, urx, ury;
boolean fill;

GIDrawRect draws a rectangle with lower left corner (*llx*, *lly*) and upper right corner (*urx*, *ury*) (WC) in the color *color*. If *fill* is TRUE, a filled rectangle is drawn; otherwise the rectangle is unfilled.

MAKING HARDCOPIES

These routines are used for making a hardcopy of an ADAS graph.

GIHardcopy(harddev)
char *harddev;

GIHardcopy makes a hardcopy of the current graph using hardcopy device *harddev*.

boolean GIInHardcopy()

GIInHardcopy returns TRUE if the current process is a child which was forked off to do a hardcopy.

NODE LATENCY RANGE

GISetLatencyRange()

GISetLatencyRange sets the minimum and maximum latencies for normalization. *GISetLatencyRange* examines the latencies for all nodes and sets the minimum and maximum internally. They can be retrieved with *GIGetLatencyRange*.

GIGetLatencyRange(lmin, lmax)
double *lmin, *lmax;

GIGetLatencyRange gets the minimum (*lmin*) and maximum (*lmax*) latency values for normalization.

NODE AND ARC STYLES

Normally nodes and arcs are drawn in the colors given by their color attributes. By setting the node or arc style, they can be drawn in colors reflecting the values of certain other attributes.

int GISetNodeStyle(style)
int style;

GISetNodeStyle sets the node style to *style*, specifying how node color is determined. A node is displayed either according to its *node_color* attribute (DFLT_STYLE), with a heated object scale based on either utilization (NUTIL) or latency(NLATENCY), or according to its status (NSTATUS). When the node style is NSTATUS, nodes primed during simulation are drawn as unfilled rectangles, while others are shown in DFLT_STYLE. *GIGetNodeStyle* returns the previous node style value.

int GISetArcStyle(style)
int style;

GISetArcStyle sets the arc style to *style*, specifying how arc colors are determined. Arcs may be drawn according to the *arc_color* attribute (DFLT_STYLE) or using a heated object scale, in which an arc's queue size determines its color (ACURR_SIZE). *GISetArcStyle* returns the previous value of the arc style.

int GISetLabelStyle(style)
int style;

GISetLabelStyle sets the node label style to *style*. The label style determines whether a node's label corresponds to its *node_name* attribute (DFLT_STYLE) or its *hardware_module* attribute (NMODULE). *GISetLabelStyle* returns the previous value of the node label style.

SETTING THE QUEUE SCALE

GISetQueueLB(lb)
int lb;

GISetQueueLB sets the lower bound of user-defined queue scale to *lb*.

GISetQueueUB(ub)
int ub;

GISetQueueUB sets the upper bound of user-defined queue scale to *ub*.

DETERMINING NODE DIMENSIONS

double GINodeHeight(n)
nodetype n;

GIGetNodeHeight returns the height of the node *n*. This is the value of the **node_height** attribute, which is in ADAS grid coordinates.

double GINodeWidth(n)
nodetype n;

GIGetNodeWidth returns the width of the node *n*. This is the value of the **node_width** attribute, which is in ADAS grid coordinates.

GINodeDimensions(n, xsz, ysz)
nodetype n;
double *xsz, *ysz;

GINodeDimensions sets the doubles pointed to by *xsz* and *ysz* to the actual dimensions of node *n*. These dimensions are in ADAS grid coordinates, but may differ from the values returned by *GINodeHeight* and *GINodeWidth* if *n* is represented by an icon.

ICONS

ADAS graph elements may be represented by icons which are stored in files. If a graph element's **icon_file_name** attribute is not empty, the icon in the named file is used to represent that element. A list of active icons is maintained in the graphics interfacem, so an icon file is only read once during a session.

GIReadIconFile(filename)
char *filename;

GIReadIconFile reads the icon description file *filename* and puts the icon into the list of active icons.

icon *GIGetIcon(filename)
char *filename;

GIGetIcon returns a pointer to the entry in the list of active icons occupied by the icon stored in file *filename*. It returns NULL if the icon is not active.

MISCELLANEOUS GRAPHICS ROUTINES

GIRefresh()

GIRefresh refreshes the graphics display.

boolean GITooSmall()

GITooSmall returns TRUE if the graph scale is so small that node labels and ports should not be drawn.

char *GIInquireDisplayDevice()

GIInquireDisplayDevice returns the name of the graphics display being used.

char *GIGetDisplayType()

GIGetDisplayType returns the name of the type of graphics display being used.

GIDrawGrid()

GIDrawGrid draws the ADAS graph grid.

boolean GIPortsVisible()

GIPortsVisible returns TRUE if the graph scale is such that node ports should be drawn. Node ports are not drawn if they would be so small as to be useless.

PART VI

ADAS Common Library Routines

INTRODUCTION

The ADAS common library (libcommon) contains routines that are used by more than one ADAS program, but do not belong in the Command Interpreter, Data Base Routines, Graphics Interface, Graphics Package, or Editor.

SHARED COMMANDS

These routines are related to ADAS top-level commands that are identical in EDIGRAF and GIPSIM.

Hardcopy()

Hardcopy is the top-level routine for the **hardcopy** command. It prints a copy of the current graph on a hardcopy device. The user chooses from a menu of available hardcopy devices.

Help()

Help is the top-level routine for the **help** command which provides help on various ADAS topics. The user selects the tool and subject for which he needs help, and the contents of a help file in ADAS help directory are displayed on the terminal screen.

GetHelpTopics(menu, helpdir)
menutype *menu;
char *helpdir;

GetHelpTopics adds the names of the help files in the help directory *helpdir* to the menu *menu*.

catfile(file)
char *file;

catfile prints the contents of the file specified by pathname *file* to standard output. It returns 0 if the file was successfully printed and 1 if the file couldn't be opened.

Macro()

Macro is the top-level routine for the **macro** command. It provides for the addition, deletion, and listing of macro commands. *Macro* returns 0 on a normal completion and -1 if an error occurred.

Reload()

Reload is the top-level routine for the **reload** command. It reloads the current graph from the corresponding disk file after the user verifies his intent to reload. Any changes made since the graph was last saved to the disk file are lost.

Script()

Script is the top-level routine for the **script** command. It prompts the user for a script file name and executes the script file.

Stats()

Stats is the top-level routine for the **stats** command. It provides statistics on graph elements of the user's choice. The user chooses the type of stats he wants from a menu. "Node" or "arc" stats give the values of all the attributes for a node or arc. "Latency" or "utilization" redraws all the nodes in a heated object scale representing their relative latencies or utilizations. A key for the color scale is displayed on the screen.

DrawStatsKey(label1, label2)
char *label1, *label2;

DrawStatsKey draws the color scale key for the stats command, labeling the key with *label1* and *label2* (minimum and maximum stats values).

node_stats(n)
nodetype n;

node_stats prints the attribute values for node *n*.

bus_stats(b)
bustype b;

bus_stats prints the attribute values for bus *b*.

partition_stats(p)
partitiontype p;

partition_stats prints the attribute values for partition *p*.

arc_stats(a)
arctype a;

arc_stats prints the attribute values for arc *a*.

SetQSBounds(label1, label2)
char *label1;
char *label2;

SetQSBounds prompts the user for the lower and upper bounds for the **stats queue** command. The strings pointed to by *label1* and *label2* are set to contain the string representations of the lower and upper bounds, respectively. *SetQSBounds* calls *GISetQueueLB* and *GISetQueueUB* to set the queue bounds for the drawing routines.

GetQSBounds(label1, label2)
char *label1;
char *label2;

GetQSBounds copies the current queue size stats labels into *label1* and *label2*.

Window()

Window is the top-level routine for the **window** command. The window command allows the user to zoom in or out on a particular point in the graph by specifying a scaling factor and the point he

wishes to zoom in on.

DoWindow(x, y, wsize)
int x, y;
float wsize;

DoWindow centers the graph at location (*x*, *y*) with window size *wsize*. With larger values of *wsize*, the portion of the graph that will be visible is greater, and consequently the graph elements appear smaller. If *wsize* is too large the window size used is GRAPHMAX.

LIST MANAGEMENT

These routines are used to manipulate alphabetical lists of strings.

SetFirstNodeAlphabetic()

SetFirstNodeAlphabetic creates an alphabetical list of the nodes in the current data base and sets a pointer to the first node in the list.

GetNextNodeAlphabetic(np)
struct node **np;

GetNextNodeAlphabetic sets the pointer pointed to by *np* to point to the current node in the alphabetical list of nodes and updates the list pointer so that the next node becomes the current one.

boolean MoreNodesAlphabetic()

MoreNodesAlphabetic returns TRUE if there are nodes remaining in the alphabetical list of nodes and FALSE if there are none.

AlphaNodeAdd(n)
nodetype n;

AlphaNodeAdd inserts node *n* into the alphabetized list of nodes.

AlphaInsert(list, s)
namelink **list;
char *s;

AlphaInsert inserts the string *s* into the alphabetized list of strings where the head pointer is pointed to by *list*.

int CompStrings(s1, s2)
char *s1;
char *s2;

CompStrings compares *s1* and *s2* alphabetically or numerically by trailing digits if the strings have identical alphabetic prefixes. For instance, "a1" would come before "b1." but "b2" would come before "b12." It returns -1 if *s1* comes before *s2*, 0 if *s1* is identical to *s2*, or 1 if *s1* comes after *s2*.

FreeList(list)
namelink **list;

FreeLisT frees the memory occupied by the list of strings whose head pointer is pointed to by *list* and sets the head pointer to NULL.

DYNAMIC MEMORY MANAGEMENT

char *myalloc(nelem, elsize)
int nelem;
unsigned elsize;

myalloc calls *calloc* to allocate memory for *nelem* elements, each of size *elsize*. It checks to make sure the memory was actually allocated and returns a pointer to the newly allocated memory.

myfree(x)
char *x;

myfree frees the dynamically-allocated storage pointed to by *x*.

SIGNAL HANDLING AND TERMINAL MODES

SigSetup()

SigSetup sets up the signal handling routines for the UNIX signals. SIGPIPE and SIGCHLD are ignored. SIGINT and SIGTERM cause *TrapSignal* to be called, and SIGHUP is handled by the client-supplied routine *adas_abort*. SIGTSTP is handled in its default manner. The remaining signals are ignored by the default action if they should be. The rest are handled by *Cleanup*.

IgnoreSignals()

IgnoreSignals causes SIGINT and SIGTERM to be ignored. This is used before processing signals so new signals coming in don't mess things up.

TrapSignal()

TrapSignal asks the user if he really wants to abort the program. If so, the client-supplied routine *adas_abort* is called. Otherwise, the program continues. Signals are ignored while *TrapSignal* is executing.

Cleanup(sig)
int sig;

Cleanup closes the graphics display, resets the signal *sig* to its default behavior, and resends *sig* to itself.

StartCBreak()

StartCBreak puts the terminal into cbreak mode where keystrokes can be detected before a carriage return is entered.

StopCBreak()

StopCBreak takes the terminal out of cbreak mode so that a carriage return must be entered before keystrokes can be detected.

CONVERSION ROUTINES

These routines perform conversions among different data types.

atoc(s, xp, yp)
char *s;
int *xp, *yp;

atoc converts the string representation, *s*, of a coordinate into the integer coordinates. The integers pointed to by *xp* and *yp* are set to the x and y coordinates, respectively. The string is of the form "x,y". If string has a bad format, the location returned is (-1,-1).

char *ctoa(x, y)
int x, y;

ctoa returns a pointer to a static buffer containing the string representation of the coordinate (*x*, *y*). The buffer is overwritten on subsequent calls.

char *itoa(n)
int n;

itoa returns a pointer to a static buffer containing the ascii representation of the integer *n*. The integer must have fewer than 25 decimal digits.

CREATING MENUS

These routines are used to create ADAS menus to be used with the command interpreter.

boolean MakeNodeMenu(gd, menu, type, port_status)
int gd;
menutype **menu;
int type;
int port_status;

MakeNodeMenu creates a menu of nodes in the graph specified by graph descriptor *gd* and sets the pointer, pointed to by *menu*, to point to the new menu. The values of *type* and *port_status* determine which nodes are added. The value of *port_status* is 0 if only nodes with unoccupied ports are to be included, 1 for only nodes with occupied ports, 2 for all nodes, and 3 for all nodes with occupied or unoccupied ports of the type(s) indicated by *type*. If *port_status* is not 2, *type* is interpreted as a bit vector indicating which types of ports should be considered. The least significant bit is for inports, the next for outports, and the third for bi-ports. *MakeNodeMenu* returns TRUE if the menu it creates is empty, and FALSE otherwise. It assumes that *gd* is a valid graph descriptor and that *type* and *port_status* are valid.

MakeArcMenu(gd, menu, tmpflag)
int gd;
menutype **menu;
int tmpflag;

MakeArcMenu creates a menu of arcs in the graph specified by graph descriptor *gd*. Also, *MakeArcMenu* sets the pointer, pointed to by *menu*, to point to the new menu. If *tmpflag* is nonzero, the menu is one of arc templates and will contain the arc template names for the graph; otherwise, it is an arcs menu containing the names of the source nodes found in the graph's arcs.

MakeArcMenu returns TRUE if the menu is empty and FALSE otherwise. It assumes *gd* is a valid graph descriptor.

MakeBusMenu(gd, menu)
int gd;
menutype **menu;

MakeBusMenu creates a menu of buses in the graph specified by graph descriptor *gd* and sets the pointer, pointed to by *menu*, to point to the new menu.

MakeNodeAndBusMenu(gd, menu, type, port_status)
int gd;
menutype **menu;
int type;
int port_status;

MakeNodeAndBusMenu creates a menu of nodes and buses in the graph specified by graph descriptor *gd* and sets the pointer pointed to by *menu* to point to the new menu. The *type* and *port_status* parameters are used as described in *MakeNodeMenu* for selecting nodes to be included. Also, if *port_status* is 1, only buses having at least one arc attached to them are included.

MakeJunctionMenu(menu, b, occ)
menutype **menu;
bustype b;
int occ;

MakeJunctionMenu creates a menu of junctions on bus *b* and sets the pointer, pointed to by *menu*, to point to the new menu. If *occ* is nonzero, only junctions with at least one arc attached are included in the menu.

MakePartitionMenu(gd, menu)
int gd;
menutype **menu;

MakePartitionMenu creates a menu of partitions in the graph specified by graph descriptor *gd* and sets the pointer, pointed to by *menu*, to point to the new menu.

boolean MakePortMenu(menu, n, ports, occ)
menutype **menu;
nodetype n;
int ports;
int occ;

MakePortMenu creates a menu of ports of node *n* and sets the pointer, pointed to by *menu*, to point to the new menu. If *occ* is 0, only unoccupied ports are included. If it is 1, only occupied ones are included; if it is 2, all ports are included. If *occ* is not 2, *ports* is interpreted as a bit vector with bits set for the types of ports to be included. The least significant bit is set for inports, the next for outports, and the third for biports. *MakePortMenu* returns TRUE if the menu is empty and FALSE otherwise.

MakeAttMenu(menu, type, statflag, editflag)
menutype **menu;
int type;

boolean statflag;
boolean editflag;

MakeAttMenu makes an alphabetical menu of attributes for the graph element type *type* (GRAPH, NODE, BUS, ARC, PARTITION, IN, OUT, or BI) and sets the pointer, pointed to by *menu*, to the new menu. If *statflag* is TRUE the menu is for attribute statues, and certain attributes are not included. If *editflag* is TRUE only the editable attributes appear in the menu. Otherwise, all attributes appear in the menu.

boolean CMMakeMenus(menu, done_flag, graph_flag, node_flag, part_flag)
menutype **menu;
boolean done_flag;
boolean graph_flag;
boolean node_flag;
boolean part_flag;

CMMakeMenus creates a menu pointed to by the pointer pointed to by *menu*. If *done_flag* is TRUE, "%done" will be the first item in the menu. If *graph_flag* is TRUE "graph" is added. If *node_flag* is TRUE the names of all nodes in the current graph are added to the menu. If *part_flag* is TRUE, the names of all partitions in the current graph are added to the menu.

DRAWING GRAPH ELEMENTS

These routines are related to the display of ADAS graphs on a graphics device. They call routines in the ADAS Graphics Interface (libgraphics).

DrawNodeAndArcs(n, erase)
nodetype n;
int erase;

DrawNodeAndArcs displays the node *n* and all the arcs attached to its ports. If *erase* is nonzero the node and arcs are erased instead of drawn.

DrawBusAndArcs(b, erase)
bustype b;
int erase;

DrawBusAndArcs displays the bus *b* and all the arcs attached to its junctions. If *erase* is nonzero the bus and arcs are erased instead of drawn.

DrawJunctionAndArcs(b, j, erase)
bustype b;
int j;
int erase;

DrawJunctionAndArcs displays junction *j* of bus *b* and all the arcs attached to it. If *erase* is nonzero the junction and arcs are erased instead of drawn.

CMRedrawScreen()

CMRedrawScreen redraws the graphics image.

CenterGraph()

boolean statflag;
boolean editflag;

MakeAttMenu makes an alphabetical menu of attributes for the graph element type *type* (GRAPH, NODE, BUS, ARC, PARTITION, IN, OUT, or BI) and sets the pointer, pointed to by *menu*, to the new menu. If *statflag* is TRUE the menu is for attribute statues, and certain attributes are not included. If *editflag* is TRUE only the editable attributes appear in the menu. Otherwise, all attributes appear in the menu.

boolean CMMakeMenus(menu, done_flag, graph_flag, node_flag, part_flag)
menutype **menu;
boolean done_flag;
boolean graph_flag;
boolean node_flag;
boolean part_flag;

CMMakeMenus creates a menu pointed to by the pointer pointed to by *menu*. If *done_flag* is TRUE, "%done" will be the first item in the menu. If *graph_flag* is TRUE "graph" is added. If *node_flag* is TRUE the names of all nodes in the current graph are added to the menu. If *part_flag* is TRUE, the names of all partitions in the current graph are added to the menu.

DRAWING GRAPH ELEMENTS

These routines are related to the display of ADAS graphs on a graphics device. They call routines in the ADAS Graphics Interface (libgraphics).

DrawNodeAndArcs(n, erase)
nodetype n;
int erase;

DrawNodeAndArcs displays the node *n* and all the arcs attached to its ports. If *erase* is nonzero the node and arcs are erased instead of drawn.

DrawBusAndArcs(b, erase)
bustype b;
int erase;

DrawBusAndArcs displays the bus *b* and all the arcs attached to its junctions. If *erase* is nonzero the bus and arcs are erased instead of drawn.

DrawJunctionAndArcs(b, j, erase)
bustype b;
int j;
int erase;

DrawJunctionAndArcs displays junction *j* of bus *b* and all the arcs attached to it. If *erase* is nonzero the junction and arcs are erased instead of drawn.

CMRedrawScreen()

CMRedrawScreen redraws the graphics image.

CenterGraph()

int type;

HasOpenPorts returns TRUE if any ports of type *type* on node *node* are not occupied. FALSE otherwise. Valid values for *type* are IN, OUT, and BI.

boolean HasUsedPorts(node, type)
nodetype node;
int type;

HasUsedPorts returns TRUE if any ports of type *type* on node *node* are occupied. FALSE otherwise. Valid values for *type* are IN, OUT, and BI.

boolean HasArcs(name)
char *name;

HasArcs returns TRUE if node *name* has any output arcs, otherwise FALSE.

SetArcJoints(joint, tmparc)
char **joint;
arctype tmparc;

SetArcJoints prompts the user to enter joints for the arc *tmparc* until the user enters "~odone". The joint values are stored in *joint*.

SETTING AND RETRIEVING NAMES

SetGraphName(s)
char *s;

SetGraphName sets the graph name to *s*. The graph type (hardware or software) is determined by the extension on *s*. The default graph type (if no extension) is software. If the graph name doesn't have an extension (either ".swg" or ".hwg"), then the current extension is tacked on. The default extension is SOFTEXT.

char *GetGraphName()

GetGraphName returns a pointer to a static buffer containing the name of the current graph which was set by *SetGraphName*. The static buffer is overwritten on subsequent calls.

SetPgmName(s)
char *s;

SetPgmName sets the program name to *s*. If the program name has a path, e.g., "/usr/foo/pgm," all but the last part is stripped off.

char *GetPgmName()

GetPgmName returns a pointer to a static buffer containing the name of the program which was set by *SetPgmName*. The static buffer is overwritten on subsequent calls.

SetCDBName(name, level)
char *name;
int level;

SetCDName sets the data base name for the graph or subgraph level *level* in the graph hierarchy to *name*. It assumes *level* <= MAXDEPTH and *name* is MAXPATH or fewer characters in length.

char *GetCDBName(level)
int level;

GetCDBName returns a pointer to a static buffer containing the name of the graph at level *level* in the graph hierarchy. The static buffer is overwritten on subsequent calls.

char * GetCurrDBDir()

GetCurrDBDir returns the name of the directory in which the current data base file resides. The first time it is called, it stores the name in a static buffer; then it returns a pointer to that buffer on that call and subsequent ones.

char * GetGraphExt()

GetGraphExt returns a pointer to a static buffer containing the graph name extension (".swg" or ".hwg") of the current data base.

char * MakeSubgrafName(node)
nodetype node;

MakeSubgrafName builds a subgraf name based on the value of the **subgraf_file_name** attribute of node *node*. It returns a pointer to the dynamically allocated string.

char * PathFromRoot(node)
nodetype node;

PathFromRoot creates the full path name of *node* from the top-level graph in the graph hierarchy and returns a pointer to the dynamically allocated string containing the path name. The full path name consists of the names of *node* and all its ancestor nodes, starting with the node in the top-level graph, separated by colons (':').

char *GetCDBTemplateName(level)
int level;

GetCDBTemplateName returns a pointer to a static buffer containing the name of the template data base for level *level* in the graph hierarchy.

boolean CMNodeName(s)
char *s;

CMNodeName returns TRUE if *s* is the name of a node in the current data base, FALSE otherwise.

MISCELLANEOUS COMMON LIBRARY ROUTINES

TraverseAndExpand(level)
int level;

TraverseAndExpand recursively traverses the graph hierarchy, expanding nodes into their

subgraphs, then traversing the subgraphs. The value of *level* gives the level in the hierarchy that is currently being traversed. Its value should be zero for the initial call to *TraverseAndExpand*.

RefreshWindow()

RefreshWindow redraws the entire graphics display. It is called from the graphics driver for the X window system to redraw the graphics window when it is exposed.

char *GetCurrDate()

GetCurrDate returns a pointer to a static buffer containing the current time and date in an operating system-dependent format. The static buffer is overwritten on subsequent calls.

boolean IsPositive(val)
int val;

IsPositive returns TRUE if *val* is greater than or equal to zero. FALSE otherwise.

boolean IsValidUB(val)
int val;

IsValidUB returns TRUE if *val* qualifies as a valid upper bound by being greater than or equal to the current lower bound. It returns FALSE otherwise.

get_env(log_name)
char *log_name;

get_env returns a pointer to a static buffer containing the value of the VMS logical name *log_name*.

ErrHandler(type, number, level, msgf, m1, m2, m3, m4, m5)
char *type;
int number;
char level;
char *msgf;
char *m1;
char *m2;
char *m3;
char *m4;
char *m5;

ErrHandler prints an error message in standard ADAS format. The string *type* is appended to the one-letter program code (usually the first letter of the program name) to form the alphabetic error code. This code, when combined with the error number *number*, identifies the error. The string *msgf* is used as the format specification for the message to be printed, and the strings *m1* through *m5* are put into that format using *sprintf*. Valid values for the error severity level, *level*, are "W" (WARNING), "E" (ERROR), "D" (DISASTER), and "A" (ABORT). *ErrHandler* assumes the existence of a client-supplied routine named *adas_abort* that terminates the program gracefully. This routine is called if the severity level is DISASTER or ABORT.

PrintHeader()

PrintHeader prints a standard ADAS program header. This header includes program name,

version number, and a copyright notice. The program name to be used is set with *SetPgmName*.

int GetMajorVer()

GetMajorVer returns the current ADAS major version number, i.e. "2" for Version 2.3.

int GetMinorVer()

GetMinorVer returns the current ADAS minor version number, i.e. "3" for Version 2.3.

strlcpy(s1, s2, n)
char *s1,
char *s2;
int n;

strlcpy copies up to *n* characters from *s2* to *s1*. It assumes *s1* is large enough to hold *n* characters.

boolean true()

true returns the boolean value TRUE. This is useful when a pointer to a function that will always return TRUE is needed.

boolean IsRelPath(path)
char *path;

IsRelPath returns TRUE if the *path* is a VMS relative path name, and FALSE otherwise. On UNIX this is accomplished with a macro.

boolean PortMismatch(node, subgrname)
nodetype node;
char *subgrname;

PortMismatch checks that the graph inports and graph outports of the subgraph named *subgrname* match in number, the inports and outports of its parent node, *node*. It also checks the graphport numbers of the graphports in the subgraph. If there is an error, *PortMismatch* prints an error message and returns TRUE. Otherwise it returns FALSE.

TmplInit(tgd, create)
int *tgd;
boolean create;

TmplInit initializes the template data base for the current graph. If *create* is TRUE, a new template data base is created. The integer pointed to by *tgd* is set to the template graph descriptor.

boolean file_exists(file)
char *file;

file_exists returns TRUE if the file named *file* exists; otherwise, FALSE. It uses the *access* system call.

boolean node_exists(name)
char *name;

node_exists returns TRUE if a node with name *name* exists in the current data base, otherwise FALSE.

SetPgmCode()

SetPgmCode sets the program error code, based on the program name returned by *GetPgmName*. The program error code is used in ADAS error messages. *SetPgmName* should always be called before *SetPgmCode*.

StatSym(st)
int st;

StatSym returns the value of the character associated with modification status *st* for the *Stats* command.

ValidLocation(loc)
char *loc

ValidLocation determines whether *loc* is a valid port location specification. It returns 1 if *loc* is valid and 0 otherwise.

boolean CMNodeOrBusName(name)
char *name;

CMNodeOrBusName returns TRUE if *name* is the name of a node or bus in the current graph, and FALSE otherwise.

CMNodeOrient(n)
nodetype n;

CMNodeOrient returns the node orientation of node *n*, either UP, DOWN, LFT, or RIGHT.

char * CMGetPortLoc(node, port, ioflag)
nodetype node;
int port;
int ioflag;

CMGetPortLoc returns a pointer to a static buffer containing the string representation of the location of port *port* of type *ioflag* on node *node*. If there is no location explicitly specified in the port's location attribute, the default location is computed and converted to string form. The location string consists of a one-letter direction code ('N', 'S', by a slash.

boolean CMNodeFromGraph(s)
char *s;

CMNodeFromGraph returns TRUE if *s* is the name of a node in the current graph, and FALSE otherwise.

CMRedrawContext()

CMRedrawContext redraws the ADAS context window.

TheEnd(status)

int status;

TheEnd terminates ADAS graphics and exits the program.

int CMGraphLocInWindow(c)
coordtype c;

CMGraphLocInWindow returns a nonzero value if coorinate *c* is in the current graph window, zero otherwise.

int IsUnoccupied(loc)
coordtype loc; returns a nonzero value if the location *loc* is a valid graph location and is not already occupied by a node or junction, zero otherwise.

boolean CMCaselessCmp(s1, s2)
char *s1;
char *s2;

CMCaselessCmp does a caseless string comparison of *s1* and *s2*. It returns a negative value if *s1* precedes *s2* alphabetically, zero if they are the same, and a positive value if *s1* follows *s2*.

int CMIsInBounds(c)
coordtype c;

CMIsInBounds returns a nonzero value if coordinate *c* is within the valid graph coordiante range, zero otherwise.

TEA AND VSCI STUFF

boolean CMBusInArcSubrange(subelement, ap)
int subelement;
arctype ap;

CMBusInArcSubrange returns TRUE if *subelement* is included in the **bus_subrange** of arc *ap*, and FALSE otherwise.

CMBusNetlist()

CMBusNetlist allows the user to select a bus from a menu, then prints the netlist for that bus.

CMCheckArcSubRange(ap)
arctype ap;

CMCheckArcSubRange checks the value of the **bus_subrange** attribute for arc *ap*. It returns NOERROR if the value is valid and CANCEL otherwise.

CMCheckNodeCount()

CMCheckNodeCount checks whether the graph attribute **graph_node_count** exists. If so, it counts the number of nodes in the current graph and sets the **graph_node_count** attribute to that value.

CMCheckPartitionCount()

CMCheckPartitionCount checks whether the graph attribute **graph_partition_count** exists. If so, it counts the number of partitions in the current graph and sets the **graph_partition_count** attribute to that value.

CMEatBlanks(buffer)
char **buffer;

CMEatBlanks removes leading and trailing blanks from the string pointed to by *buffer*.

CMEditFile(filename, def_editor)
char *filename;
char *def_editor;

CMEditFile allows the user to edit file *filename* using the editor *def_editor*.

CMFileCheck(filename, exist, err_type, err_num)
char *filename;
boolean *exist;
char *err_type;
int err_num;

CMFileCheck checks the existence of the file *filename*, setting the boolean pointed to by *exist* to TRUE if it exists and FALSE otherwise. An error message is printed if (1) an error occurs in obtaining the status of the file, (2) the file is a directory or special file, or (3) the user does not have read permission on the file. The client of *CMFileCheck* passes it the error type, *err_type*, and the error number, *err_num* for use in the error message.

CMFileCopy(dest, src, err_type, err_num)
char *dest;
char *src;
char *err_type;
int err_num;

CMFileCopy copies the file named *src* to the file named *dest*. If either file cannot be opened, an error message is printed. The *err_type* and *err_num* parameters are used as in *CMFileCheck*.

CMGetConnType(node)
nodetype node;

CMGetConnType returns the connection type of node *node*. This is INTERIOR if the node's **node_class** is LEAF or INTERNAL, and EXTERIOR otherwise.

char *CMGetEditor(spec_editor)
char *spec_editor;

CMGetEditor returns the name of the editor of the user's preference. It first checks the **EDITOR** environment variable. If that is not set, it uses *spec_editor*; if NULL, it uses the system default editor. On VMS the default editor is "edit"; on UNIX it is "/usr/ucb/vi".

CMGetFileByNodeAtt(att, ext, gd, doexist, filename)
int att;
char *ext;
int gd;

boolean doexist;
char **filename;

CMGetFileByNodeAtt displays a menu of file names retrieved from attribute *att* in the nodes in the graph specified by graph descriptor *gd*. When the user selects a name, the extension *ext* is appended to it. The existence of the file with the resulting name is checked according to the value of *doexist*. If *doexist* is TRUE and the file does not exist, and error message is printed. If it is FALSE and the file exists, the user is asked whether he wants to replace the file. If he does not, *CMGetFileByNodeAtt* returns CANCEL. The pointer, pointed to by *filename*, is set to point to a static buffer containing the selected filename with its extension.

int CMGetLeftSubRange()

CMGetLeftSubRange returns the leftmost element of the current arc subrange.

CMGetNodeAttMenu(gd, att, menu_ptr)
int gd;
int att;
menutype **menu_ptr;

CMGetNodeAttMenu creates a menu of the values of attribute *att* for the nodes in the graph specified by graph descriptor *gd*. The pointer pointed to by *menu_ptr* is set to the new menu.

CMGetNodeByAttVal(gd, att, value, node, tmpl)
int gd;
int att;
char *value;
nodetype *node;
char *tmpl;

CMGetNodeByAttVal checks the graph specified by graph descriptor *gd* for nodes in which the value of attribute *att* is *value*. It sets *node* to point to this node. If more than one node has a matching attribute, the template name for the nodes is copied into the buffer *tmpl*. If nodes with matching values have different template names, an inconsistency warning message is printed.

CMGetPortClass(node, port, ioflag)
nodetype node;
int port;
int ioflag;

CMGetPortClass returns the port class of port *port*, of type *ioflag*, on node *node*. This is either SCALAR, CONSTRAINED, or UNCONSTRAINED.

CMGetPortName(node, port, dir)
nodetype node;
int port;
int dir;

CMGetPortName returns a pointer to a static buffer containing the port name of port *port*, of type *dir*, on node *node*. This is of the form "INPORT_X", where X is the port index, *port*.

CMGetPortRefName(node, port, ioflag)
nodetype node;

int port;
int ioflag;

CMGetPortRefName returns a pointer to a static buffer containing a string associated with port *port* of type *ioflag* on node *node*. If *node* is a graph port, the string is the node's name. Otherwise, the string is the one returned by *CMGetPortName*.

int CMGetPortSubnumber(subelement)
int subelement;

CMGetPortSubnumber returns the subelement of the array port to which bus subelement *subelement* is connected.

char *CMGetPortWidth(node, port, dir)
nodetype node;
int port;
int dir;

CMGetPortWidth returns a pointer to a static buffer containing the value of the width attribute of port *port* of type *dir* on node *node*. This buffer is maintained by the ADAS data base and will be overwritten on the first subsequent call to *DBGetPortAtt*.

int CMGetRightSubrange() returns the rightmost element of the current arc subrange.

CMIsBlank(buffer)
char *buffer;

CMIsBlank returns TRUE if all the characters in the string *buffer* are white space characters (space, tab, newline, form-feed), and FALSE otherwise.

CMIsContigSubrange()

CMIsContigSubrange returns TRUE is the current subrange is contiguous, FALSE otherwise.

CMMakeFileMenu(directory, ext, menu)
char *directory;
char *ext;
menutype **menu;

CMMakeFileMenu creates a menu of files in directory *directory* with file extension *ext*. The pointer, pointed to by *menu*, is set to point to the new menu.

ltok CMNext_Token(busrange)
char busrange[];

CMNext_Token returns the next token in *busrange*.

CMReplaceFile(filename, replace)
char *filename;
boolean *replace;

CMReplaceFile asks the user whether he wants to replace file *filename*. The boolean pointed to by *replace* is set to TRUE if he answers "yes" and to FALSE otherwise.

CMSetArcSubRange(attstring)
char *attstring;

CMSetArcSubRange parses the arc **bus_subrange** value given in *attstring* into its individual bits, and puts the appropriate bit values into the global *subrange* array for future reference.

CMSetFileAtt(type, name, ext, att, file_name, exist)
int type;
char *name;
char *ext;
int att;
char **file_name;
boolean *exist;

CMSetFileAtt prompts the user for a value for att *att* of graph element *name* of type *type*. This attribute should represent a file name. If the file whose name the user enters already exists, the boolean, pointed to by *exist*, is set to TRUE and the user is asked whether he wants to replace the file. The pointer, pointed to by *file_name*, is set to point to a static buffer containing the file name entered by the user.

CMSetNodeByClass(gd, class, template, node)
int gd;
int class;
boolean template;
nodetype *node;

CMSetNodeByClass creates a menu of nodes from the graph specified by graph descriptor *gd*, using *class* as a bit vector in which bits are set for the classes of nodes to be included. If *template* is TRUE, the menu contains node templates instead of nodes. The node pointed to by *node* is set to the node selected from the menu.

CMSysError(err_code, err_num, target)
char *err_code;
int err_num;
char *target;

CMSysError prints an ADAS error message using the *err_code* and *err_num* provided. The *target* argument names the item (e.g., file) that was responsible for the error.

boolean CMYes(prompt, cancel)
char *prompt;
boolean *cancel;

CMYes presents the user with a "yes no" menu and the prompt *prompt*, and gets his response. The boolean pointed to by *cancel* is set to TRUE if the user cancels and to FALSE otherwise. *CMYes* returns TRUE if the user enters "yes" and FALSE otherwise.

PART VII

ADAS Interrupt Handling

All VMS exceptions that are not mentioned in the interrupt requirements sections of particular TLCSCs will be fielded and will generate the same result:

a. a tool specific interrupt handler will be called,

b. all open files will be closed,

c. the current contents of the ADAS internal data base, if present, will be saved in an ADAS data base file with a special suffix, and

d. the interrupt handler will let the VMS exception handling mechanism continue.

Under UNIX the various signals are handled as follows:

a. The hangup signal, SIGHUP, causes the tool specific interrupt handler to be called.

b. When the interrupt signal, SIGINT, or the software termination signal, SIGTERM, is received, the user is asked to confirm his intention to abort the program. If he assents, the tool specific interrupt handler is called. Otherwise, the program resumes execution at the point where the signal was received.

c. The signals generated upon writing to a nonexistent pipe, SIGPIPE, and when the status of a child process changes, SIGCHLD, are ignored.

d. For SIGTSTP, which is generated when the user generates a stop signal from the keyboard, the default action is performed.

e. Among the remaining signals, those whose default action is to be ignored are ignored. For any others, the graphics device is first closed, then the default action for that signal is performed.

Appendix E
BIT Module Application Notes

8-BIT EQUAL COMPARATOR APPLICATION NOTE

Objective

The objective of this application note is to describe the functionality of an 8-bit comparator module and to show how it is used in a design to facilitate test.

Block Diagram

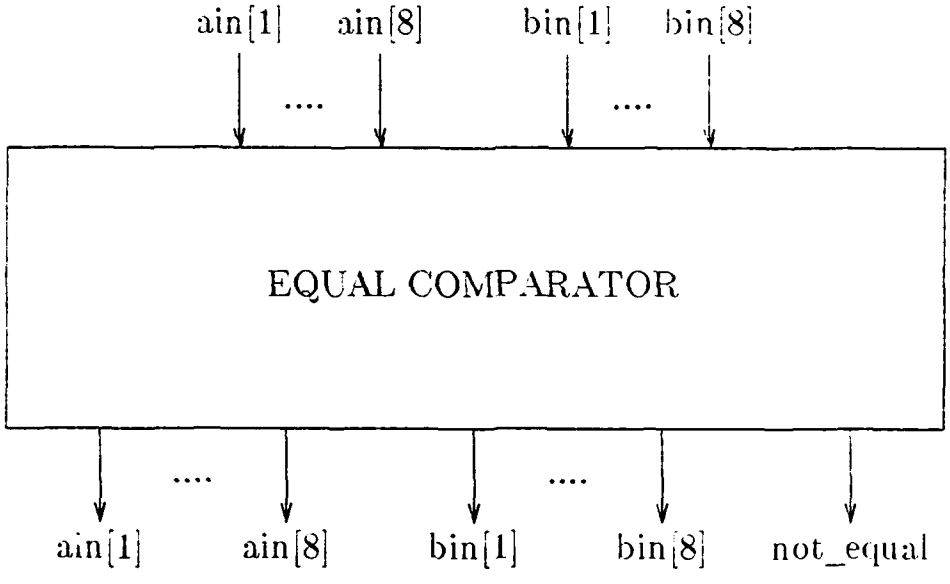


Figure 1. Block Diagram of the 8-bit Comparator Module

Pin Description

Name	Description
ain[8:1]	Operand A parallel data inputs
bin[8:1]	Operand B parallel data inputs
not_equal	If A equals B, then this signal is low

Function Table

The following table shows the complete function of the 8-bit parity generator/checker.

Function Table

Function	not_equal
A = B	L
A > B	H
A < B	H

Incorporation of the Equal Comparator Modules into Design

A hardware design can be partially tested by duplicating modules and comparing the outputs of the two modules. If there is a difference in the output values, then one or both of the modules is not calculating its result properly or the comparator is faulty. This is a concurrent form of testing which allows the user to see if there is a mismatch of data results between two equivalent modules. Figure 2 shows an example use of a Comparator module.

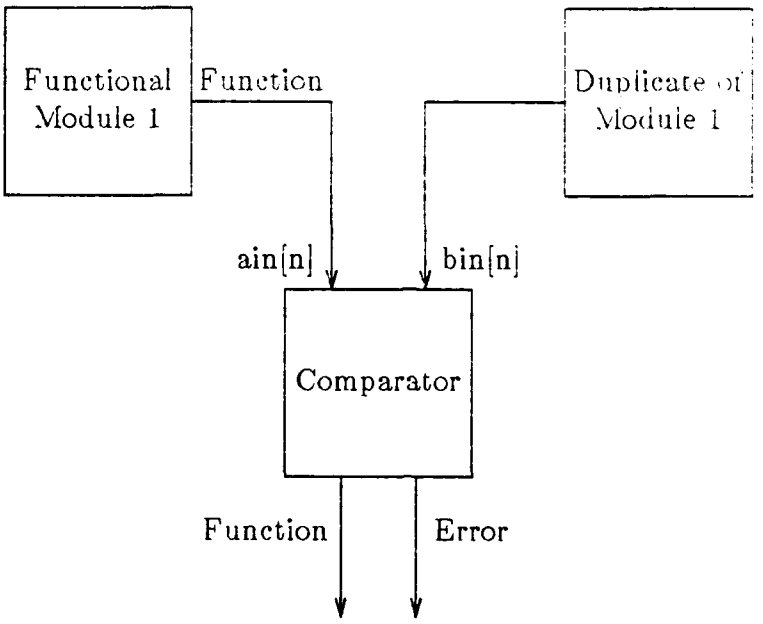


Figure 2. Example Showing the Incorporation of a Comparator Module in a Design

9-BIT PARITY GENERATOR/CHECKER APPLICATION NOTE

Objective

The objective of this application note is to describe the functionality of the 9-bit parity generator/checker module and to show how it is used in a design to facilitate test.

Block Diagram

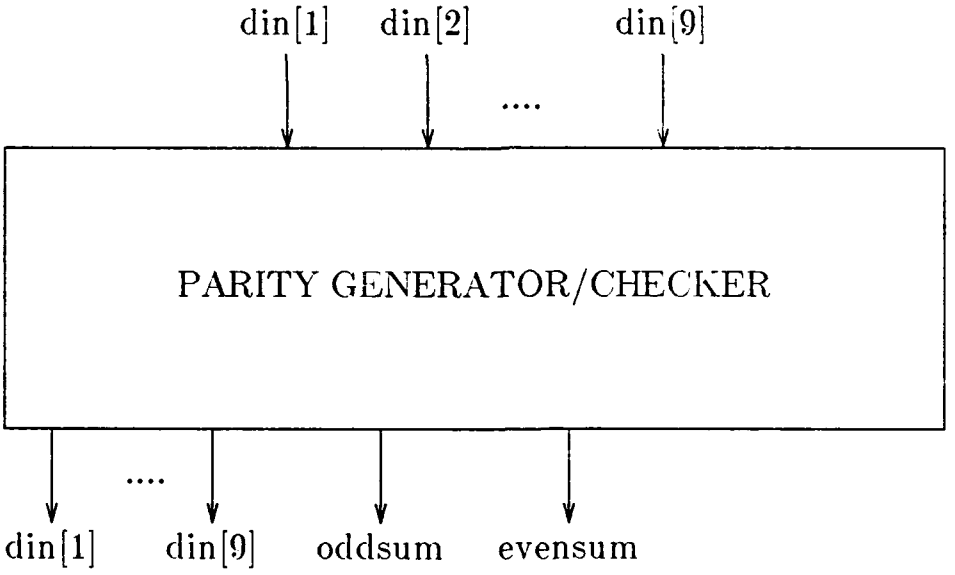


Figure 1. Block Diagram of the 9-bit Parity Generator/Checker Module

Pin Description

Name	Description
din[9:1]	Parallel data inputs of the parity generator/checker
oddsum	High output indicates an odd number of high inputs Low output indicates an even number of high inputs
evensum	High output indicates an even number of high inputs Low output indicates an odd number of high inputs

Function Table

The following table shows the complete function of the 9-bit parity generator/checker.

Function Table

Number of Inputs That Are High	Outputs	
	Evensum	Oddsum
0,2,4,6,8	H	L
1,3,5,7,9	L	H

Chaining to Accommodate a 25-bit Bus

Figure 2 shows how to use 3 parity generator/checker modules to accommodate a 25-bit bus.

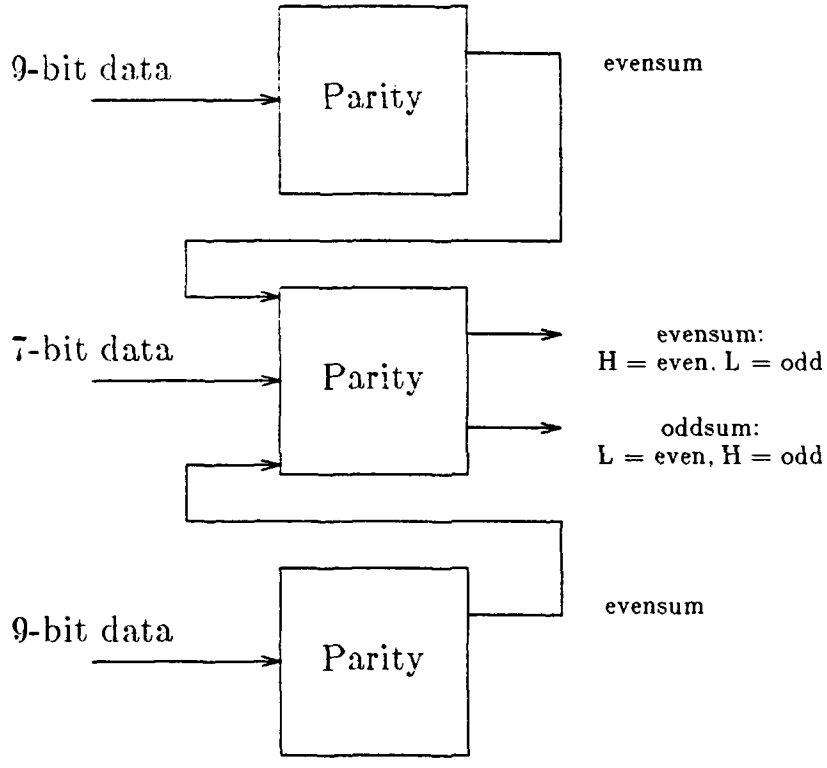


Figure 2. Application Example Showing the Chaining of Three Modules to Accommodate a 25-bit Bus

Incorporation of the Parity Generator/Checker Modules into Design

A hardware design can be partially tested by using parity generators/checkers on sources and receivers of the bus lines. This is a concurrent form of testing which allows the user to see if there is a mismatch of data between the source and the receiver.

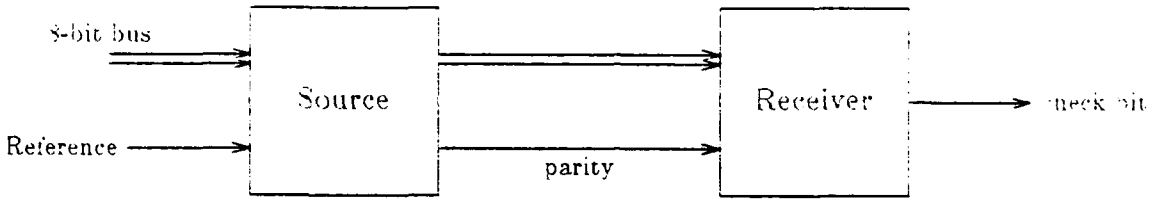


Figure 3. Example Showing the Incorporation of Odd Parity Generator/Checker Modules in a Design

BUILT-IN LOGIC BLOCK OBSERVER (BILBO) APPLICATION NOTE

Objective

The objective of this application note is to describe the functionality of a built-in logic block observer (BILBO) module and to show how it is used in a design in order to facilitate test.

Block Diagram

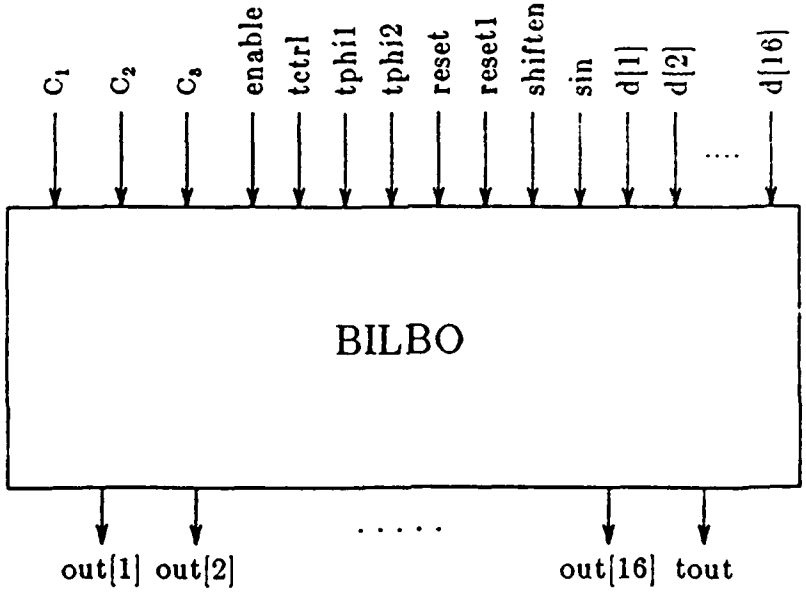


Figure 1. Block Diagram of a BILBO Module

Pin Description (see also the logic diagram and module functions)

Name	Description
C_1, C_2 and C_3	Mode control inputs
enble	Shift enble for control inputs
tctrl	Tristate control for "tout" output
tphi1/tphi2	Two-phase clock
reset	Reset input of all the flip-flops excluding the one corresponding to "shiften" input
reset1	Reset input of the flip-flop corresponding to the "shiften" input
shiften	Shift enble for data inputs
sin	Serial data input
d[16:1]	Parallel data inputs
out[16:1]	Parallel data outputs
tout	Serial test data output

Logic Diagram

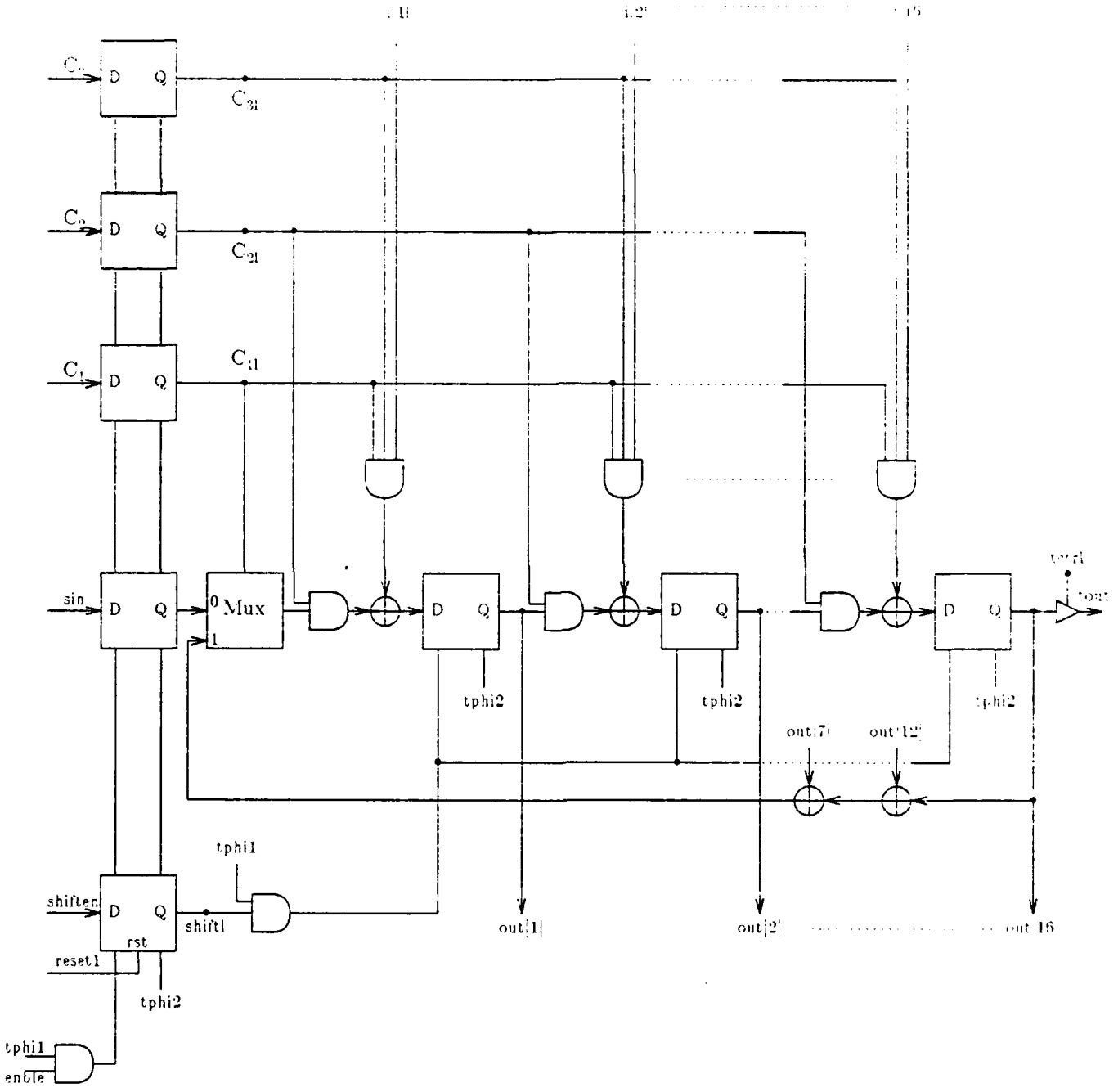


Figure 2. Logic Diagram of a BILBO Module

Module Functions

C_{1l}	C_{2l}	C_{3l}	enble	shiftl	
0	1	0/1	1	1	Serial shifting in of data through "sin"
1	0	1	0/1	1	Parallel transfer of inputs from "d[16:1]" to "out[16:1]"
1	1	0	0/1	1	Test pattern generator mode; pseudorandom outputs are produced at "out [16:1]" outputs
1	1	1	0/1	1	Parallel signature analyzer mode; parallel data vectors at "d[16:1]" are compressed into a final signature at "out[16:1]" outputs
0/1	0/1	0/1	0/1	0	Shifting of data is disabled; "out[16:1]" outputs retain their previous clock cycle values

Enble = 0 latches the control inputs "C_1", "C_2", "C_3", "shiften", and "sin", and prevents further alteration of the latched values; "tout" assumes a high impedance state (Z) when "tctrl" = 0. When "tctrl" = 1, "tout" = "out[16]".

Incorporation of the BILBO Modules into Design

A hardware design will be partitioned into ambiguity groups (AG) of interconnected chips prior to the incorporation of the BILBO modules. The example in Figure 3 shows their incorporation in a design.

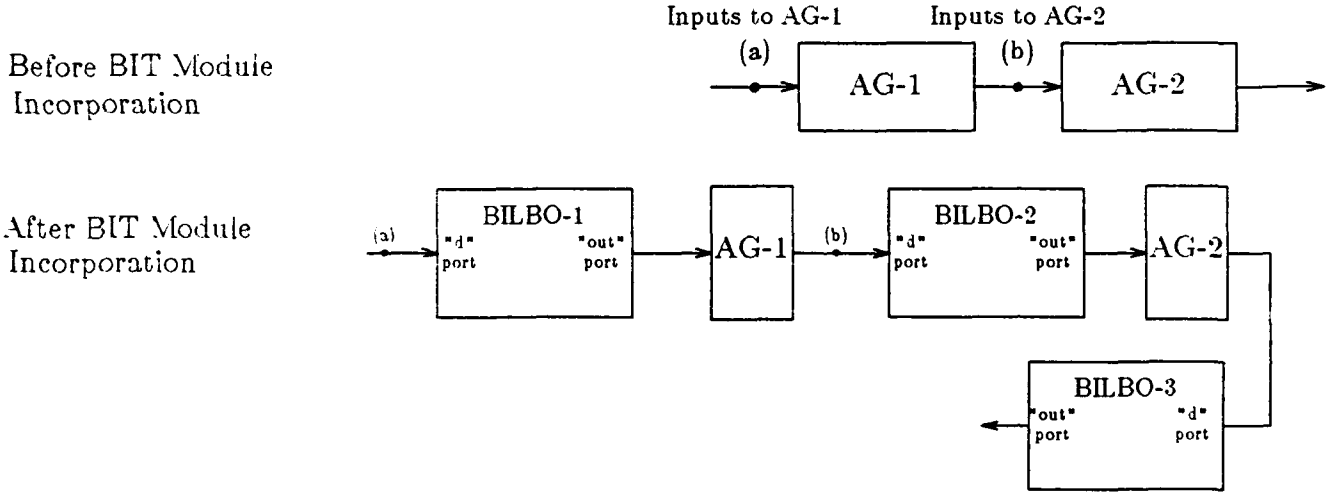


Figure 3. An Example Showing the Incorporation of the BILBO Modules in a Design

As evident from the figure, all inputs to an AG pass through a BILBO module and the AG outputs are connected to the "d" port of another BILBO module. Another example illustrating the incorporation of the BILBO modules is shown in Figure 4.

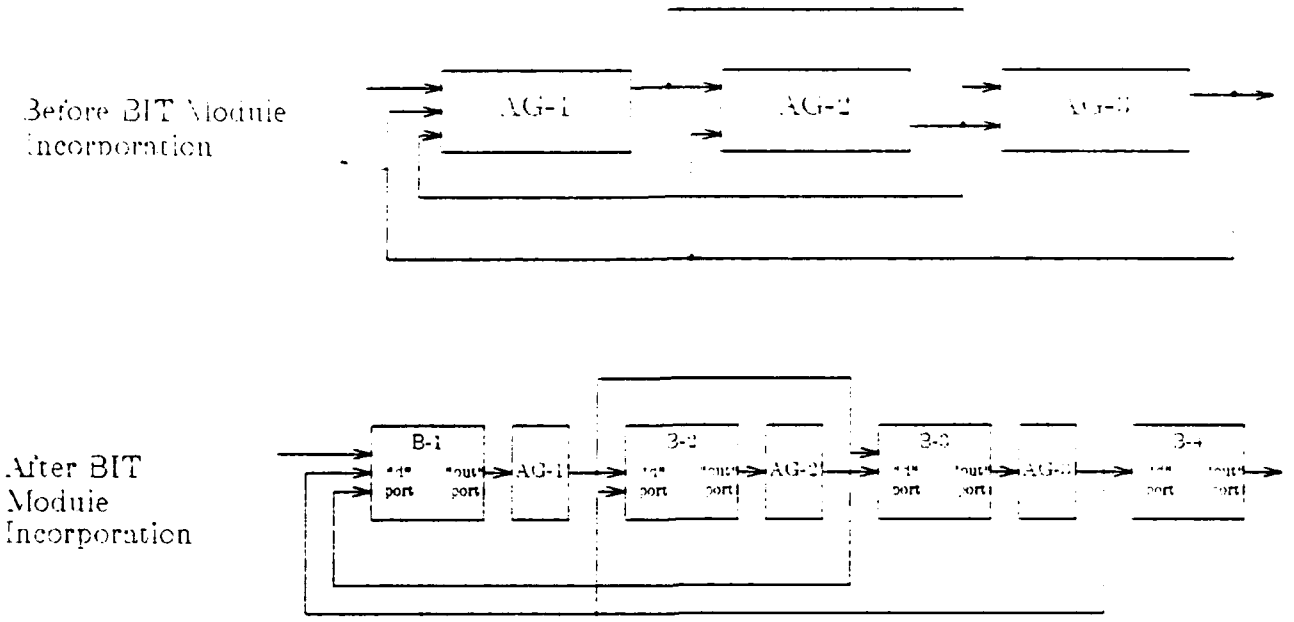


Figure 4. Another Example Illustrating the Incorporation of the BILBO Modules in a Design

A master test control unit (TCU) is assumed to be present in the system in order to perform the following functions:

1. Checking functionality of the BILBO modules:

2. Providing deterministic patterns to the AG under test, if necessary, by shifting in data through "sin" input of the relevant BILBO module(s);

3. Collecting test responses (i.e., final signatures) through "tout" nodes of the BILBO modules, performing the necessary comparison with the expected signatures, and evaluating the status of the AGs;

4. Providing all the necessary control signals such as "C_1", "C_2", "C_3", and "enble" for the BILBO modules;

5. Exercising proper control over the AG clocks and the BILBO module clocks tphi1/tphi2.

Before discussing a step-by-step procedure for testing the AGs, the following issues are to be considered:

1. Incorporation of the BILBO modules into a design introduces additional latency of one clock cycle between AGs. Use of BILBO modules is not recommended for fault isolation testing when such a latency is not acceptable during normal operation.

2. It may be preferable to treat the control inputs of an AG differently from its data inputs and use different BILBO modules in conjunction with control and data inputs, as shown in Figure 5.

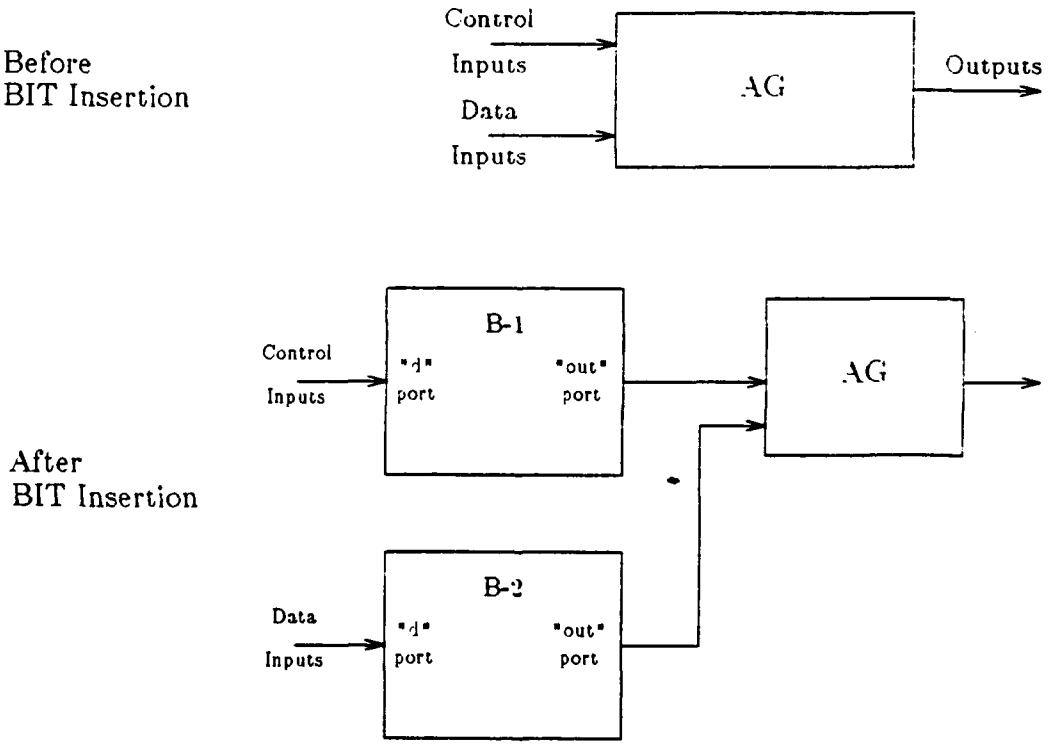


Figure 5. An Example Illustrating the Use of Different BILBO Modules for Control and Data Inputs of an AG

This feature facilitates the application of deterministic test patterns at the control inputs of an AG while allowing pseudorandom patterns to be applied at its data inputs. Even though the logic diagram shown for the BILBO module assumes a word size of 16 bits, in practice, modules with smaller word sizes (4 bits, 8 bits, etc.) may also be utilized in order to reduce the unused pins of the BIT modules and hence possibly save some board space.

It is also conceivable to pass only the data inputs of an AG through a BILBO module while allowing the control inputs to be directly connected, as shown in Figure 6.

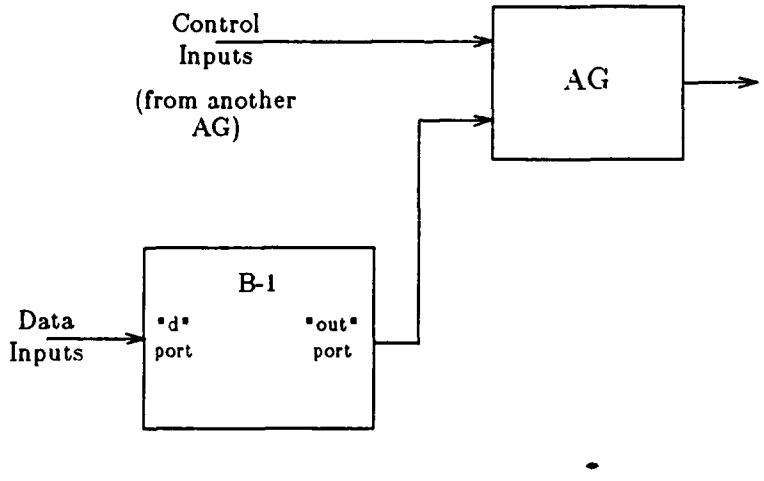


Figure 6. An Example in which a BILBO Module is Used Only for the Data Inputs of an AG

In such a scheme, it is generally assumed that the AGs that generate the control inputs are tested first, and their outputs are set to desired values during the test

of other AGs. The feasibility of properly implementing this scheme depends on several factors such as the design of AGs in the system, required fault isolation resolution, etc., and this scheme is not considered in the step-by-step procedure given subsequently.

3. If the number of inputs or outputs of an AG exceed the default word size of the BILBO module, multiple BILBO modules need to be used. The BILBO modules shown in Figure 2 cannot be cascaded to form an equivalent larger word-size module. Hence, separate BILBO modules per each set of AG I/O need to be used. This is not a limitation in general, since the BILBO module in Figure 2 uses a primitive feedback polynomial to give a maximal repetition interval for the generated test vectors [1]. Furthermore, uncorrelated test vectors can be generated by initializing the various BILBOs in the system with different seeds (initial test patterns). Also, the error escape probability, when the BILBO module is used as a signature analyzer, is acceptably low ($\leq 2^{-16}$) for a 16-stage BILBO module.

A Procedure for Testing of AGs

Preliminary Steps

1. Reset all the BILBO modules in the design by applying "reset" and "reset1" signals.

2. Bring all the AGs into predetermined and well-defined states by means of global control signals such as "reset" and "preset", and then disable the AG clocks from further altering their state.

3. Select each BILBO module by means of its "enble" signal, shift-in predetermined data (e.g., alternating ones and zeros) through "sin" input under the control of tphi1/tphi2 clocks, and verify the output data at "tout" node. Then reconfigure the BILBO as test pattern generator ($C_{1i} = 1$, $C_{2i} = 1$, $C_{3i} = 0$, shiftl = 1) and verify the output data at "tout" for a predetermined number of clock cycles. If there is an error in the scanned out data, either the BILBO module itself, or the AG connected to its outputs, or both could be faulty. The first choice for replacement in this case would be the BILBO module rather than the AG since the AG will be checked in subsequent tests.

4. Perform a parallel transfer of data from "d[16:1]" to "out[16:1]" in every BILBO module, and verify the stored data by serially shifting it out through "tout". In this context, both reset and preset capabilities of AGs may be required so that "d[16:1]" nodes can assume both logic one as well as zero values. If the scanned out data is erroneous, fault isolation can be achieved to the combination of an AG, a BILBO module, and the inherent interconnections under the single faulty AG assumption.

AG Testing

5. Shift in a predetermined control data pattern into the BILBO module associated with the control inputs of the AG under test; then, by disabling further shifting ("shiftl" = 0), the AG is set up for test under a known control mode.

6. Configure the BILBO modules associated with the data inputs of the AG under test as test pattern generators and those associated with the AG outputs as signature analyzers. It may be required to hold some or all of the remaining AG outputs at predetermined states during this test so that any faults internal to the AGs not under test do not affect the signatures relevant to the AG under test. Pseudorandom patterns generated by the test pattern generators are automatically applied to the AG inputs, and the AG responses are compressed by the signature analyzers. During this test, the AG clocks are assumed to bear appropriate frequency and phase relationships with the test clocks tphi1/tphi2.

   After a predetermined number of clock cycles, apply "reset1" signal and disable shifting in all the BILBOs. Then, select each relevant BILBO operating as a signature analyzer and shift out the stored signature through "tout" node. Compare this signature with the expected one, obtained a priori from simulation, and determine the status of the AG.

7. Repeat steps 5 and 6 for different control modes of the AG.

The above AG testing procedure is applied sequentially to all the AGs, i.e., the AGs are tested one after another using steps 5 through 7. If the test of an AG fails, fault isolation is generally to the combination of the AG under test, the BILBO modules associated with its I/O, and the inherent interconnections.

Timing Example

The mode control signals, "C_1", "C_2", and "C_3", have to be valid one clock cycle prior to the parallel data at "d[16:1]". For instance, parallel transfer of data from "d[16:1]" to "out [16:1]" and its serial shifting through the "tout" node involves the timing sequence shown in Figure 7.

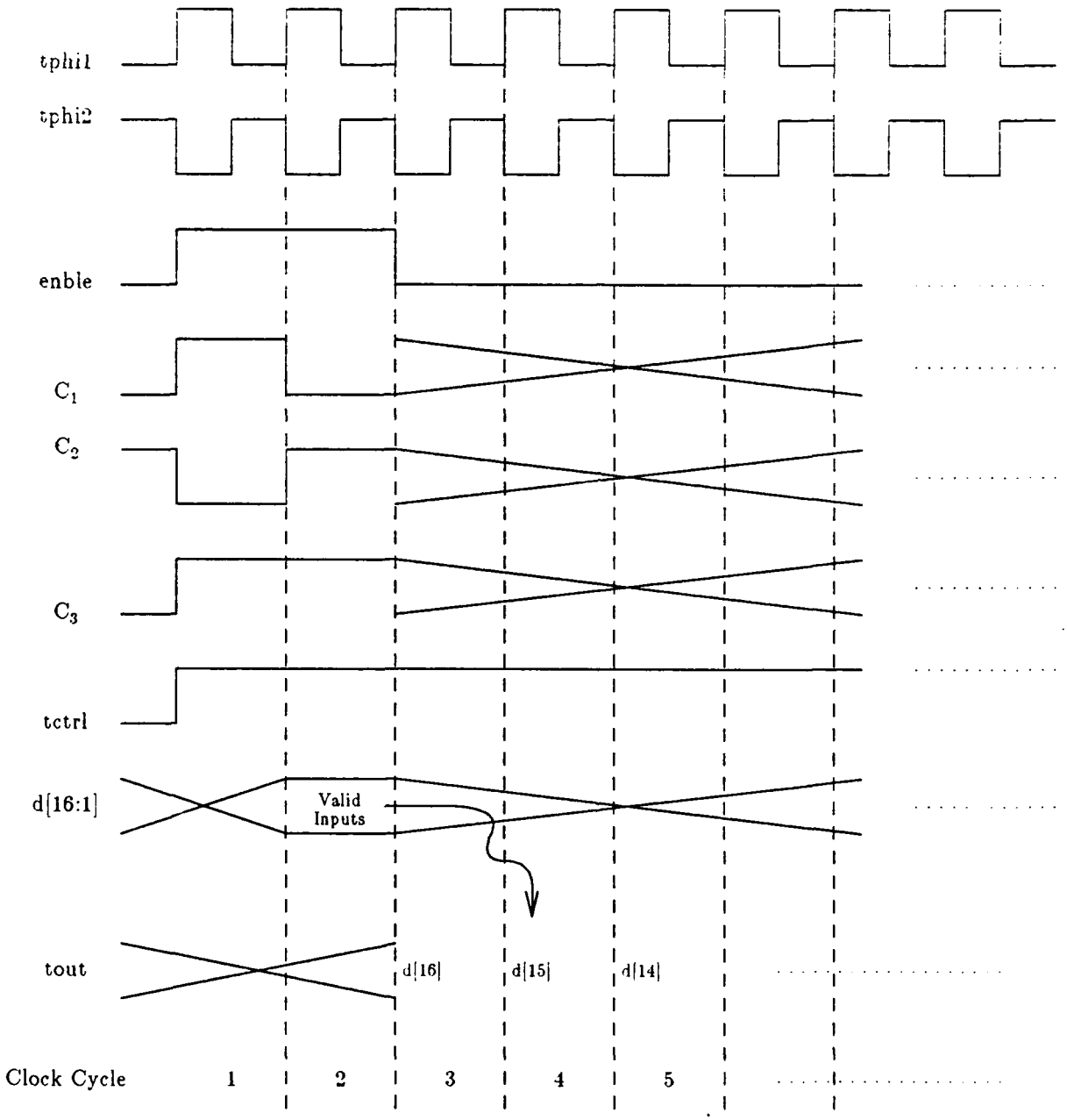


Figure 7. A Timing Diagram Illustrating the Operation of the BILBO Module

As can be inferred from Figure 7, the control mode, "C_1" = 1, "C_2" = 0, and "C_3" = 1, is set up one clock cycle before the data at "d[16:1]" inputs is valid. The parallel data at "d[16:1]" inputs is available at the "tout" node beginning at clock cycle 3 in Figure 7.

Reference

[1] Smith, J. E., "Measures of Effectiveness of Fault Signature Analysis." *IEEE Transactions on Computers* C-29, no. 6, June 1980, pp. 510-514.

MAINTENANCE NODE

APPLICATION NOTE

Objective and Overview

This application note provides a functional and structural description of a maintenance node (MN) unit and illustrates its use in controlling the in-system board test. Maintenance node is an interface module between a system-level test control unit (TCU) and the board under test, as shown in Figure 1.

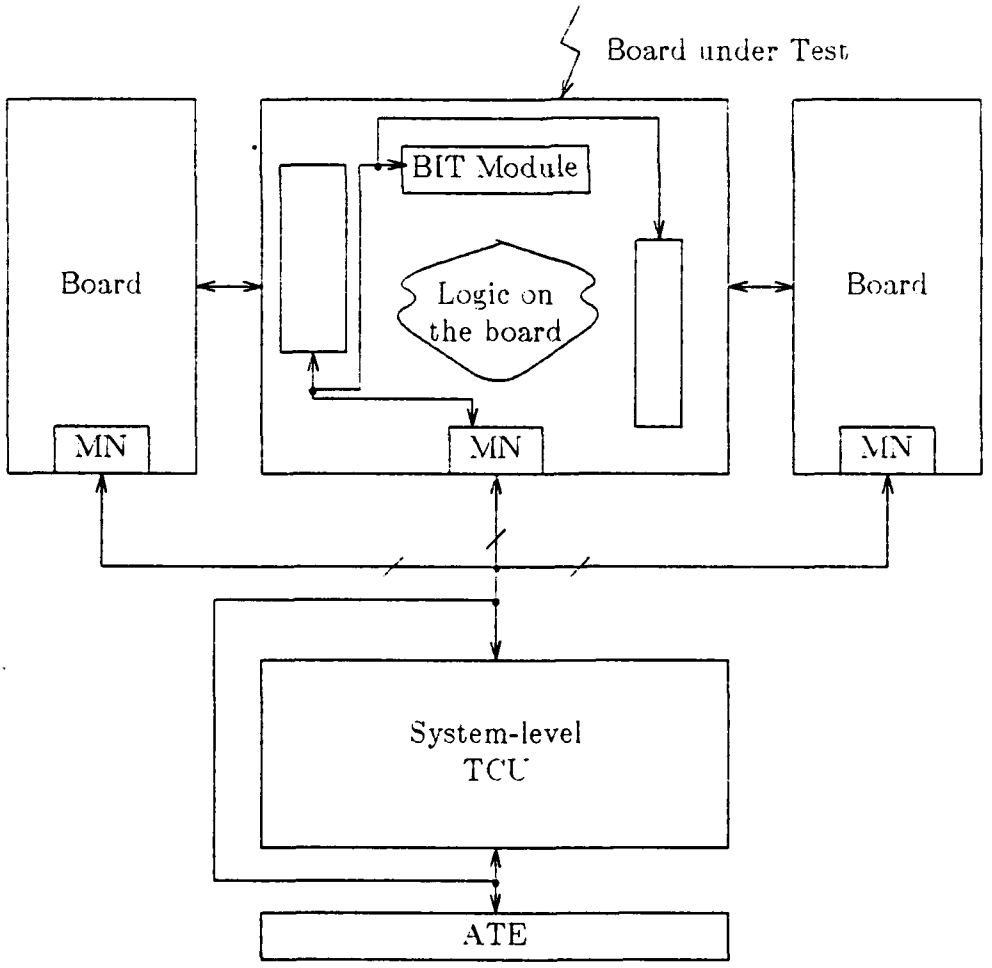


Figure 1. An Example Showing the Maintenance Node Unit as an Interface Module

It delivers required clock and control signals to the board, provided by the TCU; it transfers test input data from the TCU to the logic on the board through BIT modules; and finally, the maintenance node transfers test output data from the board to the TCU for analysis.

Maintenance Node–TCU Interface

There are ten (10) signal lines between the TCU and the MN of each board, as shown in Figure 2, out of which three (3) are clock signal lines. The TCU generates all these required clock signals from a single master clock source present in the TCU, and hence synchronization among the three clock signals is easily accomplished. The required phase and frequency relationships among the three clock signals are dependent on both the system application under consideration and the BIT technique used, and they will be addressed only to the limited extent needed to clarify the presentation in this application note.

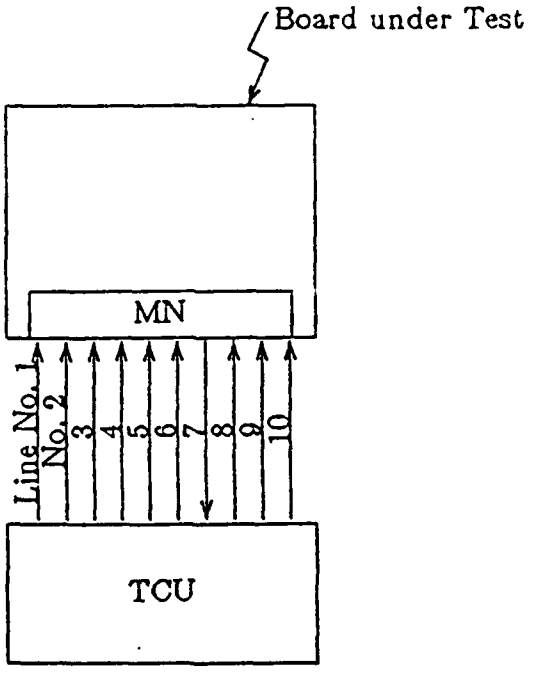


Figure 2. MN-TCU Interface Showing the Signal Lines

The functions of these signal lines are illustrated in the following sequence:

Line No. 1: BIT module clock (tphi)

The TCU provides a clock signal on this line that is processed by the MN to derive two non-overlapping clock phases for use by the BIT modules on the board. The BIT modules are assumed to have been designed using a two-phase non-overlapping clocking methodology.

Line No. 2: Auxiliary (test mode) system clock enable (aux_enab)

The TCU activates this signal line if the normal mode system clock is to be disabled and an auxiliary system clock is to be used for testing the board.

Line No. 3: Auxiliary system clock (aux_phi)

The TCU provides a clock signal on this line that is processed by the MN to derive a required number of clock phases for use by the logic on the board. The phase and frequency relationships among the individual clock phases are dependent on the system design and they are not addressed in this application note. The TCU has the responsibility of preserving the system-state by deactivating the auxiliary clock (which is generally accomplished by properly gating the clock signal generated within the TCU) for given periods of time during test when necessary.

Line No. 4: Control, data, and id (CDI) clock (cdi_phi)

The TCU provides a clock signal on this line that is processed by the MN to derive two non-overlapping clock phases for use by certain registers of the MN and of the BIT modules on the board.

Line No. 5: BIT module control and data in (cd_in)

The TCU provides the required BIT module control and data signals on this line in a bit-serial format controlled by the CDI clock.

Line No. 6: Identification and command input (id_in)

The TCU provides an identification and command code on this line in a bit-serial format to enable a given BIT module on the board to accept BIT control and data signals provided by the TCU. This code also facilitates functions such as reset of certain registers of the MN, reset of the logic on the board, and selective reset/preset of each ambiguity group (AG) output set.

Line No. 7: Data out

The TCU receives test output data from the board on this line in a bit-serial format controlled by the CDI clock.

Line Nos. 8, 9, and 10: Load address inputs (laddr[2:0])

The TCU provides three address bits on these lines to enable selective transfer of BIT module control, data, and identification (id) related signals stored in the MN registers to the MN outputs on the MN-board interface. These address bits also serve to provide the required enable signal for parallel latching of the test output data at the MN-board interface into an MN internal register before being serially transmitted on the "dataout" line.

Maintenance Node Architecture and Data Formats

An overall organization of the maintenance node is shown in Figure 3. Several variations of this basic architecture are possible and they have to be evaluated in specific situations where system, board, and chip level design details are completely known. For instance, the auxiliary clock phases may be generated within the logic of the board rather than in the maintenance node, and in such a case, only the auxiliary clock and auxiliary clock enable signal lines will be distributed to the logic on the board rather than the individual clock phases. Furthermore, integrating the MN logic into each and every BIT module is an option that may be considered since it reduces the interconnection complexity on the board. The feasibility of implementing this option depends, however, on the BIT module I/O limitations and the required versatility of the designs. In this application note, the specifics of the MN design are chosen so that the resulting MN is compatible with the various board-level BIT modules designed at RTI.

1) BIT Control and Data Register

The main purpose of this register is to provide the required control and data signals to BIT modules on the board. This is a 16-bit register with 3 bits reserved for BIT module data signals and 12 bits for control signals, as shown in Figure 4. The remaining 1 bit may be either left unused or used in a user-defined manner as a control signal. The datain_load, control_load, MN reset, and CDI clock phases are generated within the maintenance node. The shift register contents are transferred to CD[16:1] outputs only when datain_load and/or control_load signals are high.

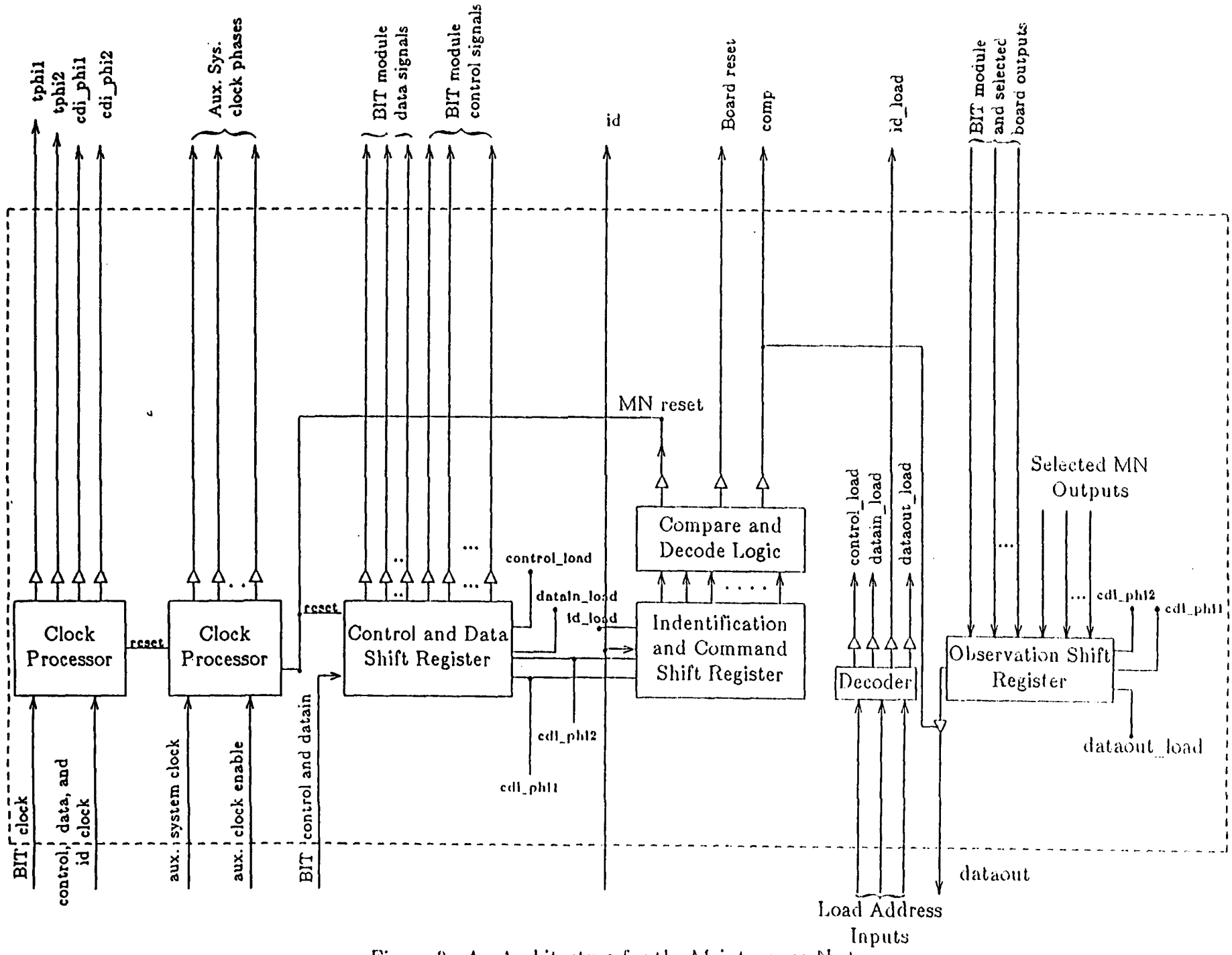


Figure 3. An Architecture for the Maintenance Node

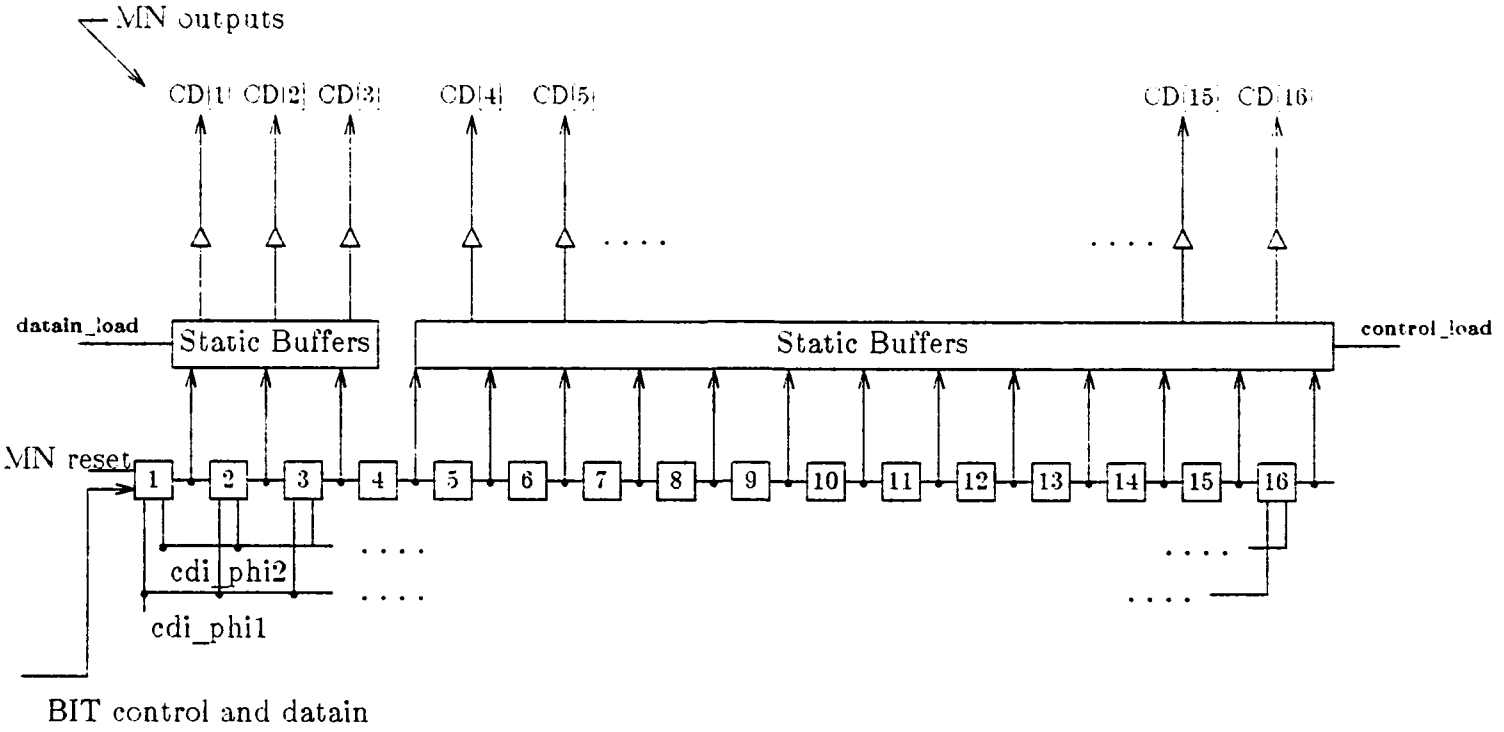


Figure 4. Control and Data Register of the Maintenance Node

Table 1 shows a suggested MN output (CD[15:1]) and BIT module input correspondence. The BIT modules considered in the table are the specific ones that have been described in other application notes. Note that there is considerable flexibility in this assignment, and the choice in Table 1 was made so that certain combinations of BIT modules present on a board can be individually reset. For instance, if the assignment in Table 1 is used, "switch" module contents are not affected when resetting the "scan-set" BIT modules on the board, and vice versa. Furthermore, the MN output CD[1] was assigned to a BIT module data input that needs to be updated most often, in order to minimize the latency in updating the data.

Table 1. A Suggested MN Output/BIT Module Input Correspondence

	BIT MODULE ON THE BOARD					
MN OUTPUT	TESTING SWITCH	SCAN-SET	BILBO	EEXOR TPG	IEXOR TPG	Any other BIT Module
CD[1]	gendatain	scanin	sin	sdata	sdata	
CD[2]	fbcoefin	scanin1	—	fbin	fbin	
CD[3]	sasin	—	—	—	—	
CD[4]	reset	—	—	reset	reset	
CD[5]	reset1	—	—	—	—	
CD[6]	reset2	—	—	—	—	
CD[7]	genctrlin	—	—	—	—	
CD[8]	genshiften	—	shiften	—	—	
CD[9]	sac1in	c_1	c_1	fbctrl	fbctrl	User-defined
CD[10]	sac2in	c_2	c_2	shiften	shiften	
CD[11]	sashiften	—	c_3	—	—	
CD[12]	ctrl1in	—	reset	—	—	
CD[13]	ctrl0in	—	reset1	—	—	
CD[14]	—	reset	—	—	—	
CD[15]	—	inh	—	—	—	

1) Tristate control inputs are tied to BIT "enable" inputs

2) — ⇒ Unused/User-defined

2) Identification Register and the Associated Logic

The main purpose of this logic is to determine whether or not the board associated with its MN is being addressed by the TCU, and to generate "MN reset" and "board reset" signals. Generation of BIT module "enable" signals and AG reset/preset signals is assumed to be performed either within the BIT modules or by additional logic associated with each AG, even though it is controlled by the MN. This assumption was made to limit the output requirements of the maintenance node at the MN-board interface to a reasonable value.

This logic consists of mainly a 24-bit shift register, two comparators, and a decoder, as shown in Figure 5. It generates "board reset," "comp," and "MN reset" signals by comparing the selected bits of the shift register with the hardwired identification (id) values and by decoding the command code. All the identification and command codes from the TCU are sent MSB first. Bits 1 to 5 constitute the subsystem (i.e., a collection of boards) id in which the board under test is located; whereas the id indicated by bits 6 to 11 is that of the board under test. Hence the maintenance node in this design can be used to address up to 31 subsystems and 63 boards per subsystem (all zero id is not used). Bits 12 to 14 constitute the MN/board reset code as indicated in Table 2.

Table 2. MN/Board Reset Code

Bits	14	13	12	Function
	0	0	1	Reset only the MN associated with the addressed board
	0	1	0	Reset all boards in the system
	0	1	1	Reset only the addressed board
	1	0	0	Reset the address board and the MN associated with it
	1	0	1	Reset all MN in the system
	1	1	0	Do none of the above
	1	1	1	
	0	0	0	

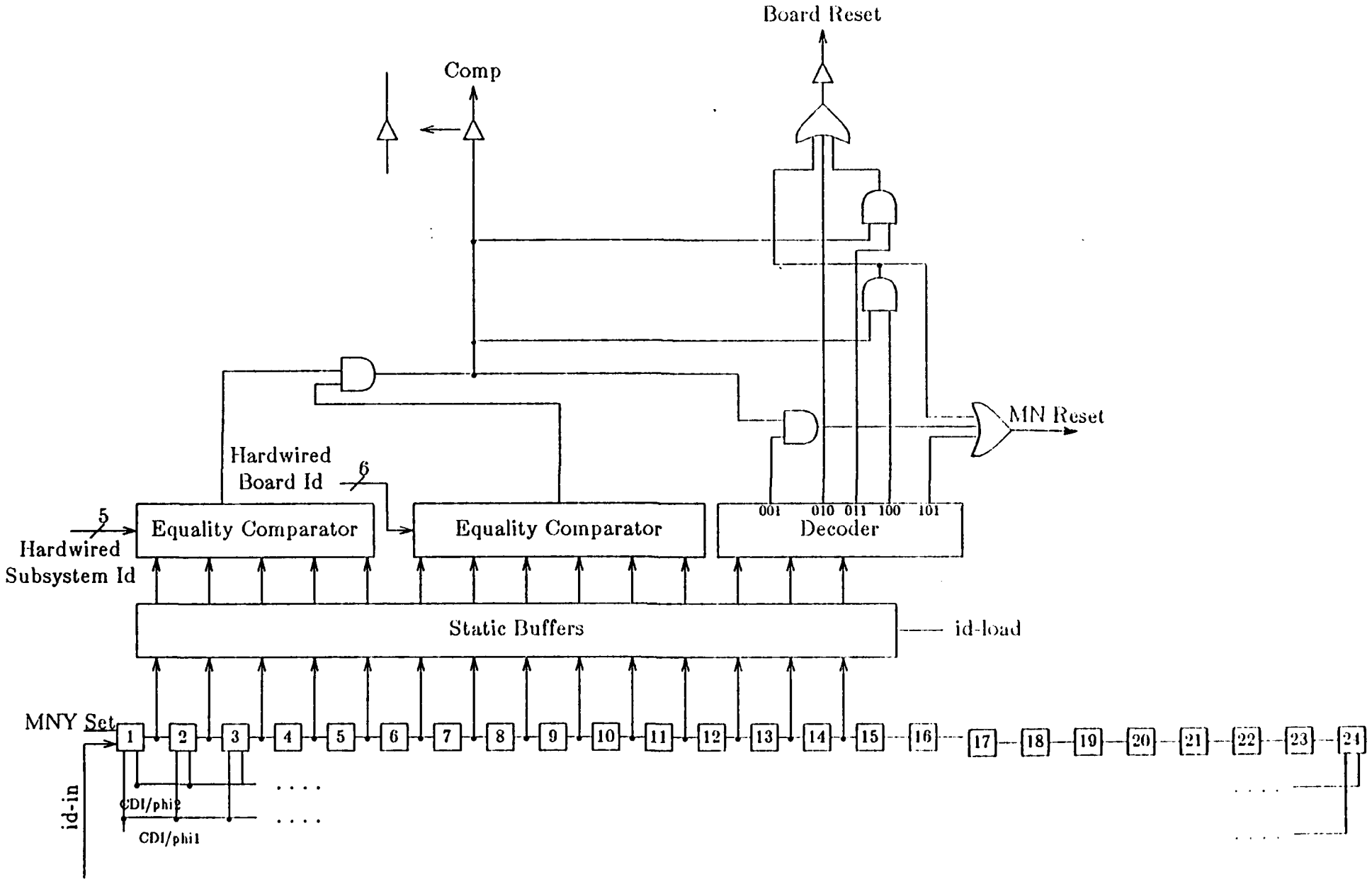


Figure 5. Identification and Command Code Shift Register and the Associated Logic

The id register contents are transferred to the compare and decode logic only when the id_load signal is high.

Bits 15 to 19 constitute the AG identification code, whereas bits 22 to 24 denote the BIT module id corresponding to an addressed AG. This implies that the MN in this design can be used to address up to 31 ambiguity groups per board and 7 BIT modules per AG. Bits 20 and 21 form the command code with interpretation as shown in Table 3.

Table 3. AG Reset/Preset Command Code

Bits 21	20	Function
0	1	Preset the addressed AG outputs
1	0	Reset the addressed AG outputs
1	1	Do none of the above
0	0	

Even though the data associated with bits 15 to 24 are utilized either in the BIT modules or in the AGs of the board and not in the MN itself, these bits are included in the id shift register to simplify the I/O and timing requirements at the MN-board interface. The logic associated with the BIT modules on the board to generate their own "enable" signals and the AG reset/preset signals is shown in Figure 6. If BIT modules do not have this logic and instead the "enable" signal line is a primary input, then each AG will have logic, as shown in Figure 7, to generate the required "enable" and AG preset/reset signals.

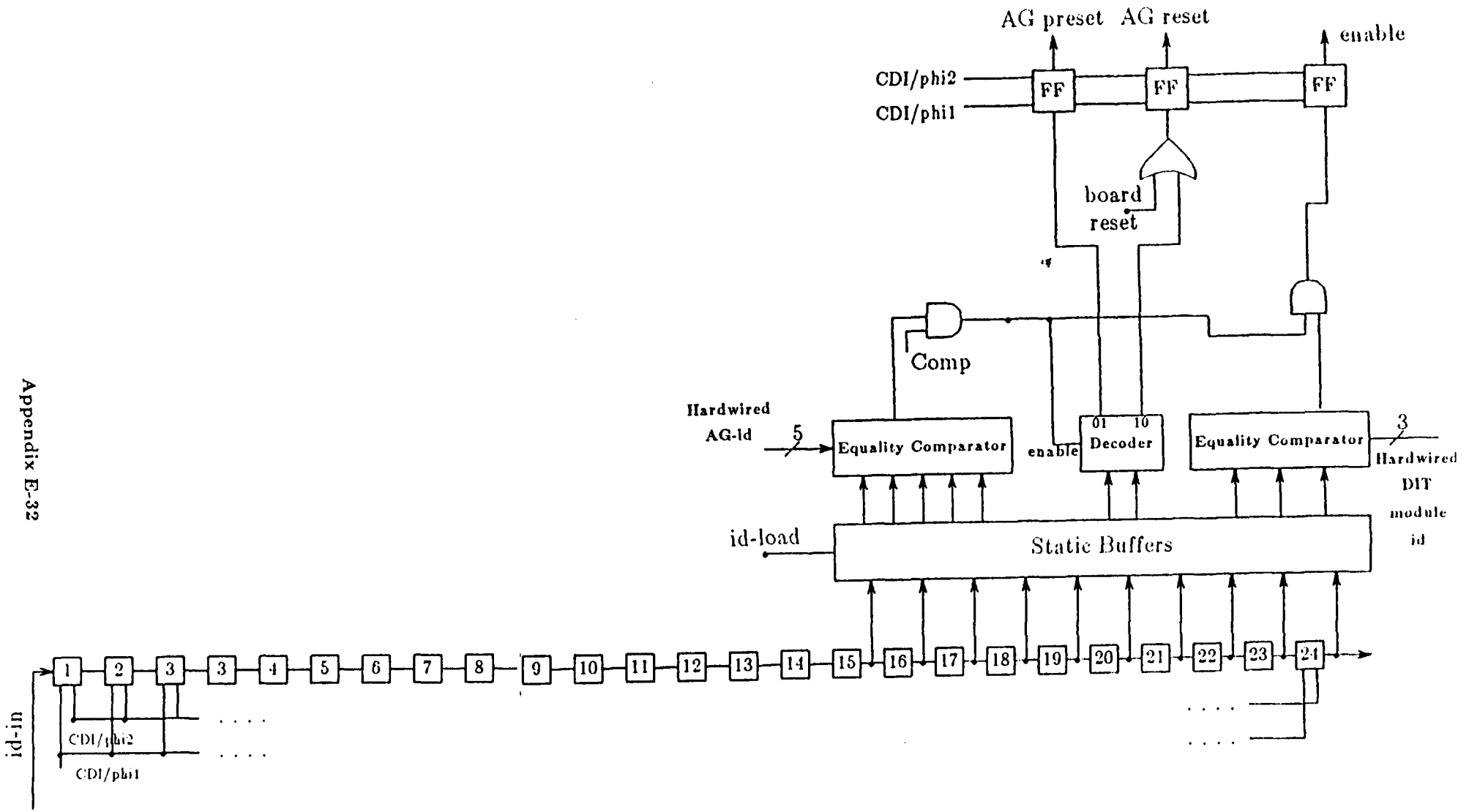


Figure 6. Logic Associated with BIT Modules to Generate the Required Test Signals

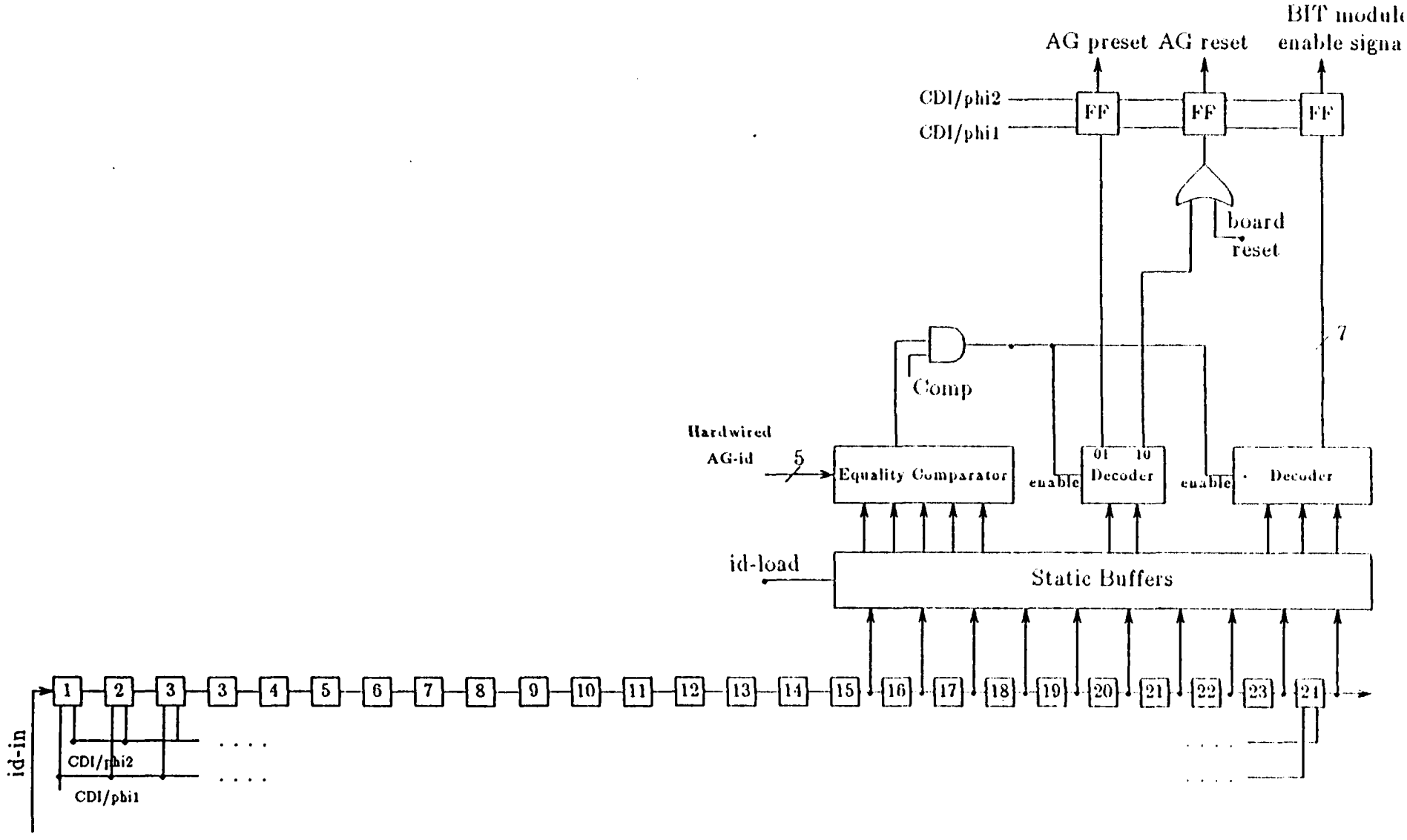
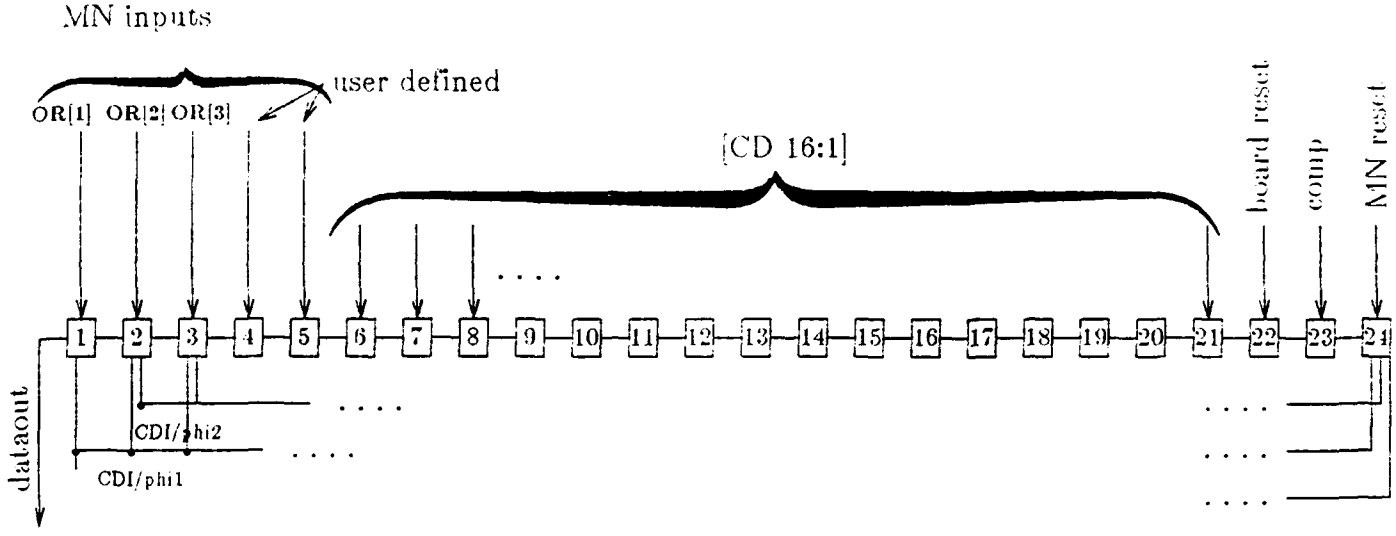


Figure 7. Logic Associated with the AGs to Generate the Required Test Signals

Note that the AG preset/reset signal lines are to be connected in an appropriate manner to the preset/reset inputs of the AGs on the board.

3) **Observation Register**

The purpose of this register is to convert the parallel test output data available at the MN-board interface to a serial format for observation at the "dataout" line on the MN-TCU interface. This is a 24-bit register with parallel inputs, as shown in Figure 8.



dataout-load
1 = Parallel input shift mode
0 = Serial input shift mode

Figure 8. The Observation Register in the Maintenance Node Unit

The "dataout_load" signal, when high, allows parallel latching of the inputs into the observation register. When it is low, the latched data can be shifted out serially. A suggested MN input (OR[3:1]) and BIT module output correspondence is shown in Table 4 for specific BIT modules considered in Table 1. The MN input

OR[1] was assigned to a BIT module data output that needs to be monitored most often, to minimize the latency in monitoring the data.

Table 4. A Suggested MN Input/BIT Module Output Correspondence

MN INPUT	BIT MODULE ON THE BOARD: TESTING SWITCH	SCAN-SET	BILBO	EEXOR TPG	IEXOR TPG	Any Other BIT Module
OR[1]	satout	scanout	tout	tout	tout	—
OR[2]	—	scanout1	—	fbout1	fbout1	—
OR[3]	—	—	—	lfout1	—	—

— ⇒ Unused/User-defined

4) **Load Address Decoder**

The purpose of this logic block is to generate the required "id_load," "datain_load," "control_load," and "dataout_load" signals. The chosen decode scenario is shown in Table 5.

Table 5. Load Address Decoder Truth Table

Lines	8	9	10	control_ load	datain_ load	dataout_ load	id_ load
	0	0	0	0	0	0	0
	0	0	1	0	0	0	1
	0	1	0	0	0	1	0
	0	1	1	0	1	0	0
	1	0	0	1	0	0	0
	1	0	1	1	1	0	0
	1	1	0	0	1	1	0
	1	1	1	1	1	1	1

The decoding function shown in Table 5 allows for independent activation of each load signal as well as certain useful combinations of various load signals.

Utilization of the Maintenance Node in Controlling the In-System Board Test

This section assumes that the board under test utilizes one of the four board-level BIT techniques given below, implemented using the BIT modules described in other RTI application notes. The BIT techniques considered are as follows:

1. Pseudorandom pattern application and response compression technique using either "BILBO" or "Testing-Switch" modules that are distributed over the board:
2. Board-level scan-set technique using scan-set BIT modules that are distributed over the board:
3. Test point monitoring on a cycle-by-cycle basis, while applying either deterministic or pseudorandom patterns at the board primary inputs. Pseudorandom patterns are assumed to be generated by using the BIT modules such as "Testing-Switch," "BILBO," and other available test pattern generator modules;
4. Test point monitoring with compression of the test output results, while applying either deterministic or pseudorandom patterns at the board primary inputs. Compression of the test output results is generally assumed to be performed by using "BILBO" modules as parallel signature analyzers.

There are numerous variations of implementing each of the above BIT techniques on a board, and consideration of all such variations is beyond the scope of this application note. Hence, only certain specific implementations will be considered, and the use of the MN in controlling the in-system go/no-go board test will be illustrated. Controlling the fault-isolation tests to an AG of the board can also be performed using the

MN. It is equivalent to administering a collection of go/no-go tests on the individual AGs rather than on the entire board, and hence, fault-isolation tests will not be separately considered in this section. One should note, however, that fault-isolation tests are generally performed only after the defective board(s) is(are) taken out of the system, and they may be controlled by the ATE rather than the system level TCE.

Example 1

Consider a board consisting of three ambiguity groups of chips and using the pseudorandom pattern application and response compression BIT technique implemented using the "testing-switch" modules, as shown in Figure 9. For details on the "testing-switch" modules and how to use them in a design, the reader is referred to the relevant application note. Even though a go/no-go test is the subject of this example, Figure 9 also shows the test logic incorporated to perform fault-isolation to an AG for off-the-system board maintenance purposes.

It is assumed that the switch modules have internal "enable" generation logic shown in Figure 6, and the AG "reset" and "preset" signals in Figure 6 can be used to reset and preset each AG output set independently of another AG if necessary. The connections indicated in Tables 1 and 4 are assumed to have been made between each of the switch modules and the MN I/O at the MN-board interface. The AGs are assumed to be synchronous in general, even though portions of asynchronous logic can also be handled. The system clock is wired such that when the "auxiliary clock enable" line is activated it is disabled and the auxiliary clock controls the board operation. It is further assumed that whenever the auxiliary system clock is held at low

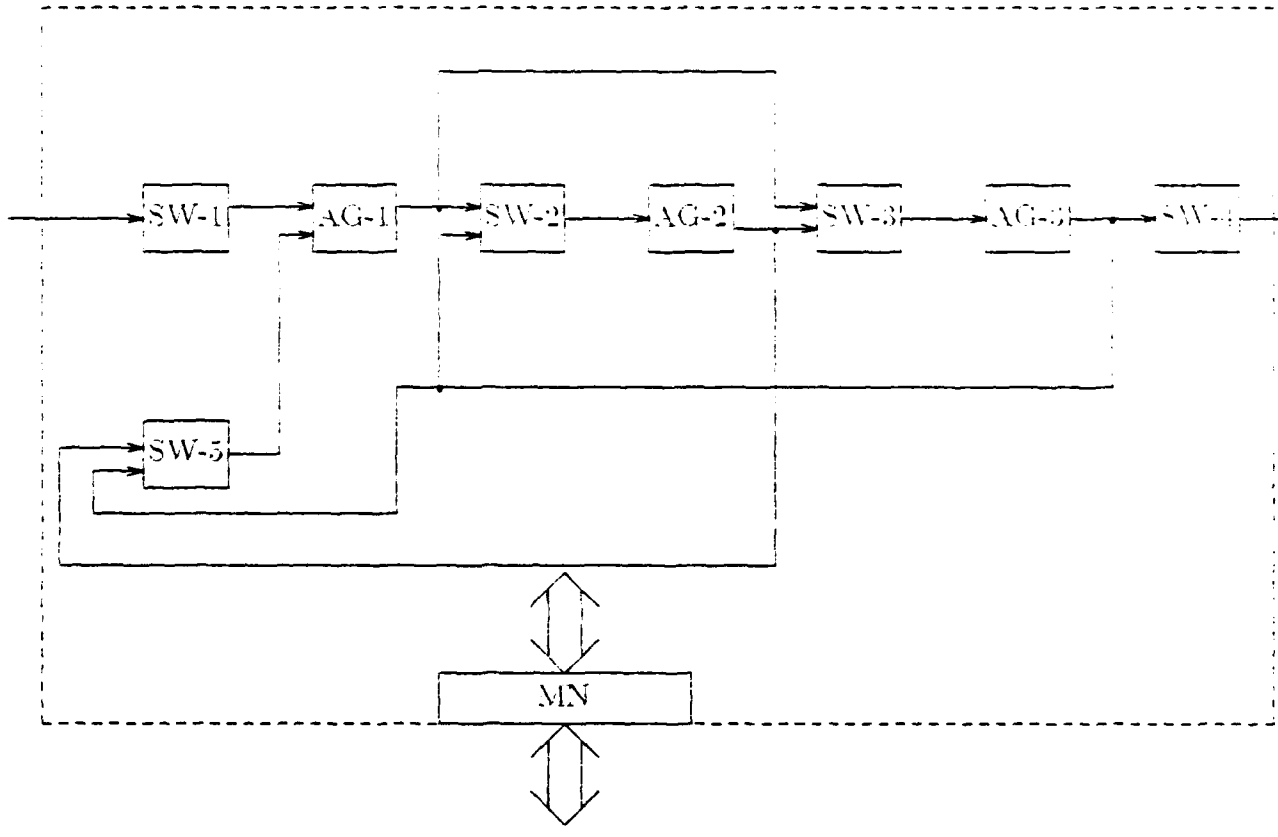


Figure 9. A Board Using the "Testing-Switch" BIT Modules

level and the AG reset/preset signals are not active, the AGs retain their previous state indefinitely. A similar assumption holds for the BIT modules on the board as well. Finally, a BIT module-AG assignment shown in Table 6 is assumed for the purpose of controlling the test.

Table 6: A BIT Module-AG Assignment for the Network in Figure 9

Bit Module(s)	**Assigned AG**
SW-1 and SW-5	AG-1
SW-2	AG-2
SW-3 and SW-4	AG-3

A Procedure for Administering a Go/No-go Test of the Board in Example 1

The following steps are generally performed by the system level TCU. Steps 1 and 2 are preliminary steps that ensure proper clock control and MN and board reset before the actual tests can be performed. Steps 3 to 12 accomplish the switch module testing, while steps 13 to 16 set up the board for a go/no-go test. The test vectors generated by the relevant switch modules are applied to the board and the output results are compressed into signatures in step 17. Finally, comparison of the actual signatures with the expected ones and determining the status of the board are accomplished in step 18. Further details on how these steps are performed are provided in the following paragraphs.

1. Activate the "auxiliary clock enable" line and hold the auxiliary clock and BIT clock lines at low level by proper gating of the clock signals generated within the TCU.

2. Shift in a 24-bit id and command word containing the appropriate 5-bit subsystem id, 6-bit board id, 3-bit command code indicating the specific MN and board reset, and 2-bit AG reset/preset code corresponding to no AG reset or preset. This word has to be shifted in through "id_in" input under the control of CDI clock. Arbitrary values may be used for the 5-bit AG id and the 3-bit BIT module id for this step. After 24 CDI clock cycles, apply the load address bits such that only the "id_load" signal is high for one CDI clock cycle. This step ensures that the necessary MN and board reset is accomplished. Raise the "dataout_load" signal to high level, which configures the observation register in the MN to a

parallel-latch mode of operation for one CDI clock cycle. Then serially shift out the register contents ("dataout_load" = 0) and verify the MN and board reset bits that appear on the "dataout" line at the MN-TCU interface.

3. Repeat step 2 with another 24-bit word containing the 3-bit command code to deactivate the MN and board reset signals. Also, the 5-bit AG id corresponding to a given AG and the 3-BIT module id corresponding to a given switch are used in the 24-bit word. This step results in the "enable" of a given switch module being high, which in turn sets up the switch to receive control and data signals from the MN.

4. Shift in a 16-bit control and data word through the "BIT control and data" line containing all zeros except "reset" = 1 and "reset1" = 1. After 16 CDI clock cycles, raise the "control_load" and "datain_load" signals to high level for one CDI clock cycle. This step ensures that the reset of *all* the switch modules on the board is accomplished.

5. Repeat step 4 with another 16-bit word containing all zeros except "ctrl1in" = 1 and "ctrl0in" = 1 to enable a test path within the chosen switch module. Then apply the BIT module clock for one clock cycle so that the control signals are latched in the chosen switch.

6. Shift in 16-bit control and data words containing the appropriate control and data bits required for initialization of the test pattern generator and the signature analyzer within the chosen switch. The initialization process consists of repeated shifting in of a 16-bit word, raising "control_load" and/or "datain_load" signals

to high level for one CDI clock cycle after every 16 CDI clock cycles, and applying the BIT module clock for one clock cycle. At the end of the initialization process, the signature analyzer in the chosen switch is configured for parallel data compression mode of operation ("sac1in" = 1 and "sac2in" = 1).

7. Repeat step 3 with "all zero" BIT module id, which results in the "enable" of the chosen switch being low. The purpose of this step is to prevent further alteration of the latched control signals in the chosen switch.

8. Apply the BIT clock for a predetermined number of clock cycles. During this step, the test patterns generated in the switch are compressed by the signature analyzer within the switch. At the end of this step, hold the BIT clock at low level by proper gating of the clock signal.

9. Repeat step 4 with "reset1" = 1 ("reset" = 0). This step disables shifting in all the signature analyzers of the switches and thus locks the resulting signature in the chosen switch.

10. Repeat step 3 with the appropriate id. which results in the "enable" of the chosen switch being high.

11. Repeat step 4 with "sashiften" = 1 and "sac2in" = 1 (all other bits zero). This action enables serial shifting in the signature analyzer of the chosen switch. Raise the "dataout_load" signal to high level. Then apply the BIT clock in synchrony with the CDI clock. Under this mode, the "satout" output of the chosen switch is connected to the "dataout" line with one CDI clock cycle latency. Monitor the signature at the "dataout" line and compare it with the expected one, obtained a

priori from simulation. This step checks the functionality of the switch to a good extent. At the end of this step, hold the BIT clock at low level by proper gating of the clock signal.

12. Repeat steps 3 to 11 enough number of times with the 5-bit AG id and the 3-bit BIT module id, so chosen as to complete the testing of all the switch modules on the board.

13. Repeat step 4 so that all the switch modules are reset again. This step ensures that all the switches on the board are in normal "pass through" mode of operation.

14. Repeat step 3 and step 6 with the 5-bit AG id corresponding to AG-1 and the 3-bit BIT module id corresponding to SW-1. At the end of this step, the test pattern generator in SW-1 will be configured to supply pseudorandom test vectors to AG-1 primary inputs.

15. Repeat step 3 and step 6 with the 5-bit AG id corresponding to AG-3 and the BIT module id corresponding to SW-4. At the end of this step, the signature analyzer in SW-4 will be configured to accept AG-3 primary outputs for data compression.

16. Repeat step 7 to bring the "enable" of SW-4 to low level.

17. Apply the BIT clock and the auxiliary system clock in synchrony for a predetermined number of clock cycles. During this step, pseudorandom vectors generated in SW-1 are applied to AG-1 primary inputs, and AG-3 primary outputs are compressed in SW-4. At the end of this step, hold the BIT clock and the auxiliary system clock at low level by proper gating of the clock signals.

18. Repeat steps 9 to 11 with the 5-bit AG id corresponding to AG-3 and the 3-bit BIT module id corresponding to SW-4, and determine the status of the board.

An incorrect signature in SW-4 at the end of step 18 indicates a faulty board, while a correct signature in general implies a non-faulty board. However, the entire test procedure may have to be repeated several times and the consistency of the results must be verified to ensure high test confidence. Furthermore, there may be a need for performing additional tests involving multiple boards to completely check the interconnections between the board under test and the other boards. However, administering these additional tests differs from the above test procedure only in the id values that need to be used in different steps for the board, AG, and the BIT modules. It mainly involves configuring the switch modules on the periphery of the boards to either the test pattern generator or signature analyzer mode of operation, applying the BIT clock and/or auxiliary system clock for a predetermined number of clock cycles, and verifying the relevant signatures as already illustrated in the above test procedure for a single board.

Example 2

Consider a board consisting of three ambiguity groups of chips and using the board-level scan-set BIT technique, implemented using the scan-set BIT modules as shown in Figure 10. For details on the scan-set BIT modules and how to use them in a design, the reader is referred to the relevant application note. Even though a go/no-go test is the subject of this example, Figure 10 also shows the test logic incorporated to perform fault-isolation to an AG for off-the-system board maintenance purposes.

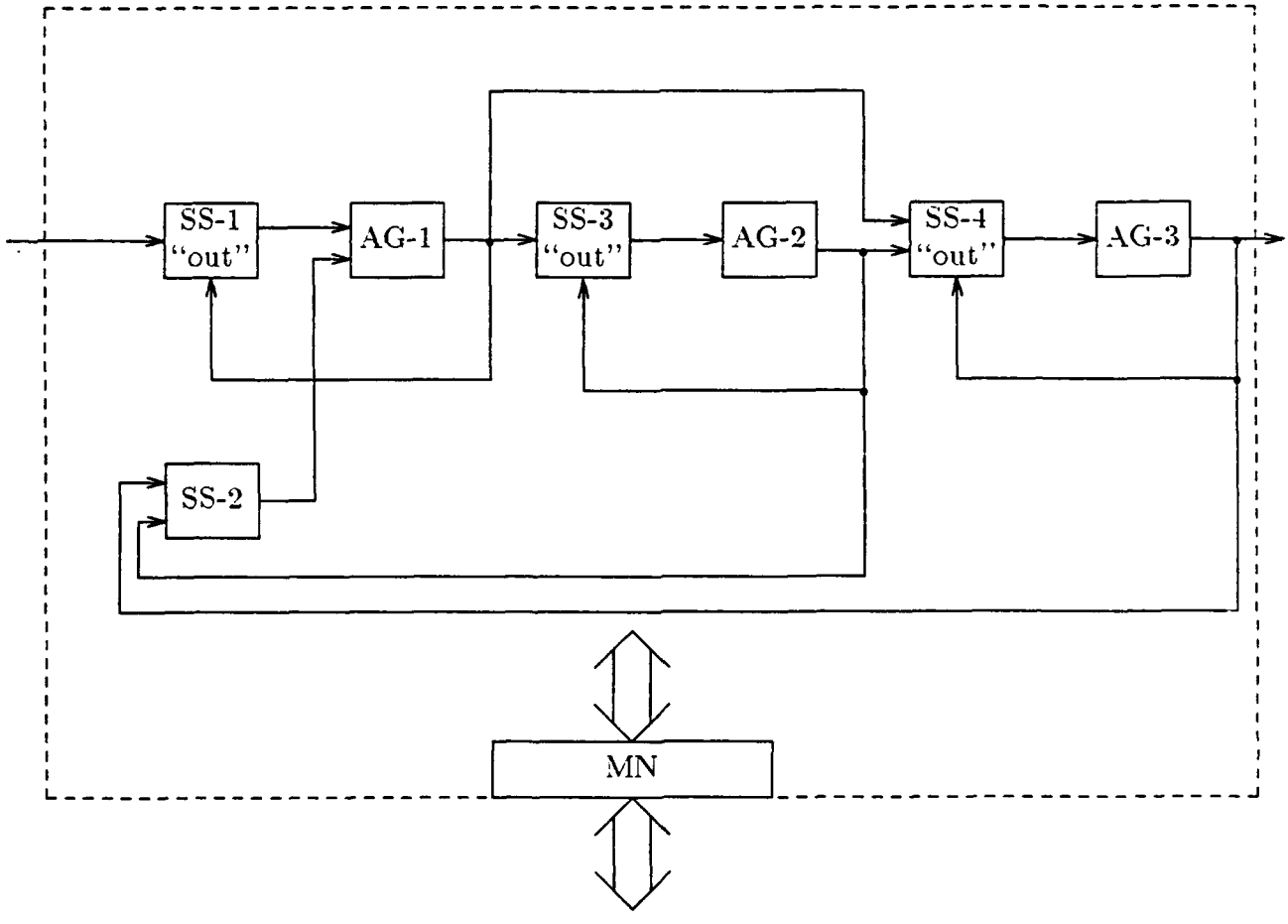


Figure 10. A Board Using the Scan-Set BIT Modules

It is assumed that the scan-set modules have internal "enable" generation logic shown in Figure 6, and the AG reset and preset signals in Figure 6 can be used to reset and preset each AG output set independently of another AG if necessary. The connections indicated in Tables 1 and 4 are assumed to have been made between each of the scan-set modules and the MN I/O at the MN-board interface. The AGs are assumed to be synchronous in general, even though portions of asynchronous logic can also be handled. The system clock is wired such that when the "auxiliary clock enable" line is activated, it is disabled, and the auxiliary clock controls the board operation. It is further assumed that whenever the auxiliary system clock is held at low level and the AG reset/preset signals are not active, the AGs retain their previous state indefinitely. A similar assumption holds for the BIT modules on the board as well. Finally, a BIT module-AG assignment shown in Table 7 is assumed for the purpose of controlling the test.

Table 7. A BIT Module-AG Assignment for the Network in Figure 10

BIT Module(s)	**Assigned AG**
SS-1 and SS-2	AG-1
SS-3	AG-2
SS-4	AG-3

A Procedure for Administering a Go/No-go Test of the Board in Example 2

The following steps are generally performed by the system level TCU. Steps 1 and 2 are preliminary steps that ensure proper clock control and MN and board reset before the actual tests can be performed. Steps 3 to 6 accomplish the scan-set BIT

module tests, while steps 7 to 10 facilitate scanning-in and the setting up of a desired test pattern and applying it to the board. Steps 11 to 13 accomplish scan-out of the board response and comparison with the expected response. Finally, step 14 completes the test by repeating the needed earlier steps with a sufficient number of test vectors.

1. Activate the "auxiliary clock enable" line, and hold the auxiliary system clock and BIT clock lines at low level by properly gating the clock signals generated within the TCU.

2. Shift in a 24-bit id and command word containing the appropriate 5-bit subsystem id, 6-bit board id, 3-bit command code indicating the specific MN and board reset, and 2-bit AG reset/preset code corresponding to no AG reset or preset. This word has to be shifted in through "id_in" input under the control of CDI clock. Arbitrary values may be used for the 5-bit AG id and the 3-bit BIT module id for this step. After 24 CDI clock cycles, apply the load address bits such that only the "id_load" signal is high for one CDI clock cycle. This step ensures that the necessary MN and board reset is accomplished. Raise the "dataout_load" signal to high level, which configures the observation register in the MN to a parallel-latch mode of operation for one CDI clock cycle. Then serially shift out the register contents ("dataout_load" = 0) and verify the MN and board reset bits that appear on the "dataout" line at the MN-TCU interface.

3. Repeat step 2 with another 24-bit word containing the 3-bit command code to deactivate the MN and board reset signals. Also, the 5-bit AG id corresponding to a given AG and the 3-bit BIT module id corresponding to a given scan-set

module are used in the 24-bit word. This step results in the "enable" of the chosen scan-set BIT module being high, which in turn sets it up to receive control signals from the MN.

4. Shift in a 16-bit control and data word through the "BIT control and data" line containing all zeros except "reset" = 1. After 16 CDI clock cycles, raise the "control_load" and "datain_load" signals to high level for one CDI clock cycle. This step ensures that the reset of all the scan-set BIT modules on the board is accomplished. Repeat this step with "reset" = 0 to deactivate the BIT module reset signal.

5. Apply the load address bits so that "datain_load" and "dataout_load" signals are at high level. Shift in data through "BIT control and data" input under the control of CDI clock. Apply the BIT clock in synchrony with the CDI clock. Under this mode, "scanout" output of the chosen scan-set module is connected to the "dataout" line at the MN-TCU interface with one CDI clock cycle latency. The output bit stream at "dataout" should follow the input bit stream with a known latency. This step checks the functionality of the scan-set module to a good extent. At the end of this step, hold the BIT clock at low level by proper gating of the clock and modify the load address bits so that all load signals are at low level.

6. Repeat steps 3 to 5 enough number of times with the 5-bit AG id and the 3-bit BIT module id, so chosen as to complete the testing of all the scan-set modules on the board.

7. Repeat step 4 so that all the scan-set modules are reset. This step ensures that they are in normal "pass through" mode of operation .

8. Repeat step 3 with the 5-bit AG id corresponding to AG-1 and the 3-bit BIT module id corresponding to SS-1.

9. Shift in a 16-bit control and data word containing all zeros except "c_2" = 1. After 16 CDI clock cycles, raise the "control_load" signal to high level for one CDI clock cycle. Then apply the BIT clock for one clock cycle so that the control signals are latched in the chosen scan-set module.

10. Raise the "datain_load" signal to high level by proper application of the load address bits. Shift in the desired test vector through "BIT control and data" input while applying CDI and BIT clocks in synchrony. After a predetermined number of CDI clock cycles, the test vector will be available at the "uutin" outputs of the SS-1 module for parallel application to AG-1 primary inputs. Then apply the auxiliary system clock for one clock cycle. Hold both the auxiliary and the BIT clocks at low level at the end of this step by properly gating the clock signals.

11. Repeat step 3 with the 5-bit AG id corresponding to AG-3 and the 3-bit BIT module id corresponding to SS-4.

12. Repeat step 9 with "c_1" = 1 ("c_2" = 0). This step enables parallel latching of the AG-3 primary outputs into SS-4.

13. Repeat step 9 with all zero 16-bit word. Raise "dataout_load" signal to high level, and apply the BIT clock in synchrony with the CDI clock. This step reconfigures

SS-4 into serial-shift mode of operation, and the "scanout" output of SS-4 is connected to the "dataout" line at the MN-TCU interface with one CDI clock cycle latency. Monitor the output data at the "dataout" line for a predetermined number of CDI clock cycles, and compare it with the expected result. Hold the BIT clock at low level at the end of this step by properly gating the clock.

14. Repeat steps 8 to 13 enough number of times with different test vectors, and determine the status of the board.

An incorrect output on the "dataout" line in step 14 indicates a faulty board, while correct outputs throughout step 14 generally imply a non-faulty board. However, the entire test procedure may have to be repeated several times and the consistency of the results must be verified to ensure test confidence. Furthermore, there may be a need for performing additional tests involving multiple boards to completely check the interconnections between the board under test and the other boards. However, administering these additional tests differs from the above test procedure only in the id values that need to be used in different steps for the board, the AG, and the scan-set BIT modules.

Example 3

Consider a board consisting of three ambiguity groups of chips and using cycle-by-cycle test point monitoring as the BIT technique. Test point monitoring is assumed to be performed by using the scan-set BIT modules, and pseudorandom patterns generated by a BILBO module are applied to AG-1 primary inputs as shown in Figure 11. For details on the individual BIT modules and how to use them in a design, the reader is referred to the relevant application note. Even though a go/no-go test is the subject of this example, Figure 11 also shows the test logic incorporated to perform fault-isolation to an AG for off-the-system board maintenance purposes.

It is assumed that the BILBO and the scan-set BIT modules have internal "enable" generation logic shown in Figure 6, and the AG reset and preset signals in Figure 6 can be used to reset and preset each AG output set independently of another AG if necessary. The connections indicated in Tables 1 and 4 are assumed to have been made between each of the BIT modules and the MN I/O at the MN-board interface. An MN output that was not assigned to any of the BIT modules on the board needs to be assigned to the "muxctrl" input of the multiplexer, and "CD[5]" output of the MN was arbitrarily chosen to be connected to the "muxctrl" input.

The AGs are assumed to be synchronous in general, even though portions of asynchronous logic can also be handled. The system clock is wired such that when the "auxiliary clock enable" line is activated, it is disabled, and the auxiliary clock controls the board operation. It is further assumed that whenever the auxiliary system clock is held at low level and the AG reset/preset signals are not active, the AGs retain their

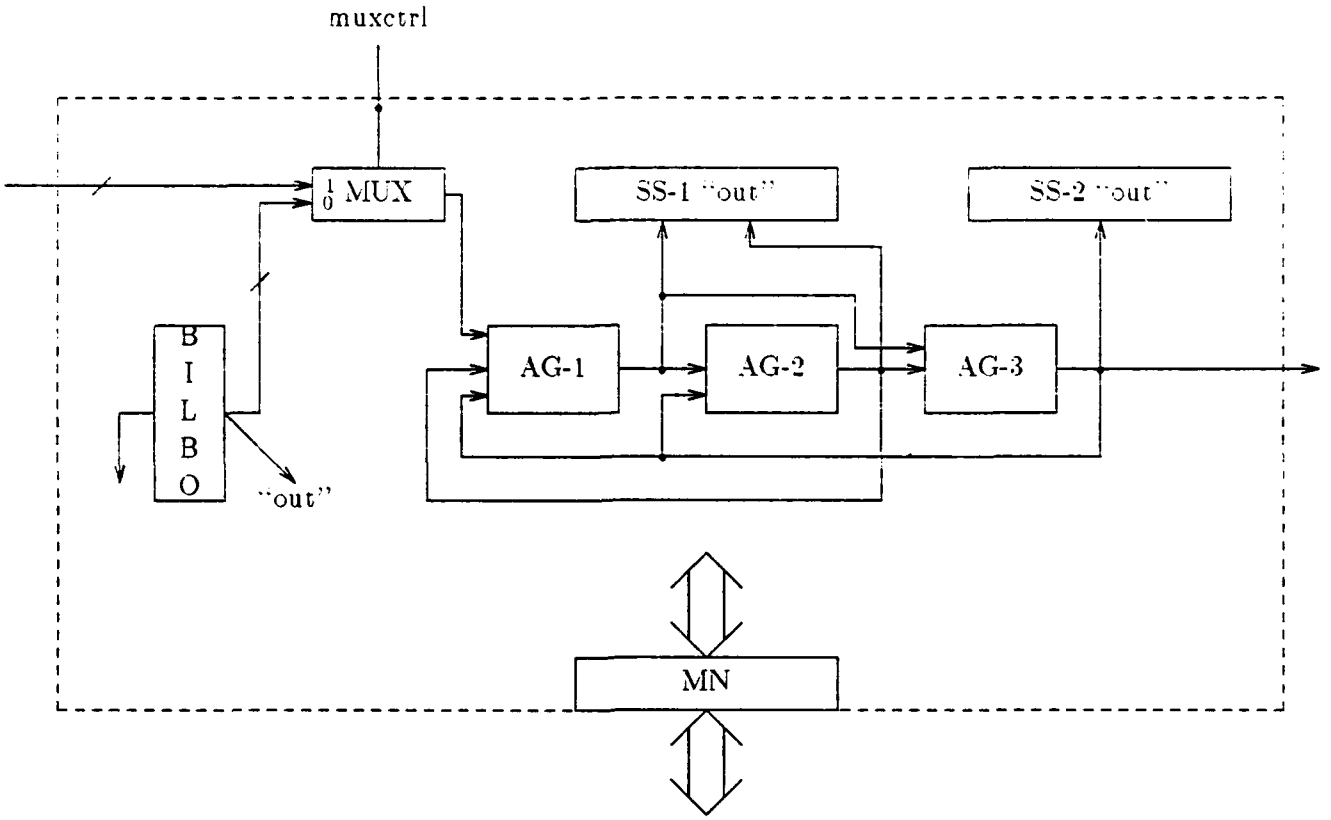


Figure 11. A Board Using the Cycle-by-Cycle Test Point Monitoring BIT Technique

previous state indefinitely. A similar assumption holds for the BIT modules on the board as well. Finally, a BIT module-AG assignment shown in Table 8 is assumed for the purpose of controlling the test.

Table 8. A BIT Module-AG Assignment for the Network in Figure 11

Bit Module	**Assigned AG**
BILBO	AG-1
SS-1	AG-2
SS-2	AG-3

A Procedure for Administering Go/No-go Test of the Board in Example 3

The following steps are generally performed by the system level TCU. Steps 3 to 6 accomplish the scan-set BIT module testing, whereas steps 7 to 12 test the functions of the BILBO module. Application of the test vector generated by the BILBO to the board is accomplished in steps 13 to 15, while step 16 facilitates the comparison of the actual board response to the expected response. Steps 17 through 19 reconfigure the BILBO to generate the next test vector. Finally, step 20 completes the test by repeating the needed earlier steps enough number of times so that a sufficient number of test vectors generated by the BILBO are applied to the board and the results are verified.

1. Activate the "auxiliary clock enable"line, and hold the auxiliary system clock and BIT clock lines at low level by properly gating the clock signals generated within the TCU.

2. Shift in a 24-bit id and command word containing the appropriate 5-bit subsystem id, 6-bit board id, 3-bit command code indicating the specific MN and board reset, and 2-bit AG reset/preset code corresponding to no AG reset or preset. This word has to be shifted in through "id_in" input under the control of CDI clock. Arbitrary values may be used for the 5-bit AG id and the 3-bit BIT module id for this step. After 24 CDI clock cycles, apply the load address bits such that only the "id_load" signal is high for one CDI clock cycle. This step ensures that the necessary MN and board reset is accomplished. Raise the "dataout_load" signal to high level, which configures the observation register in the MN to a parallel-latch mode of operation for one CDI clock cycle. Then serially shift out

the register contents ("dataout_load" = 0) and verify the MN and board reset bits that appear on the "dataout" line at the MN-TCU interface.

3. Repeat step 2 with another 24-bit word containing the 3-bit command code to deactivate the MN and board reset signals. Also, the 5-bit AG id corresponding to AG-2 and the 3-bit BIT module id corresponding to SS-1 are used in the 24-bit word. This step results in the "enable" of SS-1 being high, which in turn sets it up to receive control signals from the MN.

4. Shift in a 16-bit control and data word through "BIT control and data" input containing all zeros except "reset" = 1 and "reset1" = 1. After 16 CDI clock cycles, raise the "control_load" and "datain_load" signals to high level for one CDI clock cycle. This step ensures that the reset of all the BIT modules on the board is accomplished. Repeat this step with "reset" = 0 and "reset1" = 0 to deactivate the BIT module reset signals.

5. Apply the load address bits so that the "datain_load" and "dataout_load" signals are at high level. Shift in data through "BIT control and data" input under the control of CDI clock. Apply the BIT clock in synchrony with the CDI clock. The output bit stream at "dataout" line should follow the input bit stream with a known latency. This step checks the functionality of the chosen scan-set module to a good extent. At the end of this step, hold the BIT module clock at low level by properly gating the clock and modify the load address bits such that all load signals are at low level.

6. Repeat steps 3 to 5 enough number of times with the 5-bit AG id and the 3-bit BIT module id, so chosen as to complete the testing of all the scan-set modules on the board.

7. Repeat step 4 so that all the BIT modules are again reset.

8. Repeat step 3 with the 5-bit AG id corresponding to AG-1 and the BIT module id corresponding to the BILBO.

9. Shift in a 16-bit control and data word containing all zeros except "shiften" = 1 and "c_2" = 1. After 16 CDI clock cycles, raise the "control_load" and "datain_load" signals to high level for one CDI clock cycle. Then apply the BIT clock for one clock cycle. This step enables serial shifting of data in the BILBO.

10. Repeat step 5. Under this mode, "tout" output of the BILBO is connected to the "dataout" line at the MN-TCU interface with one CDI clock cycle latency. Similarly, the "BIT control and data" line is connected to the "sin" input of the BILBO with one CDI clock cycle latency. This step checks the serial shifting function of the BILBO.

11. Repeat step 9 with "shiften" = 1, "c_1" = 1, and "c_2" = 1. This step enables the test pattern generator mode of the BILBO.

12. Raise the "dataout_load" signal to high level. Apply the BIT clock in synchrony with the CDI clock. Under this mode, the output on the "dataout" line should match the precomputed values obtained from simulation of the BILBO. This step generally assumes that the BILBO has a non-zero output vector at its "out" port at the end of step 11. Hold the BIT clock at low level at the end of this step by

properly gating the clock.

13. Shift in a 16-bit control and data word containing all zeros except "reset1" = 1. After 16 CDI clock cycles, raise the "control_load" and "datain_load" signals to high level for one CDI clock cycle. This step disables shifting in the BILBO during the time when AG-3 primary outputs will be scanned out through the SS-2 module. Repeat this step with "c_1" = 1 ("reset1" = 0).

14. Repeat step 3 with the 5-bit AG id corresponding to AG-3 and the 3-bit BIT module id corresponding to SS-2.

15. Apply the auxiliary system clock for one clock cycle. Then, apply the BIT clock for one clock cycle. During this step, the test vector generated by the BILBO gets applied to AG-1 primary inputs and the board state changes. Furthermore, the SS-2 module is configured to accept AG-3 primary outputs in parallel.

16. Raise the "dataout_load" signal to high level. Repeat step 9 with all zero 16-bit control and data word, and apply the BIT module clock in synchrony with the CDI clock. This step reconfigures SS-2 to serial-shift mode of operation after parallel-latching of AG-3 primary outputs. Monitor the output data at the "dataout" line for a predetermined number of CDI clock cycles and compare it with the expected result. Hold the BIT clock at low level at the end of this step by properly gating the clock signal.

17. Repeat step 3 with the 5-bit AG id corresponding to AG-1 and the 3-bit BIT module id corresponding to the BILBO.

18. Repeat step 9 with "shiften" = 1, "c_1" = 1, and "c_2" = 1.

19. Apply the BIT clock for one clock cycle. At the end of this step, a different test vector will be available at the "out" port of the BILBO.

20. Repeat steps 13 to 19 a predetermined number of times and determine the status of the board.

An incorrect output at the "dataout" line in step 20 indicates a faulty board, while correct outputs throughout step 20 generally imply a non-faulty board. However, the entire test procedure may have to be repeated several times and the consistency of the results must be verified to ensure test confidence. Furthermore, there may be a need for performing additional tests involving multiple boards to completely check the interconnections between the board under test and the other boards. However, administering these additional tests differs from the above test procedure only in the id values that need to be used in different steps for the board, the AG, the BIT modules, and in setting the value of "muxctrl" input.

Example 4

Consider a board consisting of three ambiguity groups of chips and using test point monitoring with output data compression as the BIT technique. Test point monitoring is assumed to be performed by using a BILBO module used as a parallel signature analyzer, while test vectors at the AG-1 primary inputs are provided by an internal exclusive-or LFSR test pattern generator as shown in Figure 12. For details on the individual BIT modules and how to use them in a design, the reader is referred to the relevant application note. Even though a go/no-go test is the subject of this example, Figure 12 also shows the test logic incorporated to perform fault-isolation to an AG for off-the-system board maintenance purposes.

It is assumed that the BILBO and the tpg modules have internal "enable" generation logic shown in Figure 6, and the AG reset and preset signals in Figure 6 can be used to reset and preset each AG output set independently of another AG if necessary. The connections indicated in Tables 1 and 4 are assumed to have been made between each of the BIT modules and the MN I/O at the MN-board interface. Four MN outputs that are not assigned to the BILBO and the tpg modules need to be assigned to the "muxctrl" inputs, and an arbitrary choice shown in Table 9 was made.

Table 9. An MN Output/Multiplexer Control Signal Correspondence for the Network in Figure 12

MN Output	Signal
CD[5]	muxctrl1
CD[6]	muxctrl2
CD[7]	muxctrl3
CD[14]	muxctrl4

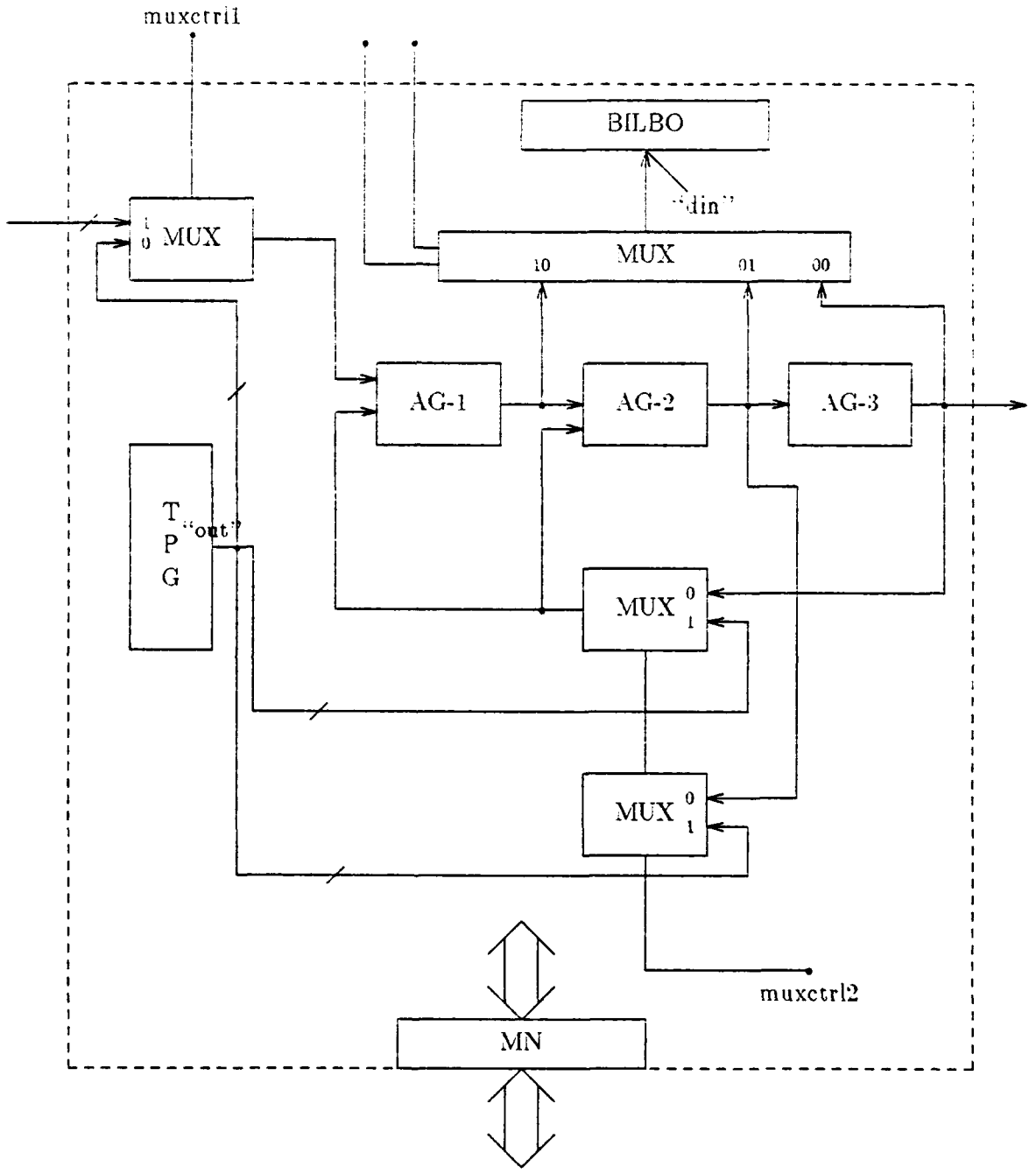


Figure 12. A Board Using Test Point Monitoring BIT Technique with Output Response Compression

The AGs are assumed to be synchronous in general, even though portions of asynchronous logic can also be handled. The system clock is wired such that when the "auxiliary clock enable" line is activated, it is disabled, and the auxiliary clock controls the board operation. It is further assumed that whenever the auxiliary system clock is held at low level and the AG reset/preset signals are not active, the AGs retain their previous state indefinitely. A similar assumption holds for the BIT modules on the board as well. Finally, a BIT module-AG assignment shown in Table 10 is assumed for the purpose of controlling the test.

Table 10. A BIT Module-AG Assignment for the Network in Figure 12

BIT Module	Assigned AG
tpg	AG-1
BILBO	AG-3

A Procedure for Administering Go/No-go Test of the Board in Example 4

The following steps are generally performed by the system level TCU. Steps 3 to 8 accomplish the testing of the BILBO module, while steps 9 to 14 test the functions of the tpg BIT module. Steps 15 to 17 set up the board for a go/no-go test. Test vectors generated by the tpg module are applied to the board and the response is compressed by the BILBO used as a signature analyzer in step 18. Finally, steps 19 and 20 facilitate comparison of the final signature analyzer stored in the BILBO with the expected one and determination of the board status.

1. Activate the "auxiliary clock enable" line, and hold the auxiliary system clock and the BIT clock lines at low level by properly gating the clock signals generated within the TCU.

2. Shift in a 24-bit id and command word containing the appropriate 5-bit subsystem id, 6-bit board id, 3-bit command code indicating the specific MN and board reset, and 2-bit AG reset/preset code corresponding to no AG reset or preset. This word has to be shifted in through "id_in" input under the control of CDI clock. Arbitrary values may be used for the 5-bit AG id and the 3-bit BIT module id for this step. After 24 CDI clock cycles, apply the load address bits such that only the "id_load" signal is high for one CDI clock cycle. This step ensures that the necessary MN and board reset is accomplished. Raise the "dataout_load" signal to high level, which configures the observation register in the MN to a parallel-latch mode of operation for one CDI clock cycle. Then serially shift out the register contents ("dataout_load" = 0) and verify the MN and board reset bits that appear on the "dataout" line at the MN-TCU interface.

3. Repeat step 2 with another 24-bit word containing the 3-bit command code to deactivate the MN and board reset signals. Also, the 5-bit AG id corresponding to AG-3 and the 3-bit BIT module id corresponding to the BILBO are used. This step results in the "enable" of the BILBO being high, which in turn sets it up to receive control signals from the MN.

4. Shift in a 16-bit control and data word through "BIT control and data" input containing all zeros except "reset" = 1 and "reset1" = 1. After 16 CDI clock

cycles, raise the "control_load" and "datain_load" signals to high level for one CDI clock cycle. This step ensures that the reset of all the BIT modules on the board is accomplished.

5. Repeat step 4 with "shiften" = 1 and "c_2" = 1 (all the remaining bits being zero). Then apply the BIT clock for one clock cycle. This step enables serial shifting of data in the BILBO module.

6. Apply the load address bits so that "datain_load" and "dataout_load" are at high level. Shift in data through "BIT control and data" input under the control of CDI clock. Apply the BIT clock in synchrony with the CDI clock. The output bit stream at the "dataout" line should follow the input bit stream with a known latency. Hold the BIT clock at low level at the end of this step by properly gating the clock.

7. Repeat step 4 with "shiften" = 1, "c_1" = 1, "c_2" = 1, and "c_3" = 1 (all the remaining bits being zero). Then apply the BIT clock for one clock cycle. This step enables the parallel signature analyzer mode of the BILBO.

8. Raise the "dataout_load" signal to high level. Apply the BIT clock in synchrony with the CDI clock. Under this mode, the output on the "dataout" line should match the precomputed values obtained from simulation of the BILBO. This step generally assumes that the BILBO has a non-zero output vector at its "out" port at the end of step 6. Hold the BIT clock at low level at the end of this step by properly gating the clock signal.

9. Repeat step 4. This step resets all the BIT modules on the board again.

10. Repeat step 3 with the 5-bit AG id corresponding to AG-1 and the 3-bit BIT module id corresponding to the tpg module.

11. Repeat step 4 with "shiften" = 1 (all the remaining bits being zero). Then apply the BIT clock for one clock cycle. This step enables serial shifting of data in the tpg module.

12. Repeat step 6. This step checks the serial shifting function of the tpg module.

13. Repeat step 4 with "shiften" = 1, "fbin" = 1, and "fbctrl" = 1 (all the remaining bits being zero). Then apply the BIT clock for one clock cycle.

14. Repeat step 8. During this step, the output on the "dataout" line should match the precomputed values obtained from simulation of the test pattern generator module.

15. Shift in 16-bit control and data words containing the appropriate control and data bits required for initialization of the tpg. The initialization process consists of repeated shifting in of a 16-bit control and data word, raising the "control_load" and/or "datain_load" signals to high level for one CDI clock cycle after every 16 CDI clock cycles, and applying the BIT clock for one clock cycle. During this initialization process, the 16-bit word should have zeros in the bits corresponding to the multiplexer control signals.

16. Repeat step 3 with the 5-bit AG id corresponding to AG-3, and the 3-bit BIT module id corresponding to the BILBO.

17. Repeat step 7.

18. Apply the auxiliary system clock and the BIT clocks in synchrony for a given number of clock cycles. During this step, test vectors generated by the tpg are applied to AG-1 primary inputs, and AG-3 primary outputs are compressed by the BILBO module. Hold the auxiliary system clock and the BIT clock at low level at the end of this step by properly gating the clock signals.

19. Repeat step 4 with "reset1" = 1 ("reset" = 0). This step disables shifting in the BILBO and thus locks the resulting signature in the BILBO.

20. Raise the "dataout_load" signal to high level. Repeat step 5. Apply the BIT clock in synchrony with the CDI clock. Monitor the signature stored in the BILBO and compare it with the expected one.

An incorrect signature at the end of step 20 indicates a faulty board, while a correct signature generally implies a non-faulty board. However, the entire procedure may have to be repeated several times and the consistency of the test results must be verified to ensure test confidence. Furthermore, there may be a need for performing additional tests involving multiple boards to completely check the interconnections between the board under test and the other boards. However, administering these additional tests differs from the above test procedure only in the id values that need to be used in different steps for the board, the AG, the BIT modules, and in setting the value of "muxctrl1" input.

Summary

This application note has provided a functional and structural description of a maintenance node (MN) unit and illustrated its use in controlling the in-system board test by considering four examples. The examples chosen encompass all the board-level BIT techniques supported by the TEA system.

SPECIFIC IMPLEMENTATION OF THE MAINTENANCE NODE

Objective

This note provides a mixed-level (i.e.. functional block/gate/transistor level) description of the maintenance node unit (MN) that has been simulated and verified for its intended functionality using the CADAT simulator.

The "auxiliary clock processor" in the MN has to be tuned to the specific application under consideration so that it generates the required clock phases. satisfying the desired phase and frequency relationships. It is assumed in this application note that the auxiliary clock processor is a non-overlapping two-phase clock generator.

Block Diagram

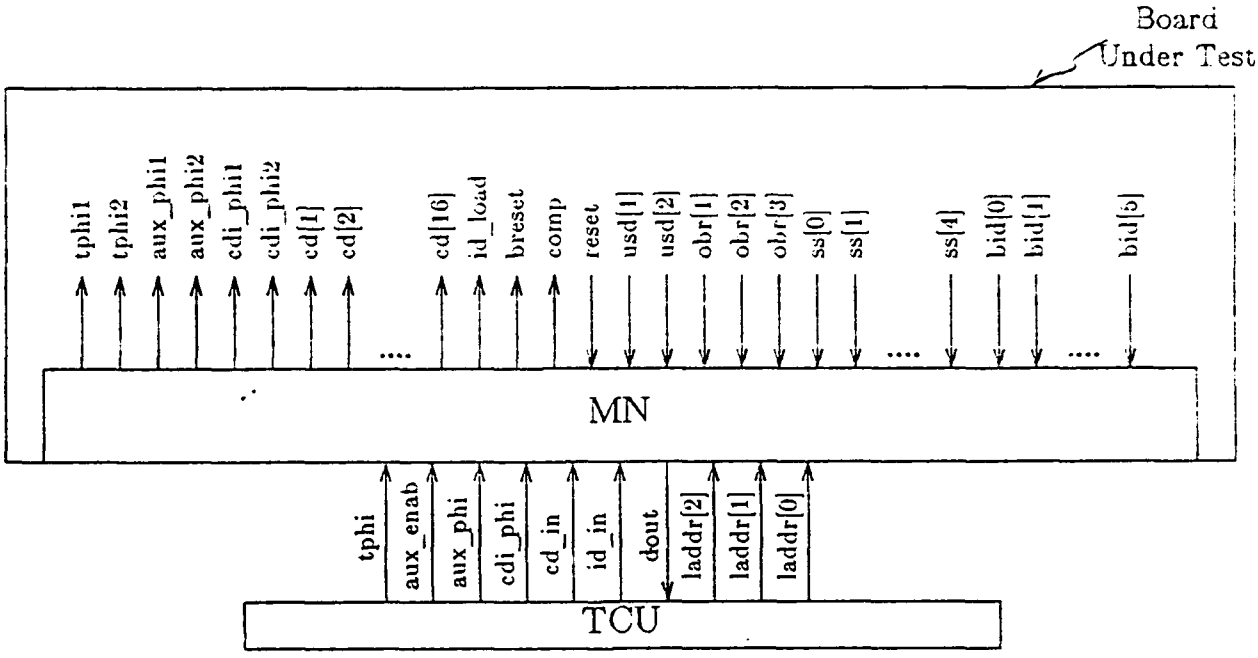


Figure 1. Block Diagram of the MN Showing Its I/O Pins

Pin Description (See also the logic diagrams and the application note on the utilization of the MN)

Name	Description
aux_enable	Auxiliary system clock enable
aux_phi	Auxiliary system clock
aux_phi1, aux_phi2	Non-overlapping phases of the auxiliary system clock
bid[5:0]	6-bit board identification code (id) (generally hardwired)
breset	Board-reset output from the MN
cdi_phi	Control, data, and id (CDI) clock
cdi_phi1, cdi_phi2	Non-overlapping phases of the CDI clock
cd_in	Serial control and data input
cd[16:1]	Parallel control and data outputs
comp	"Compare successful" signal pin
dout	Serial data output from the MN
id_in	Serial identification address input
id_load	The "identification-load" signal pin
laddr[2:0]	"Load address decoder" inputs
obr[3:1], usd[2:1]	"Observation register" inputs
reset	"Power-on reset" input for the MN
ss[4:0]	5-bit subsystem id (generally hardwired)
tphi	BIT module clock
tphi1, tphi2	Non-overlapping phases of the BIT module clock

Logic Description

The MN has the following functional groups:

1) Control and data register
2) Identification and command register, and the associated logic
3) Observation register
4) Load address decoder
5) Auxiliary system clock processor
6) Control, data, and id (CDI) clock processor
7) BIT module clock processor

Mixed-level (i.e., functional block/gate/transistor level) descriptions of these constituent groups are provided in the following paragraphs.

1) Control and data register

Figure 2 shows the control and data register portion of the MN. Its CADAT description uses the primitives "dff-l," "nlmos," "buf," and gates such as "and" and "inv."

2) Identification and command register, and the associated logic

Figure 3 illustrates the functions of the id and command register portion of the MN. Its CADAT description uses the primitives "dff-l," "nlmos," "buf," "comparator," "decbin" (decoder), and other basic gates.

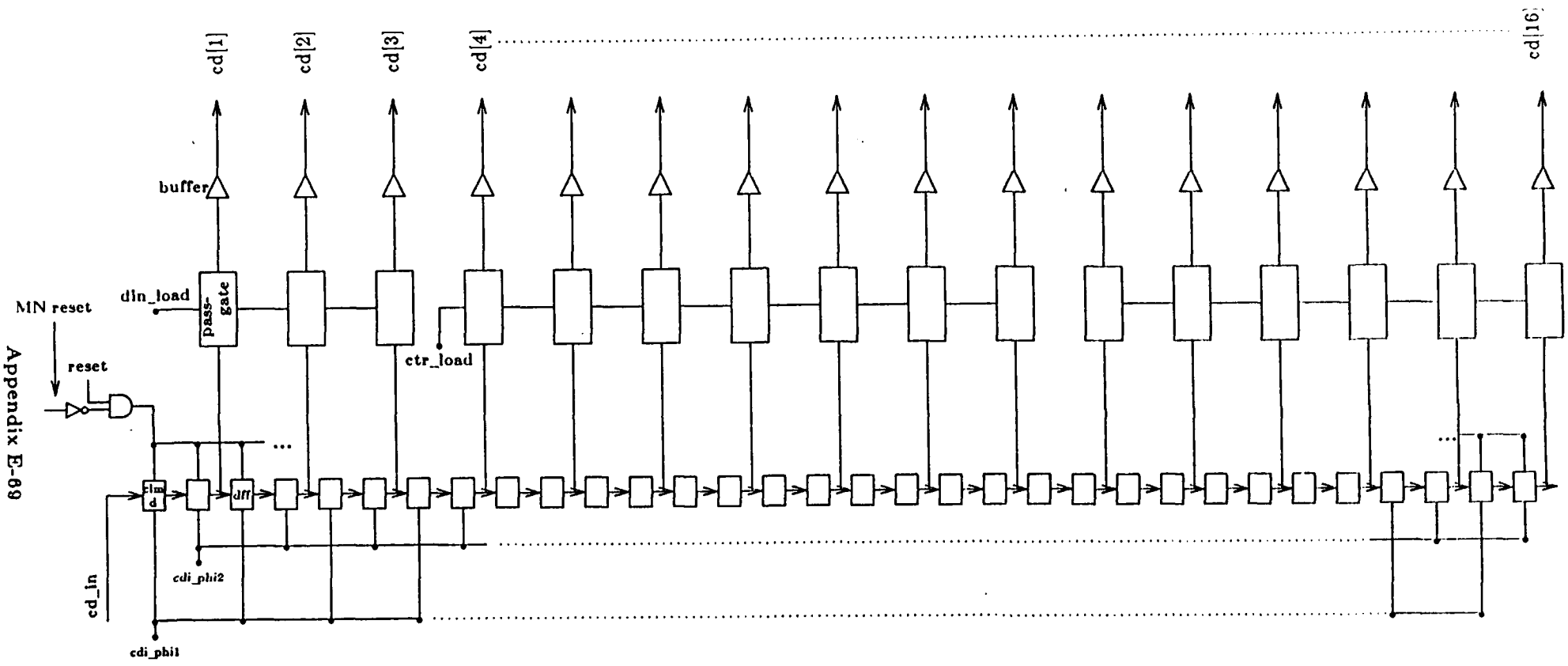


Figure 2. Control and Data Register Portion of the MN

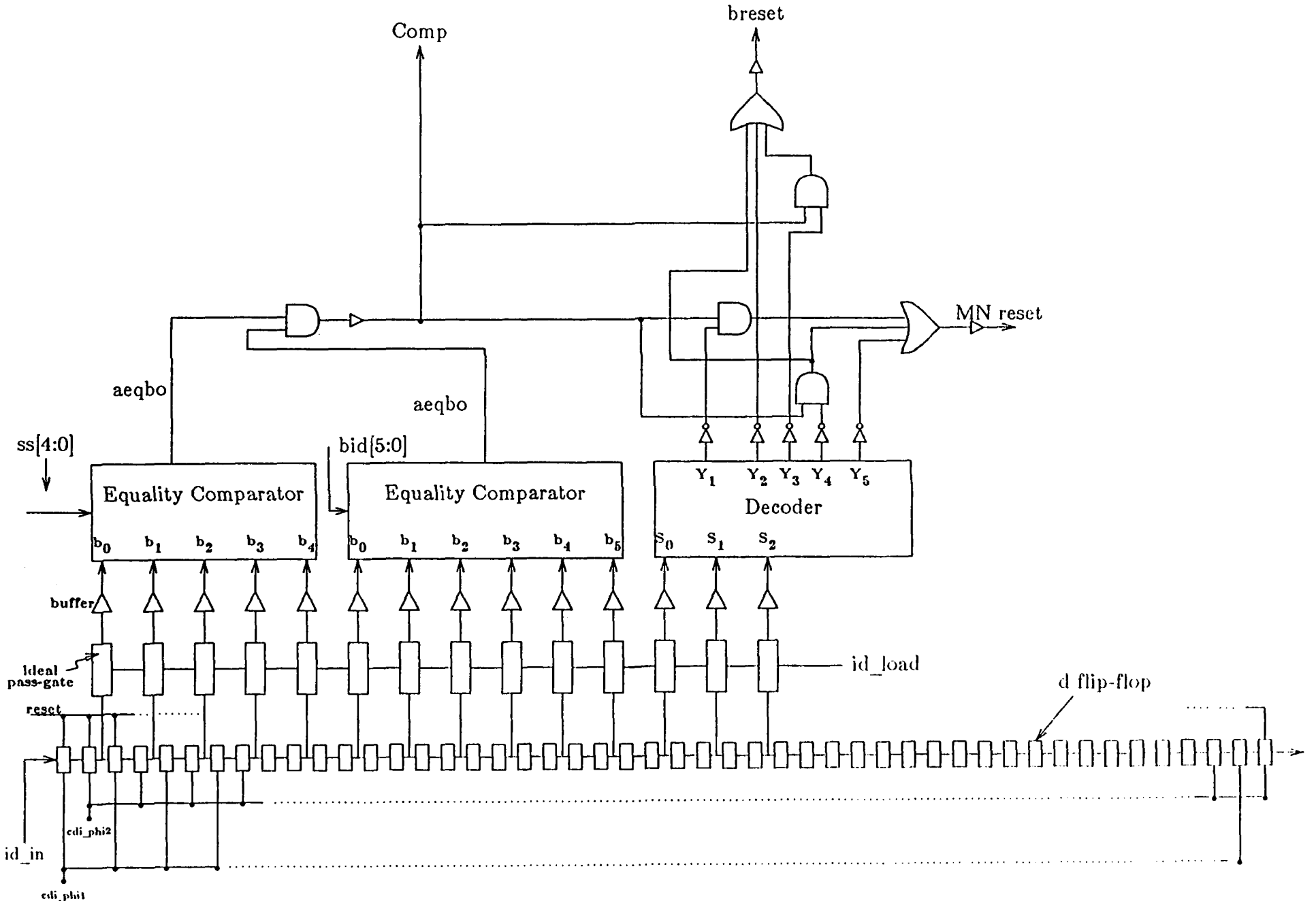


Figure 3. Identification and Command Register Portion of the MN

3) **Observation register**

The observation register portion of the MN is shown in figure 4. Its CADAT description uses the primitives "dff-l," "mux," and "buf."

4) **Load address decoder**

Figure 5 shows an implementation of the load address decoder. Its CADAT description uses the basic gates and the primitive "buf."

5) **Auxiliary system clock processor**

A non-overlapping two-phase clock generator shown in figure 6 is assumed to be the auxiliary system clock processor in the chosen specific implementation. Its CADAT description uses a "3s-h" (tristate buffer) primitive in addition to the basic gates.

6 & 7) **CDI and BIT module clock processors**

These are the same as in figure 6, except no output tristate buffers are needed. However, their CADAT description utilizes the tristate buffers with their control input tied to power supply (Vdd) for reasons of uniformity.

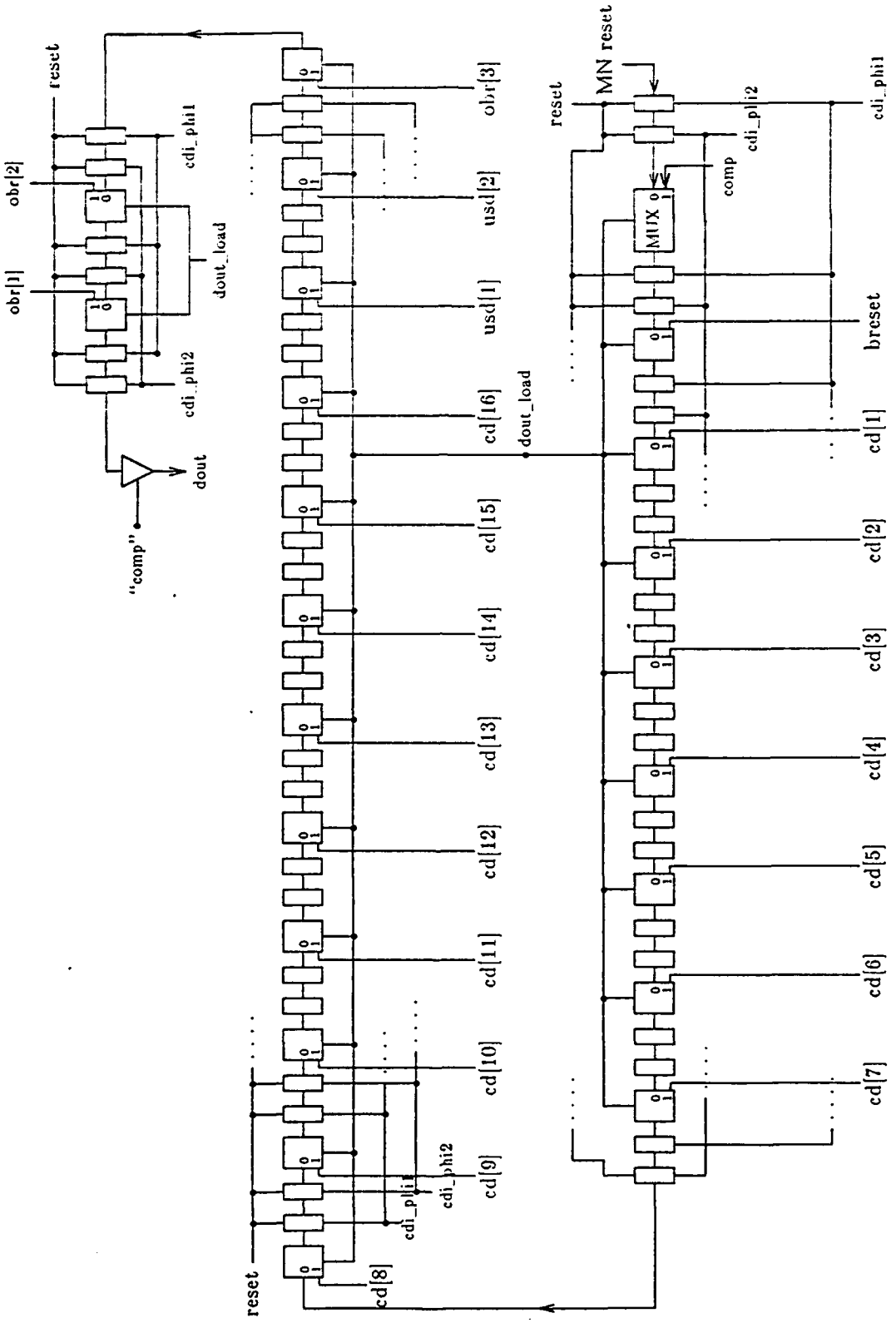


Figure 4. Observation Register Portion of the MN

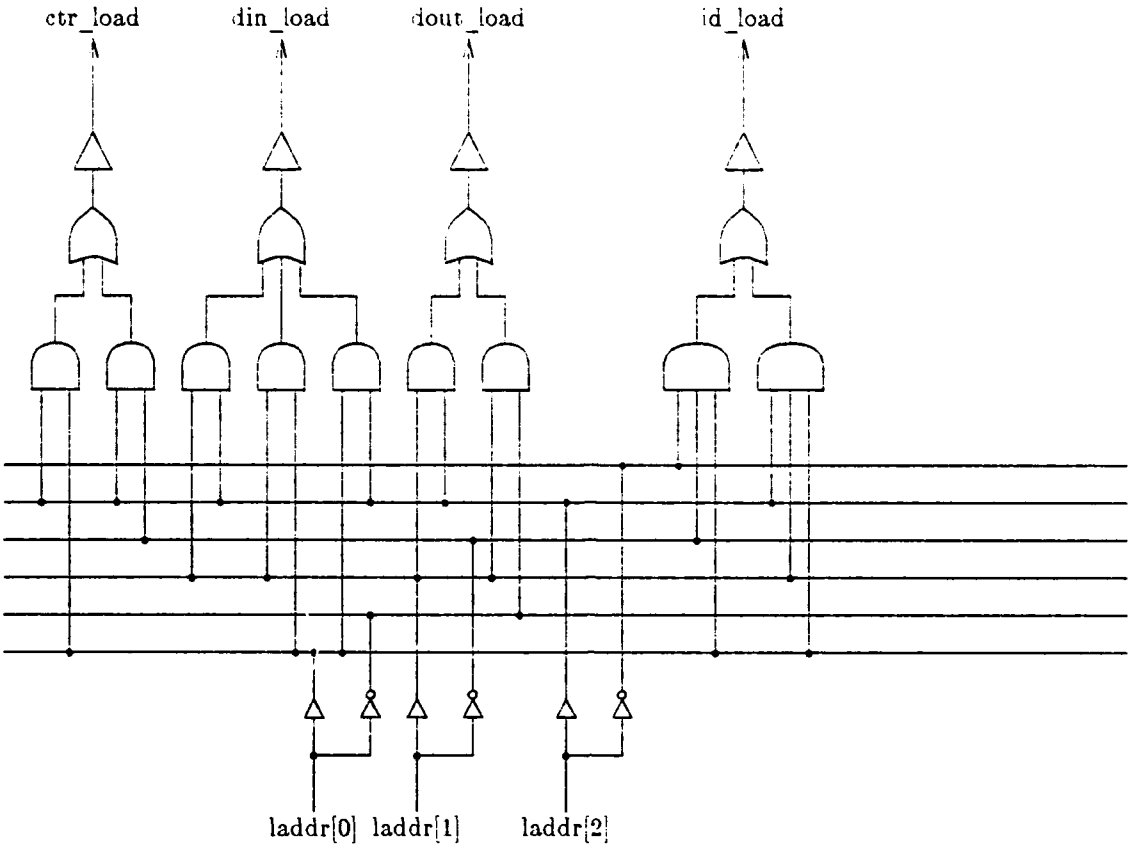


Figure 5. Load Address Decoder

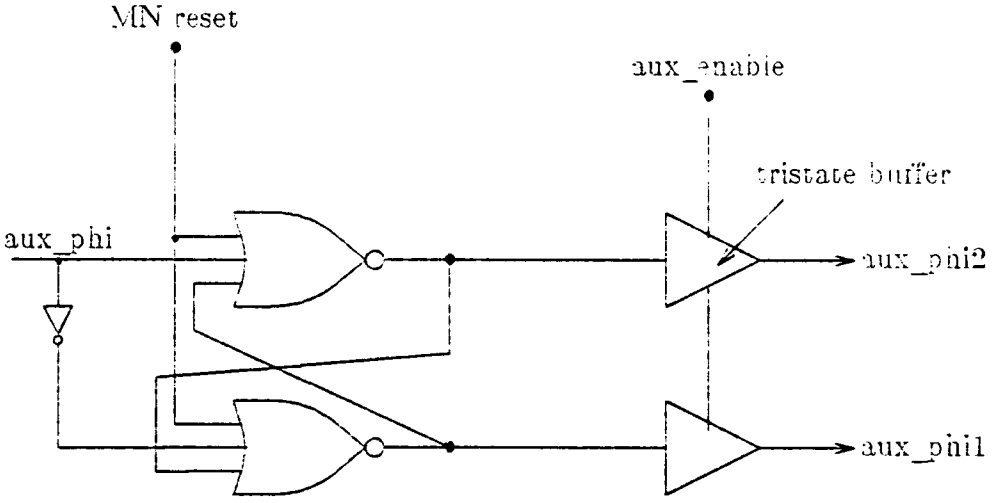


Figure 6. Two-phase Clock Generator Used as Auxiliary System Clock Processor

PROGRAMMABLE FEEDBACK PSEUDORANDOM TEST PATTERN GENERATOR (EXTERNAL EXCLUSIVE-OR IMPLEMENTATION) APPLICATION NOTE

Objective

The objective of this application note is to describe the functionality of the programmable feedback pseudorandom test pattern generator module using an external exclusive-or feedback structure.

Block Diagram

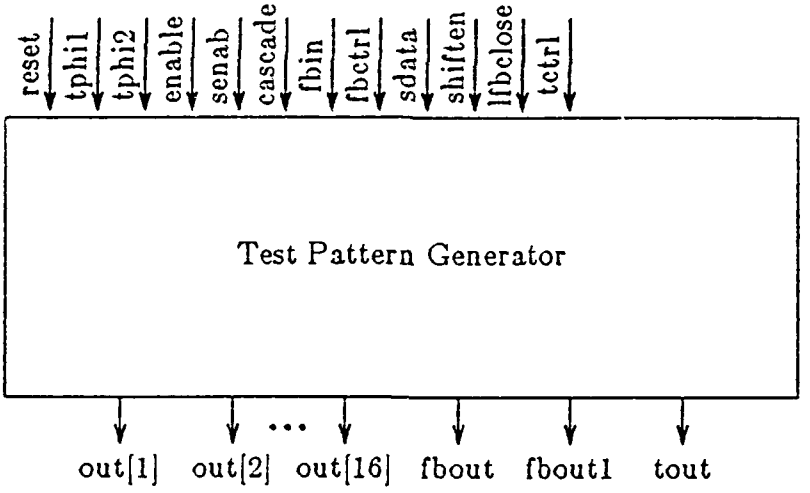


Figure 1. Block Diagram of a Pseudorandom Test Pattern Generator Module.

Pin Description (see also the logic diagram and module functions)

Name	Description
reset	Master reset of all the flip-flops
tphi1/tphi2	Two-phase clock
enable1	Shift enable1 for the control inputs
tctrl	Tristate control for "tout" output
senab	Enable control for the serial data (used in cascading multiple test pattern generator modules)
cascade	Data input used in cascading multiple test pattern generator modules
lfin	Input of the feedback network (used in cascading multiple test pattern generator modules)
fbin	Feedback coefficient input
fbctrl	Control input to enable1 feedback connection
sdata	Serial data input
shiften	Shift enable1 for the data in the test pattern generator
lfbclose	Input pin used in cascading multiple test pattern generator modules
out [16:1]	Data outputs of the test pattern generator
fbout	Feedback coefficient output
fbout1	Feedback coefficient output for test control purposes
lfout	Output of the feedback network (used in cascading multiple test pattern generator modules)
lfout1	Output of the feedback network for test control purposes
tout	Serial test data output

Logic Diagram

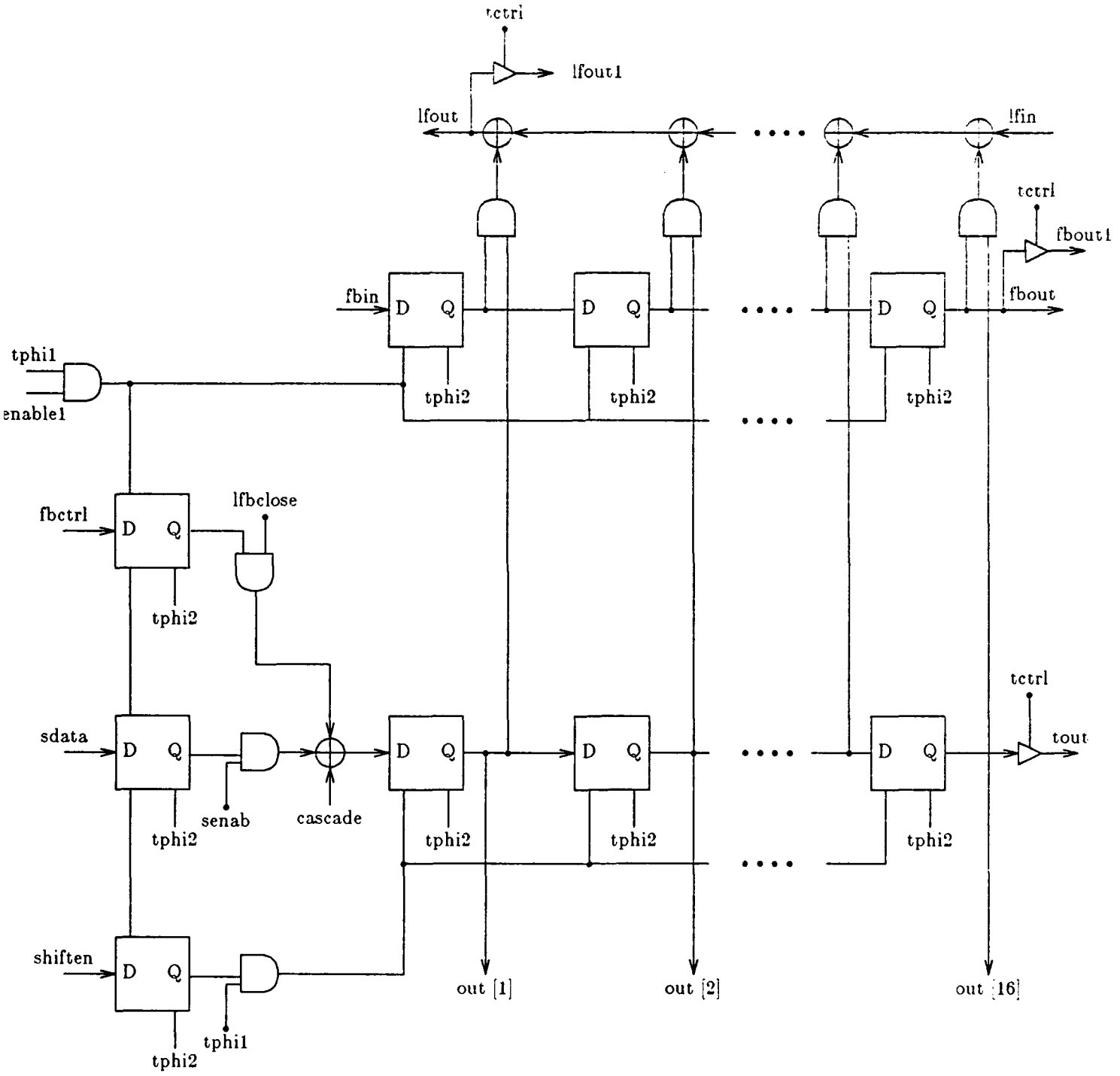


Figure 2. Logic Diagram of the Test Pattern Generator Module

Module Function:

The primary function of this module is to generate pseudorandom test vectors for application to a unit under test (UUT). All the required control circuitry is incorporated within the module itself so that the master test control unit (TCU) can properly control the operation of multiple test pattern generators in a design with minimal need for additional "glue" logic.

Required Pin Connections

Certain pin connections are necessary to use the test pattern generator module. If the module is to be used by itself (i.e., as a 16-bit test vector generator), the connections shown in Figure 3 are required.

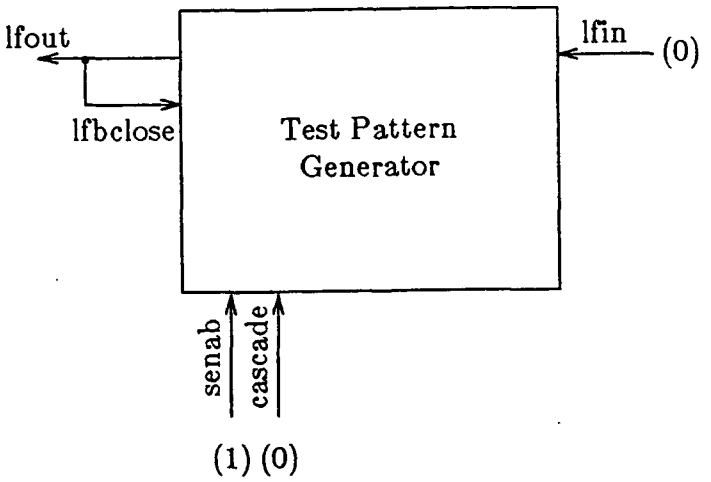


Figure 3. Required Pin Connections to Use a Single Test Pattern Generator Module

If multiple test pattern generator modules are to be used as **a single programmable feedback test pattern generator**, the connections shown in Figure 4 (illustrated for three modules) are required. The terms "enable1," "fbctrl," "sdata," and "shiften" are common to all the test pattern generator modules in this mode of operation.

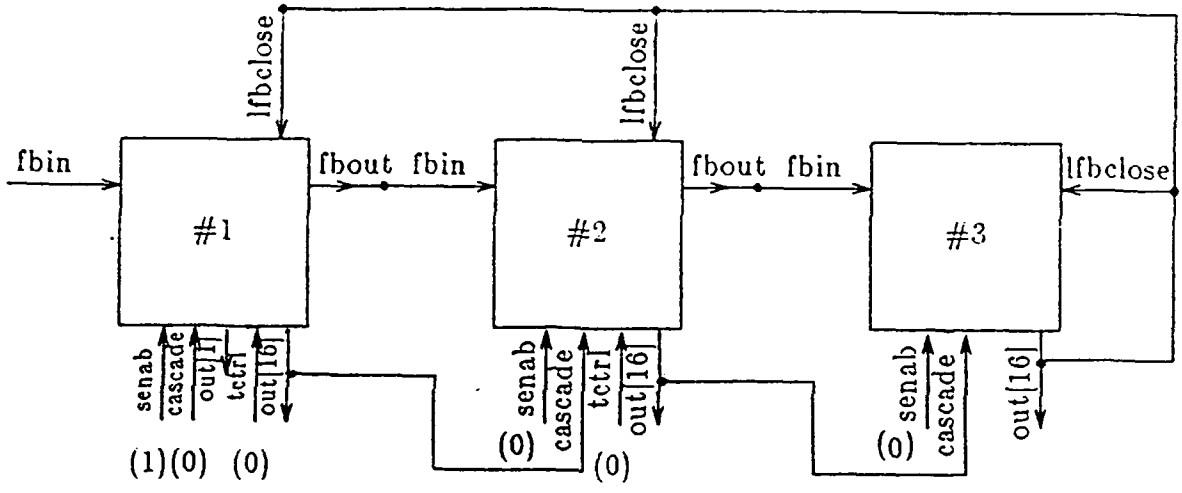


Figure 4. Required Pin Connections to Use Multiple Test Pattern Modules as an Equivalent Single Module

Propagation delay associated with the exclusive-or gate chain in the feedback path is of genuine concern when using multiple test pattern generator modules (as shown in Figure 4) as it has an adverse impact on the maximum test clock frequency. In certain applications, an internal exclusive-or implementation of the test pattern generator, which is described in a different application note, might be advantageous to use since the feedback path in an internal exclusive-or implementation does not involve gate delays. It should be noted, however, that the maximum allowed test clock fre-

quency always reduces (albeit by different extents) as a result of cascading the test pattern generator modules, irrespective of the implementation used. Hence, in terms of speed, the best possible hardware implementation of the test pattern generator module is a fixed word-length module with no provisions for cascading. However, such a design reduces the flexibility in utilizing the module and must be carefully considered.

Operation of the Module

All the test pattern generator modules in the design are reset by a master "reset" signal common to all the modules. Then, each module is selected by means of its enable1 signal. Predetermined test patterns are shifted through the feedback coefficient and data shift registers through "fbin" and "sdata" inputs respectively. The outputs at "fbout," "lfout," and "tout" pins are verified by the test control unit (TCU) against the unexpected values. These tests are performed for "shiften" = 1 and "fbctrl" = 0; "shiften" = 1 and "fbctrl" = 1; and "shiften" =0 control signal values. If any of these tests fail, the test pattern generator module could be faulty and might have to be replaced. Upon completion of these tests, the coefficients (binary 1 and 0) corresponding to the desired feedback polynomial are shifted in through "fbin" input and, simultaneously, test pattern generator data outputs are initialized to a given not-all-zero state by shifting data through "sdata" input. During this initialization period, "shiften" = 1 and "fbctrl" = 0 logic values are maintained. After a predetermined number of clock cycles, "fbctrl" is raised high and then "enable1" is brought low in the succeeding clock cycle so that the feedback coefficients and the

control signals are latched. From this point on, pseudorandom test vectors are available at "out [16:1]" outputs every clock cycle. The repetition interval of the generated test patterns is dependent on the feedback polynomial chosen, and a maximal repetition interval of $2^{16} - 1$ is obtained by choosing a primitive feedback polynomial [1]. A typical timing sequence to shift in a feedback polynomial $\overset{\text{(lsb)}}{0}0000010000010001\overset{\text{(msb)}}{}$'s shown in Figure 5. The test pattern generator is initialized to $\overset{\text{(lsb)}}{0}0000010000010001\overset{\text{(msb)}}{}$. After seventeen clock cycles, the test pattern generator is available for use.

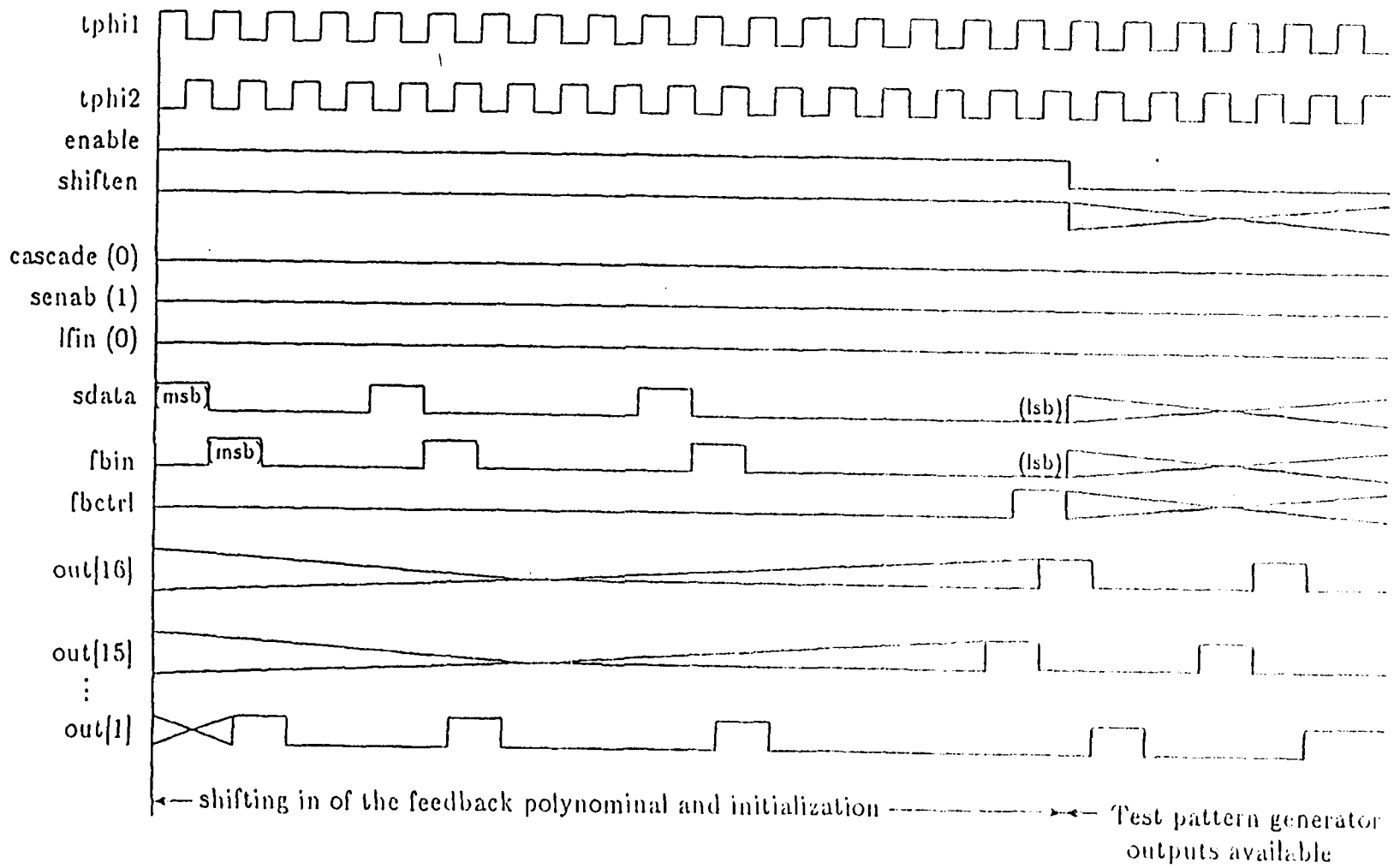


Figure 5. Timing Diagram for the Operation of the Test Pattern Generator Module

Use of the Test Pattern Generator Module in a Design

Independent of the BIT technique used in the design, the test pattern generator module can be used to provide pseudorandom inputs to a unit under test (UUT). Note that the UUT can be one or multiple ambiguity groups (AGs), depending on the BIT technique used. The applicability of pseudorandom patterns in testing the UUT must be verified before actually using the test pattern generator module. Figure 6 shows a typical application of the test pattern generator module.

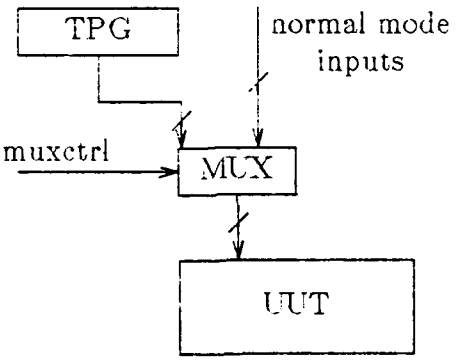


Figure 6. A Typical Application of the Test Pattern Generator Module

In applications where only serial inputs are available for shifting in test vectors, the configuration in Figure 7 can be used.

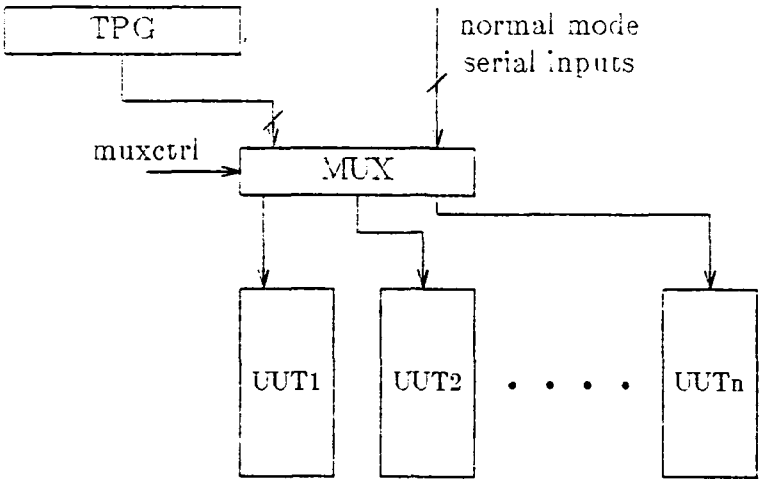


Figure 7. Another Typical Application of the Test Pattern Generator Module

References

1. Smith,J.E. "Measures of Effectiveness of Fault Signature Analysis," *IEEE Transactions on Computers*, **C-29 No.6**, pp. 510-514 (June 1980).

PROGRAMMABLE FEEDBACK PSEUDORANDOM TEST PATTERN GENERATOR (INTERNAL EXCLUSIVE-OR IMPLEMENTATION) APPLICATION NOTE

Objective

The purpose of this application note is to describe the functionality of the programmable feedback pseudorandom test pattern generator module using an internal exclusive-or feedback structure.

Block Diagram

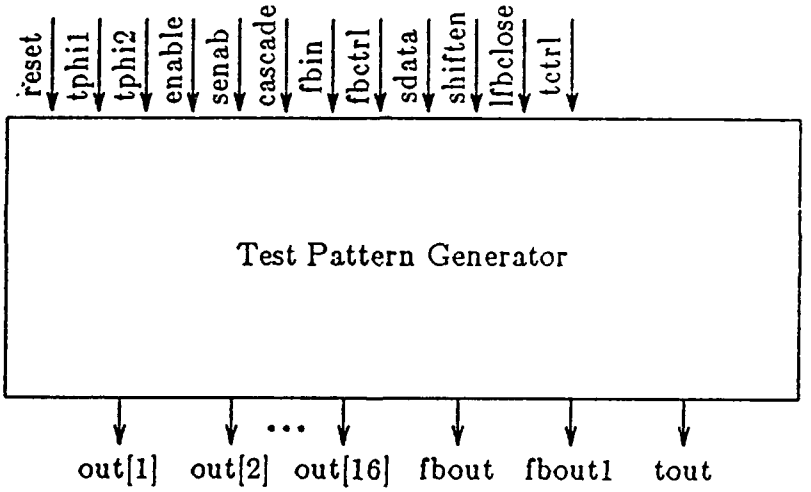


Figure 1. Block Diagram of a Pseudorandom Test Pattern Generator Module

Pin Description (see also the logic diagram and module functions)

Name	Description
reset	Master reset of all the flipflops
tphi1/tphi2	Two-phase clock
enable1	Shift enable1 for the control inputs
tctrl	Tristate control for "tout" output
senab	Enable control for the serial data (used in cascading multiple test pattern generator modules)
cascade	Data input used in cascading multiple test pattern generator modules
fbin	Feedback coefficient input
fbctrl	Control input to enable1 feedback connection
sdata	Serial data input
shiften	Shift enable1 for the data in the test pattern generator
lfbclose	Input pin used in cascading multiple test pattern generator modules
out [16:1]	Data outputs of the test pattern generator
fbout	Feedback coefficient output
fbout1	Feedback coefficient output for test control purposes
tout	Serial test data output

Logic Diagram

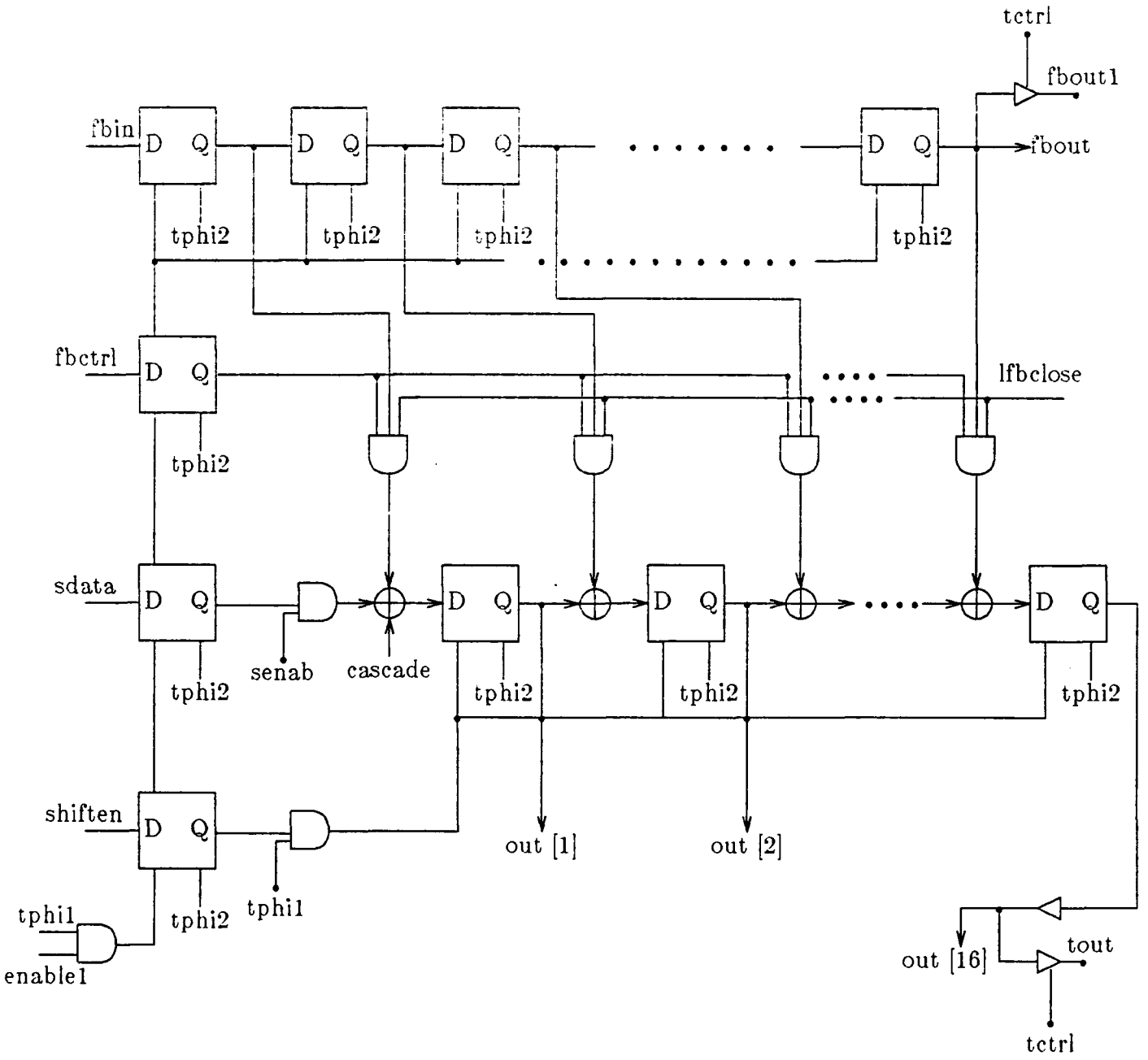


Figure 2. Logic Diagram of the Test Pattern Generator Module

Module Function

The primary function of this module is to generate pseudorandom test vectors for application to a unit under test (UUT). All the required control circuitry is incorporated within the module itself, so that the master test control unit (TCU) can properly control the operation of multiple test pattern generators in a design with minimal need for additional "glue" logic.

Required Pin Connections

Certain pin connections are necessary to use the test pattern generator module. If the module is to be used by itself, i.e., as a 16-bit test vector generator, the connections shown in Figure 3 are required.

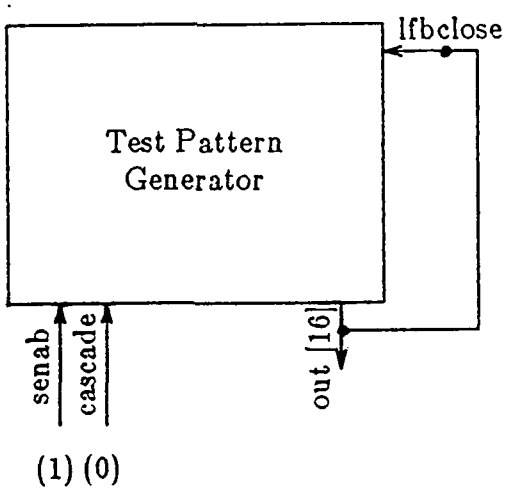


Figure 3. Required Pin Connections to Use a Single Test Pattern Generator Module

If multiple test pattern generator modules are to be used as a **single programmable feedback test pattern generator**, the connections shown in Figure 4 (illustrated for three modules) are required.

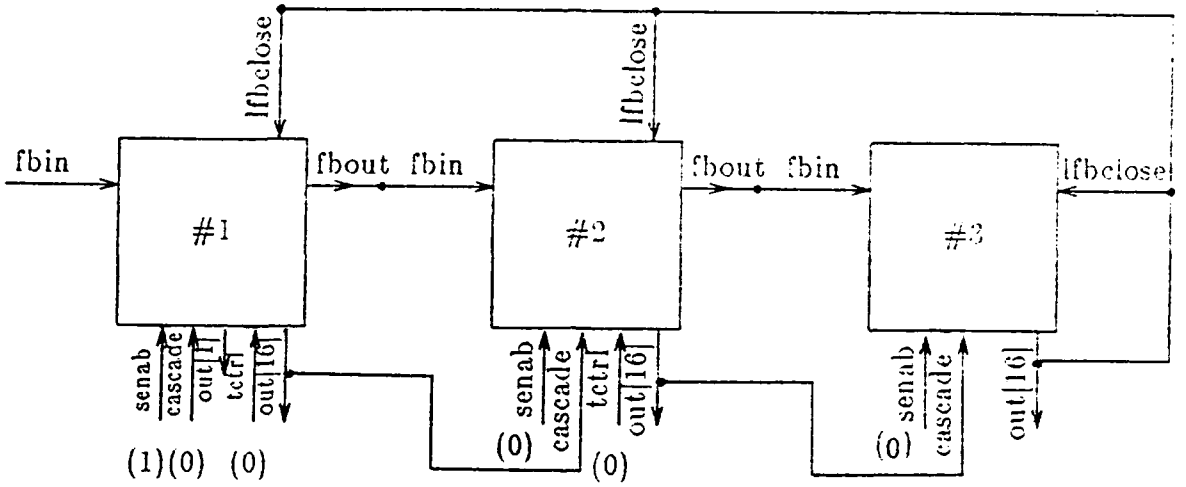


Figure 4. Required Pin Connections to Use Multiple Test Pattern Modules as an Equivalent Single Module

The terms "enable," "fbctrl," "sdata," and "shiften" are common to all the test pattern generator modules in this mode of operation.

Unlike the feedback path in the external exclusive-or implementation (which is described in a separate application note), the feedback path in this implementation does not involve gate delays. Hence, a potentially higher speed of operation (higher test clock frequency) is possible provided the driver associated with the most significant bit of the test pattern generator is designed properly. However, when multiple test pattern generator modules are cascaded as in Figure 4, speed of operation will be affected to a certain extent because of higher loading on the "out [16]" node

corresponding to the most significant bit of the test pattern generator "out[16]" node of #3 in Figure 4).

Operation of the Module

All the test pattern generator modules in the design are reset by a master "reset" signal common to all the modules. Then, each module is selected by means of its "enable1" signal. Predetermined test patterns are shifted through the feedback coefficient and data shift registers through "fbin" and "sdata" inputs respectively. The outputs at "fbout1" and "tout" pins are verified against the expected values. These tests are performed for "shiften"= 1 and "fbctrl"= 0, "shiften"= 1 and "fbctrl"= 1, and "shiften"= 0 control signal values. If any of these tests fail, the test pattern generator module could be faulty and might have to be replaced.

Then, the coefficients (binary 1 and 0) corresponding to the desired feedback polynomial are shifted in through "fbin" input, and, simultaneously, test pattern generator data outputs are initialized to a given not-all-zero state by shifting data through "sdata" input. During this initialization period, "shiften"= 1 and "fbctrl"= 0 logic values are maintained. After a predetermined number of clock cycles, "fbctrl" is raised high and then "enable1" is brought low in the succeeding clock cycle so that the feedback coefficients and the control signals are latched. From this point on, pseudorandom test vectors are available at "out[16:1]" outputs every clock cycle. The repetition interval of the generated test patterns is dependent on the feedback polynomial chosen, and a maximal repetition interval of 2^{16} - 1 is obtained by choosing a primitive feedback polynomial [1]. A typical timing sequence to shift in a feedback

polynomial. $^{(lsb)}_{1}$ 0000010000100$0^{(msb)}_{1}$, is shown in Figure 5. The test pattern generator is initialized to $^{(lsb)}_{0}$ 0000010000100$0^{(msb)}_{1}$. After seventeen clock cycles, the test pattern generator is available for use.

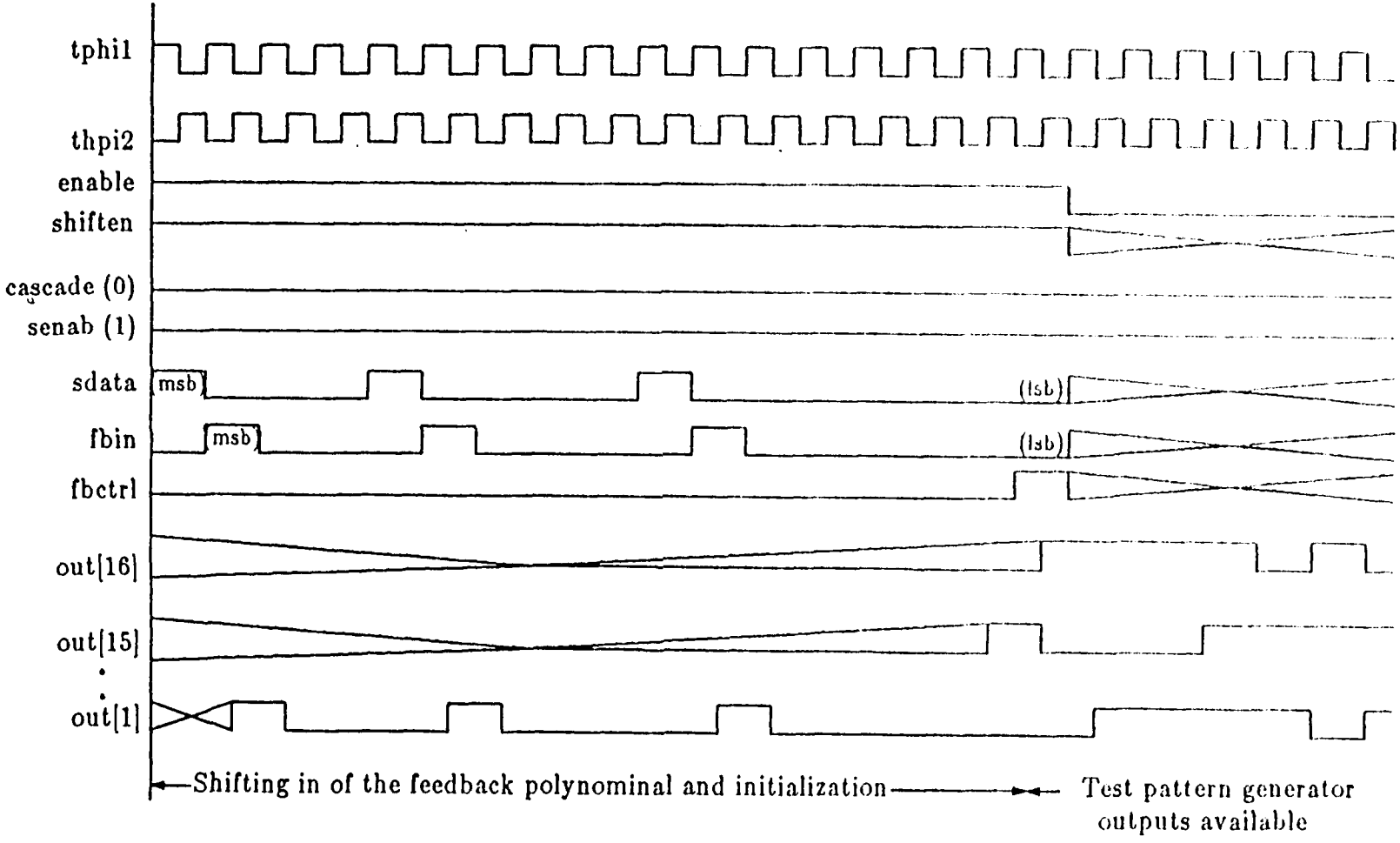


Figure 5. Timing Diagram for the Operation of the Test Pattern Generator Module

Use of the Test Pattern Generator Module in a Design

Please see the corresponding subsection in the application note on "Programmable Feedback Pseudorandom Test Pattern Generator (External Exclusive-or Implementation)."

References

1. Smith, J.E. "Measures of Effectiveness of Fault Signature Analysis." *IEEE Transactions on Computers*, **C-29 No.6**, pp. 510-514 (June 1980).

SCAN-SET BIT MODULE

APPLICATION NOTE

Objective

The objective of this application note is to describe the functionality of the scan-set BIT module, and to show how it is inserted in a design in order to facilitate test.

Block Diagram

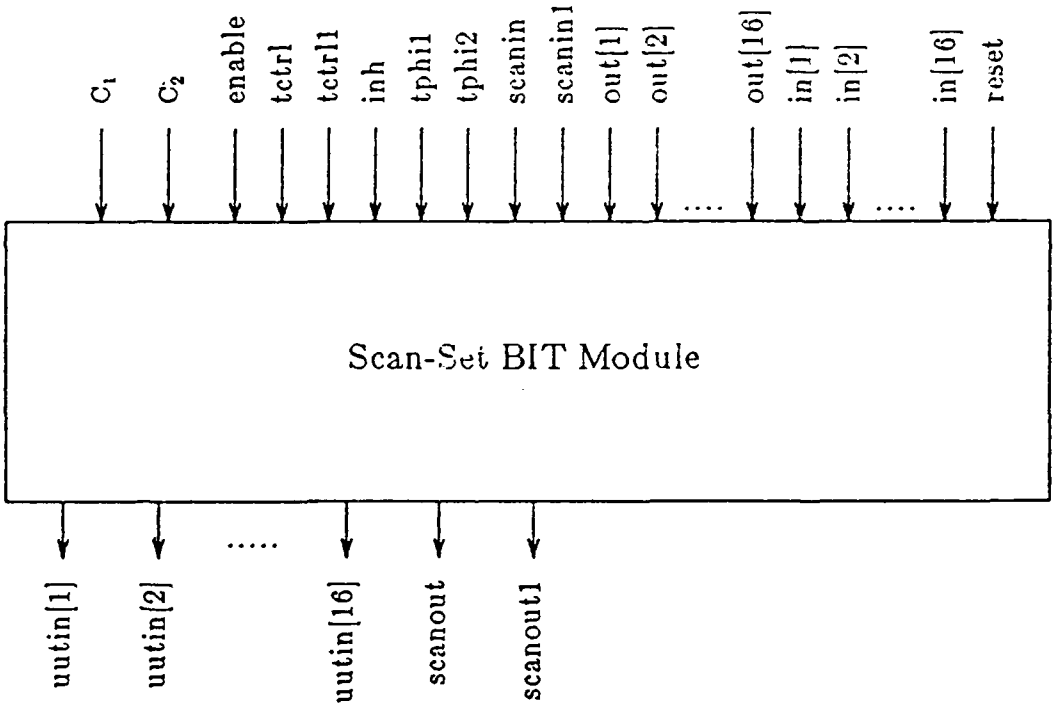


Figure 1. A Block Diagram of the Scan-set BIT Module

Pin Description (see also the logic diagram and module functions)

Name	Description
C_1, C_2	Mode control inputs
enble	Shift enable for control and data inputs
tctrl, tctrl1	Tristate controls for "scanout" and "scanout1" outputs
inh	Inhibit signal that causes the outputs uutin[16:1] to be 0 when active (1)
tphi1,tphi2	Two-phase clock
scanin. scanin1	Serial data inputs
out[16:1]	16 parallel input pins that will be connected to the outputs of the ambiguity group (AG) under test
in[16:1]	16 parallel input pins that will be connected to the outputs of an AG corresponding to normal mode of operation
uutin[16:1]	16 parallel output pins that will be connected to the inputs of the AG under test
scanout, scanout1	Serial data outputs
reset	Master reset of all the flip-flops

Logic Description

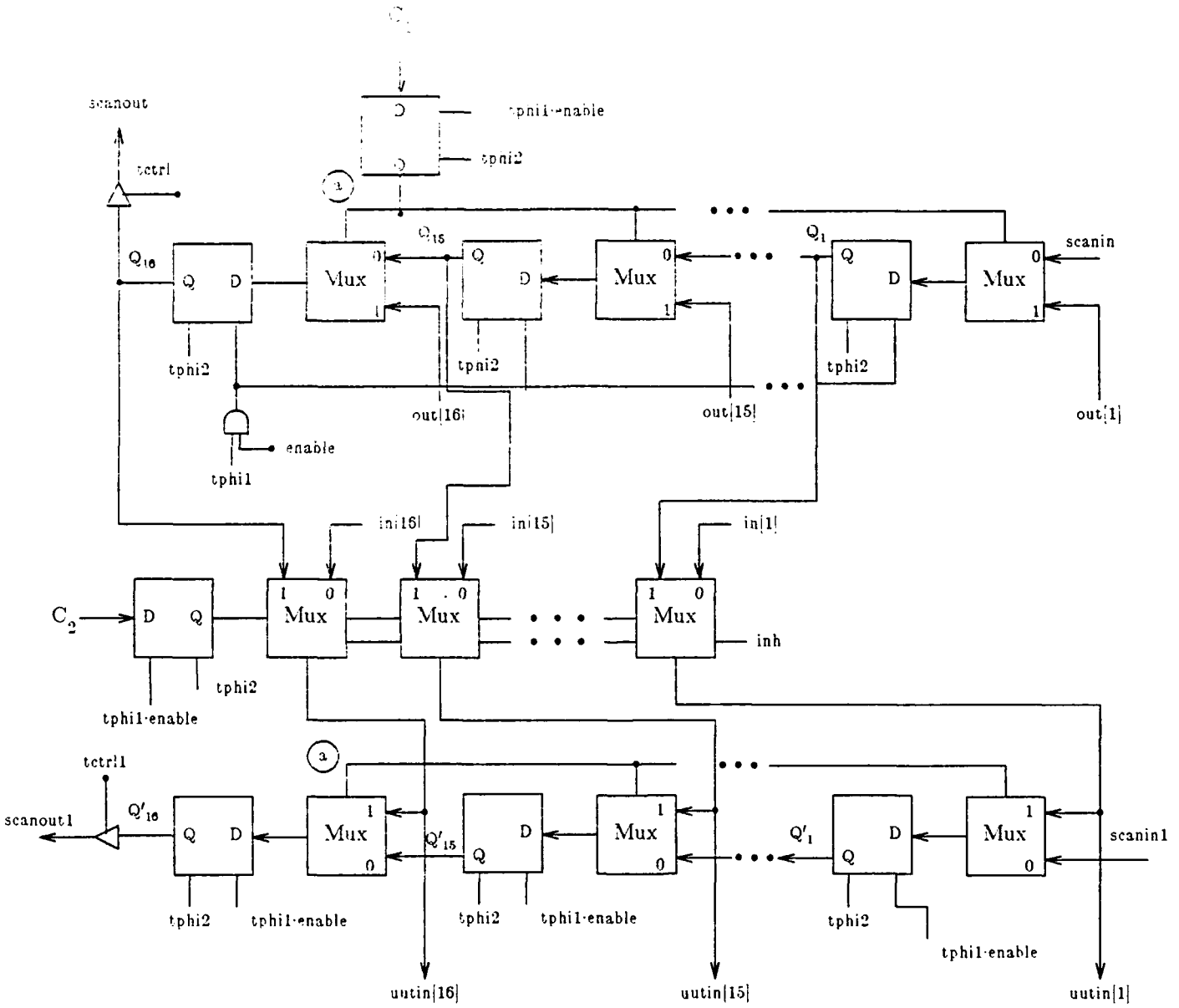


Figure 2. Logic Diagram of the Scan-set BIT module

Module Functions

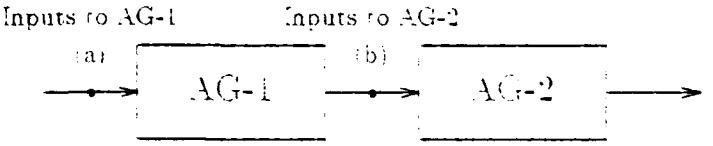
enble	C_1	C_2	inh	tctrl	tctrl1	
1	1	0/1	0/1	0/1	0/1	Parallel transfer of out[16:1] to Qs and uutin[16:1] to Q's
1	0	0/1	0/1	0/1	0/1	Serial shifting-in of data from "scanin" and "scanin1"
1	0/1	1	0	0/1	0/1	Q outputs are connected to uutin[16:1]
1	0/1	0/1	1	0/1	0/1	uutin[16:1] = 0
1	0/1	0	0	0/1	0/1	in[16:1] connected to uutin[16:1]
0/1	0/1	0/1	0/1	0	0/1	Scanout = Z (high impedance)
0/1	0/1	0/1	0/1	1	0/1	Scanout = Q_{16}
0/1	0/1	0/1	0/1	0/1	0	Scanout1 = Z (high impedance)
0/1	0/1	0/1	0/1	0/1	1	Scanout1 = Q'_{16}

enable = 0 latches the control inputs C_1 and C_2 and disables shifting of data in the shift registers.

Incorporation of the BIT Modules into Design

A hardware design will be partitioned into Ambiguity Groups (AGs) of interconnected chips before the BIT modules are inserted in the design. Insertion of the scan-set BIT modules is illustrated using the example in Figure 3.

Before BIT Insertion



After BIT Insertion

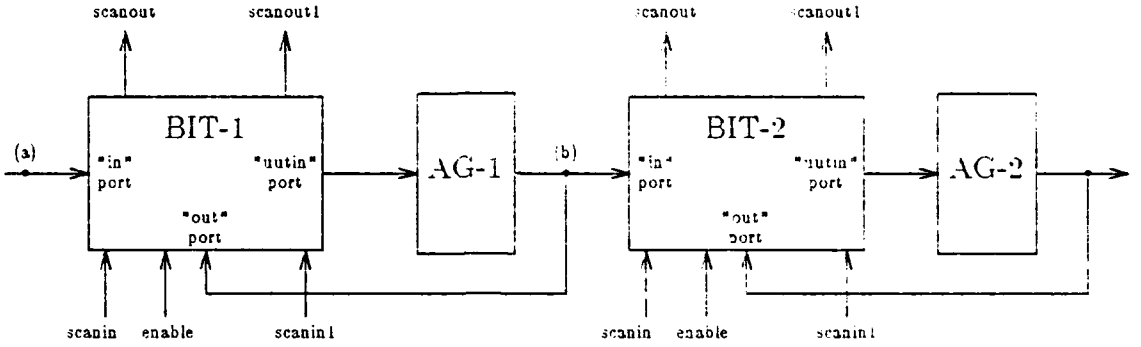


Figure 3. An Example Showing the Insertion of the Scan-set BIT Modules in a Design

As evident from the figures, all inputs to an AG pass through the BIT module(s) associated with that AG, and all the AG outputs will be connected to the "out" port(s) of the associated BIT module(s).

Another example illustrating the insertion of the scan-set BIT modules is shown in Figure 4.

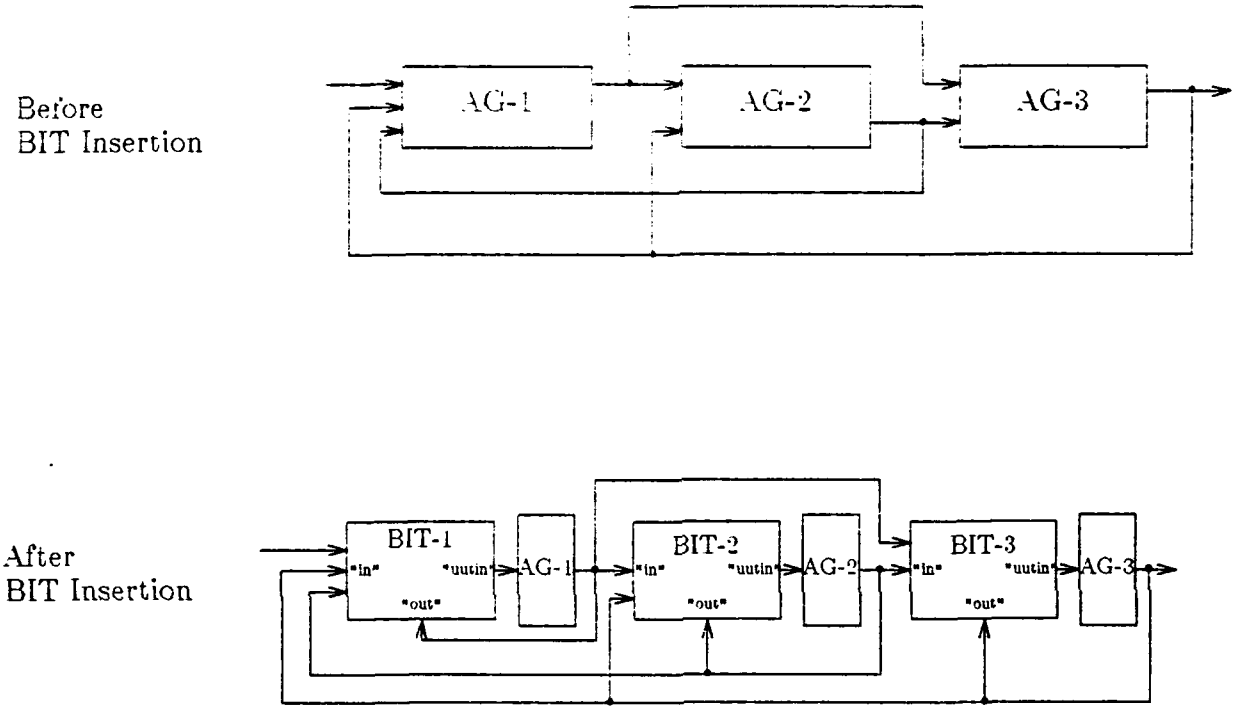


Figure 4. Another Example Illustrating the Insertion of the Scan-set BIT Modules in a Design

A master Test Control Unit (TCU) is assumed to be incorporated into the design in order to perform the following functions:

1. Checking the functionality of the BIT modules;

2. Providing either deterministic or pseudorandom test patterns to the AG under test by shifting in data through "scanin" inputs of the associated scan-set BIT modules;

3. Collecting test response data through "scanout" outputs of the BIT modules, and performing the necessary data compression and/or comparison with the expected responses;

4. Providing all the necessary control signals, such as "C_1", "C_2", and "enable", for the BIT modules;

5. Exercising control over the AG clocks (phi1/phi2 assumed) and the BIT module clocks (tphi1/tphi2).

Before discussing a step-by-step procedure for testing the AGs, the following issues are to be considered:

1. It may be preferable to treat the control inputs of an AG differently from its data inputs, and to use different scan-set modules in conjunction with control and data inputs, as shown in Figure 5.

   This feature facilitates the application of deterministic test patterns at the control inputs of an AG while allowing pseudorandom patterns to be applied at its data inputs. Even though the logic diagram shown for the scan-set BIT module assumes a word size of 16 bits, in practice, modules with smaller word sizes (4 bits, 8 bits, etc.) may also be utilized in the designs in order to reduce the unused pins of the BIT module.

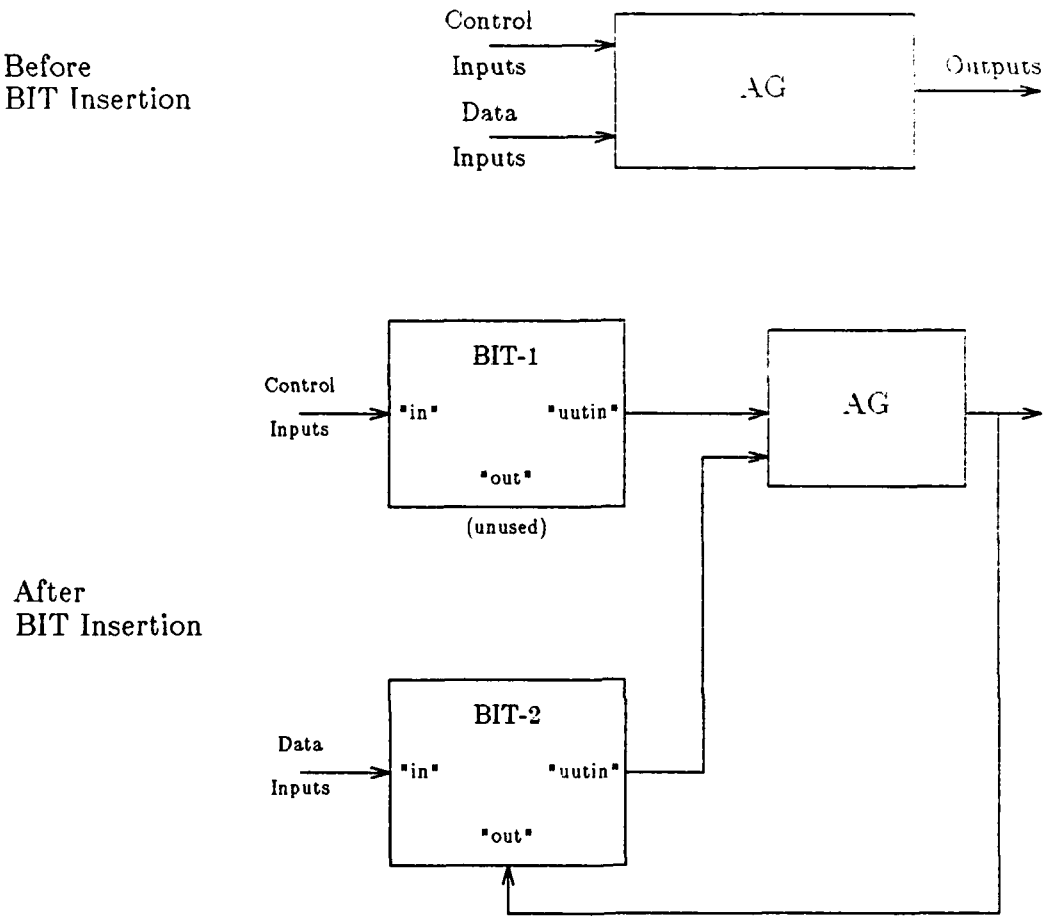


Figure 5. An Example Illustrating the Use of Different BIT Modules for Control and Data Inputs of an AG

It is also conceivable to pass only the data inputs of an AG through a scan-set BIT module while allowing the control inputs to be directly connected to the AG, as shown in Figure 6.

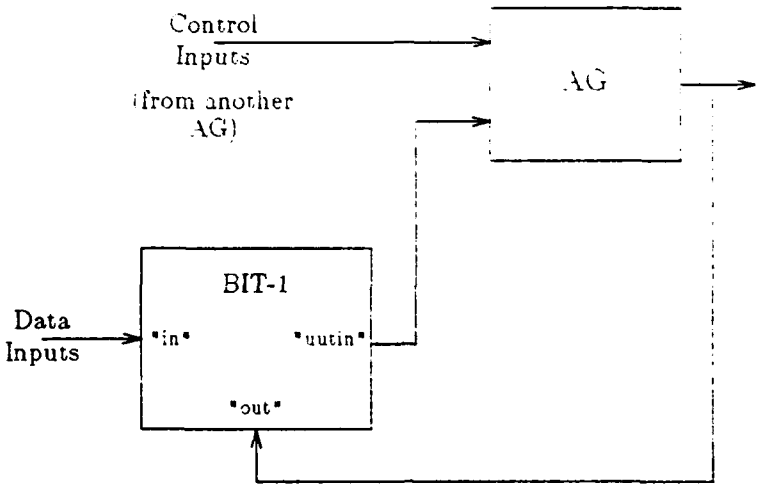


Figure 6. An Example in which a BIT Module is Used only for the Data Inputs of an AG

In such a scheme, it is generally assumed that the AGs that generate the control inputs are tested first, and their outputs are set to the desired values during the testing of other AGs. The feasibility of properly implementing this scheme depends on several factors, such as the design of the AGs in the system, required fault isolation resolution, etc., and it is not considered in the step-by-step procedure given later.

2. If the number of inputs or outputs of an AG exceed the default word size of the BIT module, multiple modules need to be used. One way of interconnecting multiple scan-set BIT modules is shown in Figure 7.

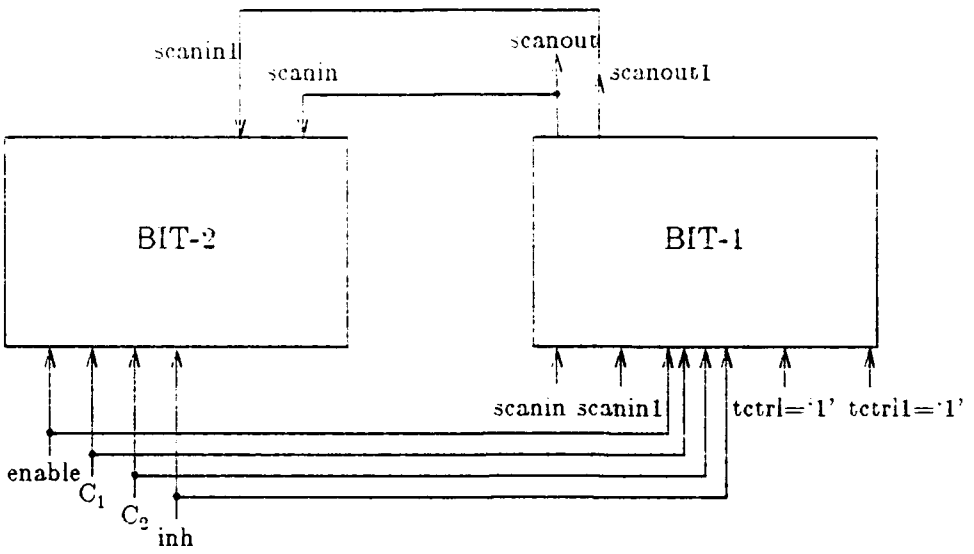


Figure 7. Use of Multiple Scan-set BIT Modules

In this example, "scanout" and "scanout1" of BIT-1 are connected to "scanin" and "scanin1", respectively. Furthermore, "tctrl" and "tctrl1" of BIT-1 are tied to '1'. "C_1", "C_2", "inh", and "enable" are common to both modules. ("Reset", "tphi1", and "tphi2" are always common to all scan-set BIT modules.)

A Procedure for Testing of AGs

Preliminary Steps

1. Reset all the BIT modules in the design by applying a master "reset" signal.

2. Bring all the AGs into predetermined and well-defined states by means of global control signals such as "reset" and "preset", and then disable the clocks of the AGs (phi1/phi2 clocks assumed) from further altering their state.

3. Select each BIT module by means of its "enable" signal, shift in predetermined data (e.g., alternating ones and zeros) through "scanin" and "scanin1" inputs under the control of tphi1, tphi2 clocks, and verify the output data at "scanout" and "scanout1" pins. If there is an error in the scanned-out data, the BIT module itself could be faulty and may have to be replaced.

4. Change the control at C_1 (C_1=1) and perform the parallel transfer of data from "out[16:1]" to Qs, then set C_1=0 again and verify the stored data by serially shifting it through "scanout". In this context, both reset and preset capabilities of an AG may be required so that "out[16:1]" pins can assume both logic '1' and '0'. Similarly, perform the parallel transfer of data from "in[16:1]" to Q's by setting C_1=1 and C_2=0, and verify the stored data by serially shifting it out through "scanout1" (C_1=0). If the scanned-out data is erroneous, fault isolation can be achieved to the combination of an AG, a set of associated scan-set BIT modules, and the inherent interconnections, under the assumption that any, but only one, AG can be faulty during testing.

AG Testing

5. Shift in a predetermined control data pattern into the BIT module associated with the control inputs of the AG under test; then, by disabling further shifting, the AG is set up for test under a known control mode.

6. Shift in a test data pattern through "scanin" of the BIT module associated with the data inputs of the AG (C_1=0, C_2=1). After a predetermined number of tphi1/tphi2 clock cycles, the data pattern is ready for parallel transfer to the AG under test. Then, enable the AG clocks (phi1/phi2) for one clock cycle and disable them again.

7. If the AG outputs are to be verified at this time, set the control signal C_1 (C_1=1) and perform parallel transfer of "out[16:1]" to Qs. Set C_1=0 and shift out the stored data through "scanout", then verify the output data against the expected response data. If the AG outputs need not be verified at this time, then skip this step.

8. Repeat steps 6 and 7 as many times as needed (determined from a priori simulations of the AG).

9. Repeat steps 5 through 8 as many times as needed (i.e., for different control modes of the AG).

The above AG testing procedure will be applied sequentially to all the AGs, i.e., the AGs will be tested one after another using steps 5 through 9. If test of an AG fails (i.e., incorrect response) fault isolation is to the combination of the AG, the BIT module(s) associated with its I/O, and the inherent interconnections, under the assumption that any, but only one, AG can be faulty during testing.

Timing Consideratons

The mode control signals (C_1 and C_2) have to be valid one clock cycle before the actual data. For instance, parallel transfer of "in[16:1]" to Q′ outputs and serial shifting-out of that data through "scanout1" involves the timing sequence shown in Figure 8.

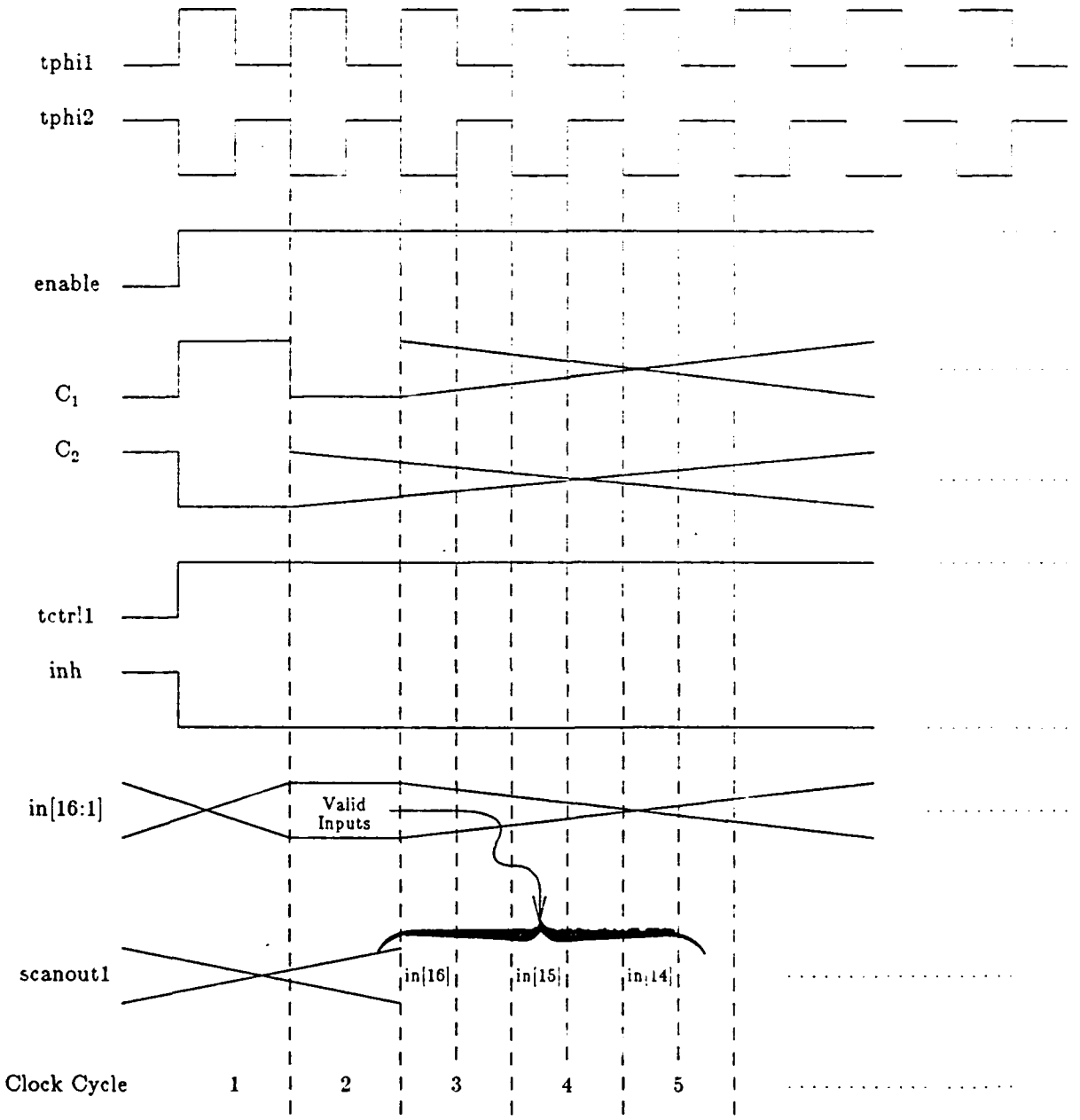


Figure 8. A Timing Sequence of the Signals in the Scan-set BIT Module

As can be inferred from Figure 8, the control mode, C_1=1 and C_2=0, is set up one clock cycle before the actual data inputs "in[16:1]" are valid. The parallel inputs "in[16:1]" will be available (i.e., stable) at "scanout1" serial output beginning at clock cycle three. A similar timing sequence must be used in order to perform a parallel data transfer from "out[16:1]" to Q outputs and to perform further serial shifting of the data through "scanout" output.

TESTING-SWITCH

APPLICATION NOTE

Objective

The objective of this application note is to describe the functionality of the testing-switch module and to show how it is used in a design to facilitate test.

Block Diagram

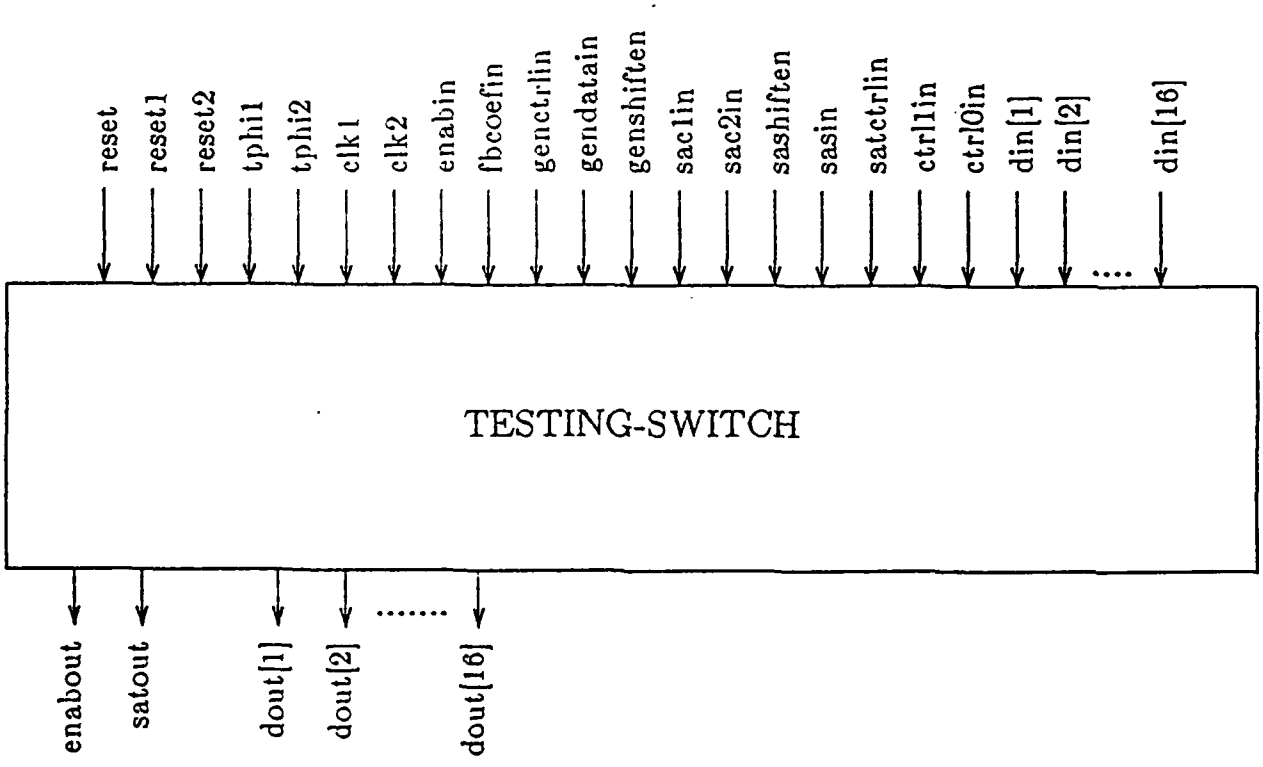


Figure 1. Block Diagram of the Testing-switch Module

Pin Description (see also the Logic Diagrams and Module Functions)

Name	Description
reset	Reset input of all the flip-flops excluding the ones corresponding to "sashiften" and "enabin" inputs
reset1	Reset input of the flip-flop corresponding to the "sashiften" input
reset2	Reset input of the flip-flop corresponding to the "enabin" input
tphi1/tphi2	Two-phase clock
clk1/clk2	Two-phase clock for the flip-flop corresponding to the "enabin" input
enabin	Shift enable for control inputs
fbcoefin	Feedback coefficient input of the test pattern generator associated with the testing-switch
genctrlin	Control input to enable feedback connection in the test pattern generator associated with the testing-switch
gendatain	Serial data input of the test pattern generator associated with the testing-switch
genshiften	Shift enable for the data in the test pattern generator associated with the testing-switch
sac1in,sac2in	Mode control inputs of the signature analyzer associated with the testing-switch
sashiften	Shift enable for the data in the signature analyzer associated with the testing-switch
sasin	Serial data input of the signature analyzer associated with the testing-switch
satctrlin	Tristate control for "satout" output

ctrl1in,ctrl0in	Mode control inputs of the testing-switch
din[16:1]	Parallel data inputs of the testing-switch
enabout	Output of the flip-flop corresponding to the "enabin" input
satout	Serial test data output of the signature analyzer associated with the testing-switch
dout[16:1]	Parallel data outputs of the testing-switch

Logic Diagrams

Figure 2 shows an overall description of the testing-switch module, while Figure 3 illustrates the logic of the test pattern generator associated with the testing-switch. Figure 4 shows the logic of the signature analyzer associated with the testing-switch. Finally, Figure 5 illustrates the control of the data exchange network.

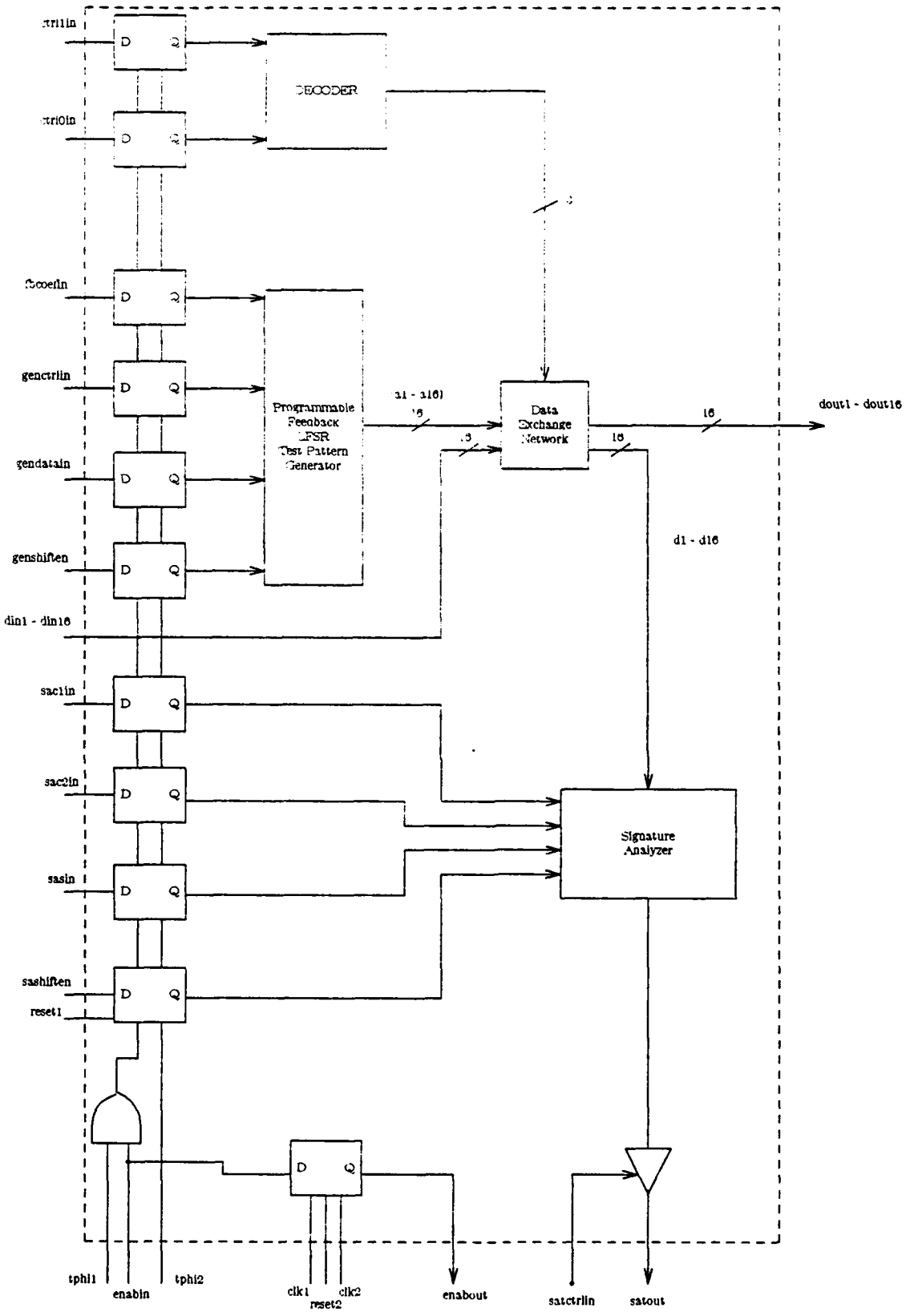


Figure 2. Description of the Testing-switch Module

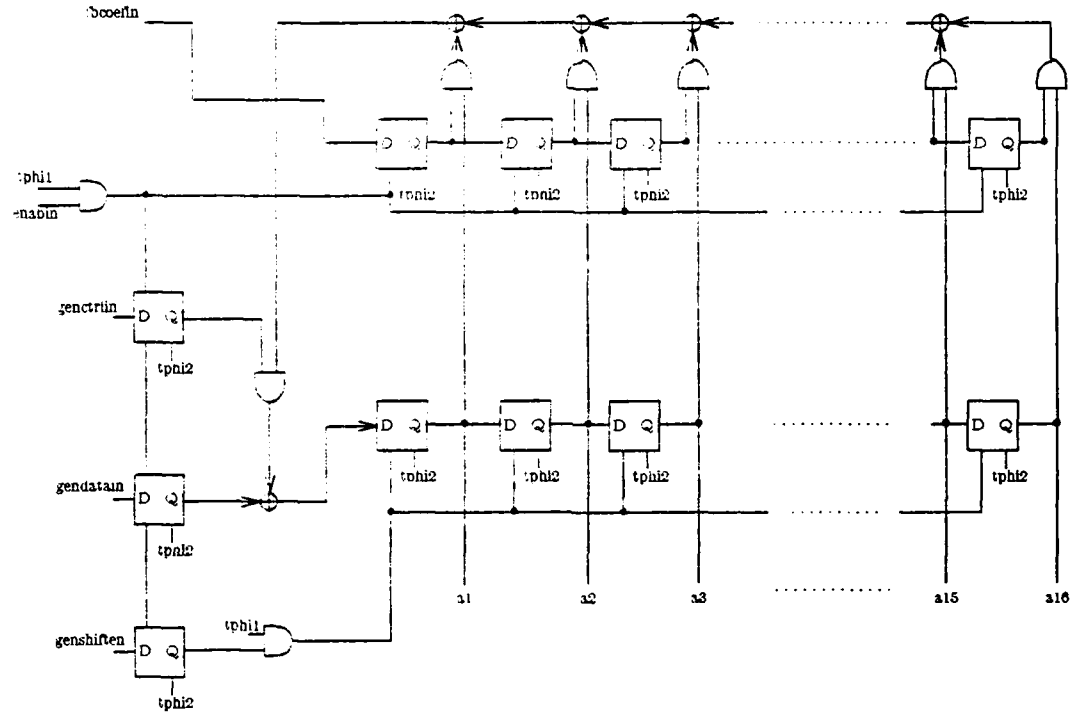


Figure 3. An External Exclusive-or Implementation of the Test Pattern Generator

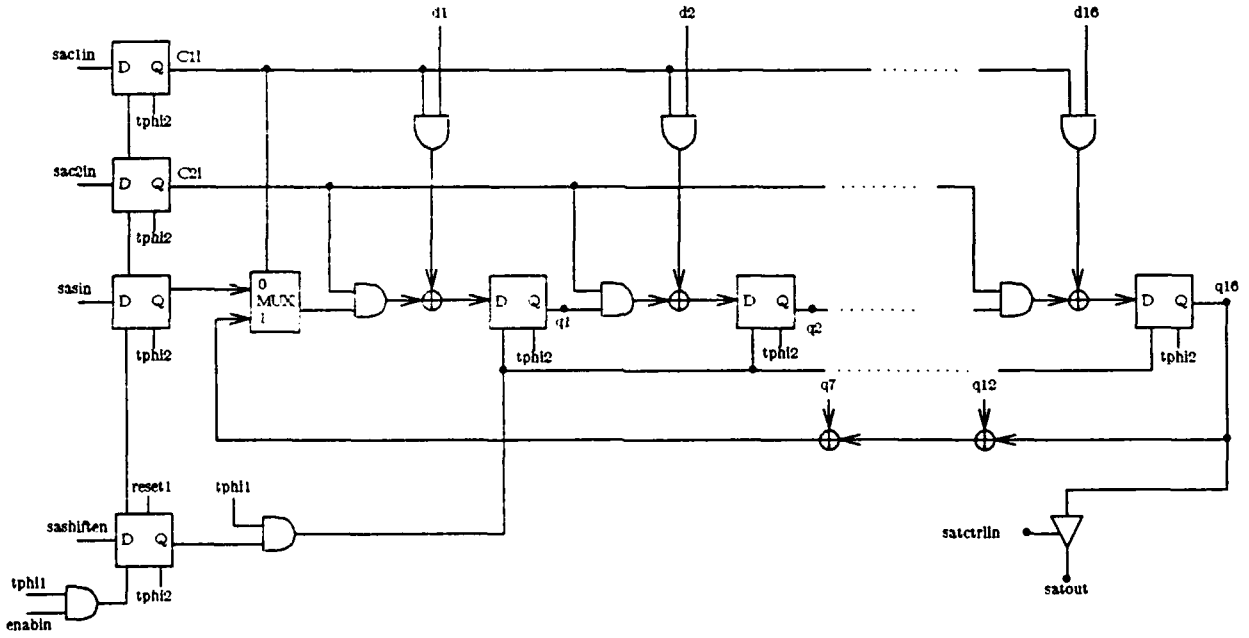
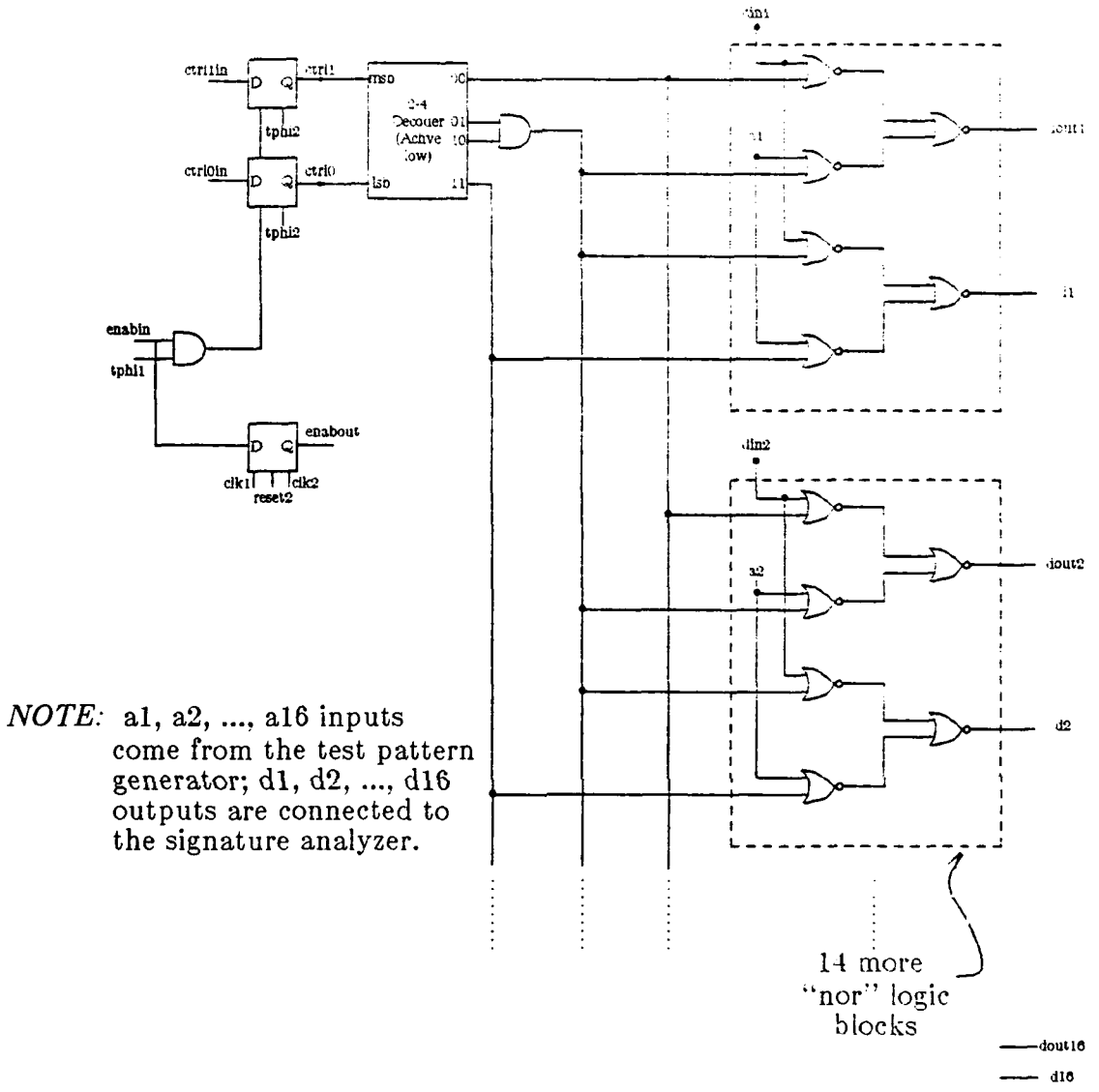


Figure 4. An Implementation of the Signature Analyzer



NOTE: a1, a2, ..., a16 inputs come from the test pattern generator; d1, d2, ..., d16 outputs are connected to the signature analyzer.

Figure 5. Control of the Data Exchange Network

Module Functions

The testing-switch module mainly performs three functions: it provides pseudo-random test patterns to an ambiguity group (AG) (i.e., smallest group of chips to which a fault can be isolated), it compresses an AG output response into a signature,

and it transfers data from input to output without performing any logic operation on it. These functions are illustrated in the following sequence:

ctrl1	ctrl0	Function
0	0	Parallel transfer of inputs from "din[16:1]" to "dout[16:1]"
0	1	Test pattern generator outputs at "a[16:1]" are passed on to the normal data outputs
1	0	"dout[16:1]"; simultaneously, normal data inputs at "din[16:1]" are passed on to the signature analyzer inputs at "d[16:1]"
1	1	During this test mode, test pattern generator outputs at "a[16:1]" are passed on to the signature analyzer inputs at "d[16:1]"

Signature Analyzer Modes of Operation

C1l	C2l	Function
0	1	Serial shifting of data in the signature analyzer
1	1	Parallel data vectors at "d[16:1]" are compressed into a signature at "q[16:1]" outputs, provided the latched value of "sashiften" is high

Incorporation of the Testing-switch Modules into Design

A hardware design will be partitioned into ambiguity groups (AGs) of interconnected chips prior to the incorporation of the testing-switch modules. The example in Figure 6 shows their incorporation in a design.

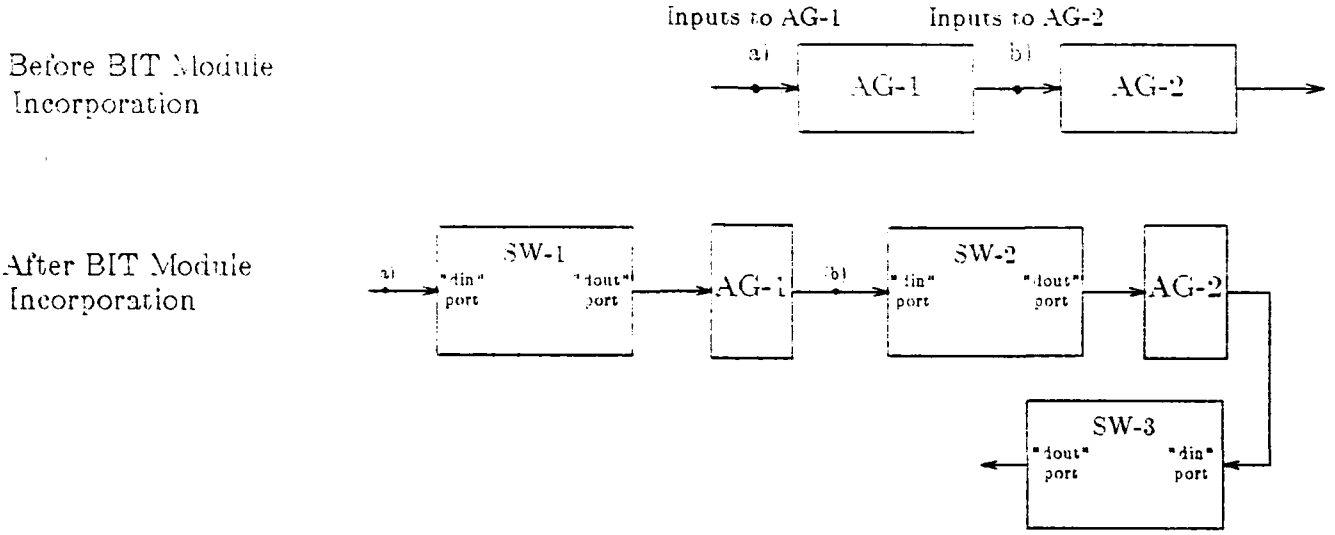


Figure 6. Example Showing the Incorporation of the Testing-switch Modules in a Design

As evident from the figure, all inputs to an AG pass through a testing-switch module, and the AG outputs are connected to the "din" port of another testing-switch module. Another example illustrating the use of the testing-switch modules is shown in Figure 7.

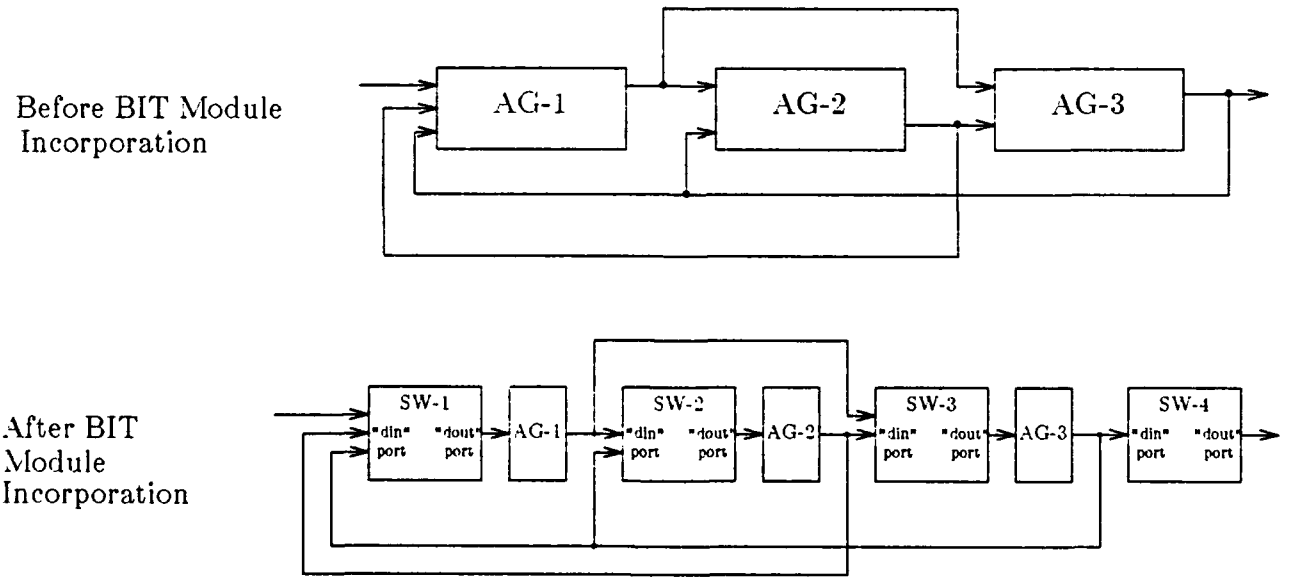


Figure 7. Another Example Illustrating the Incorporation of the Testing-switch Modules in a Design

A test control unit (TCU) is assumed to be present in the system in order to perform the following functions:

1. Checking the functionality of the testing-switch modules;

2. Providing deterministic patterns to the AG under test, if necessary, by shifting in data through "gendatain" input of the relevant testing-switch module(s);

3. Collecting test responses (i.e., final signatures) through "satout" nodes of the testing-switch modules; performing necessary comparisons with the expected signatures; and evaluating the status of the AGs;

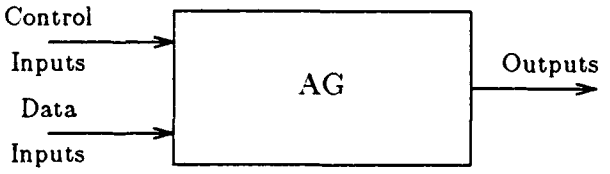
4. Providing all the necessary control signals to properly control the operation of the testing-switch modules;

5. Exercising proper control over the AG clocks and the testing-switch module clocks.

Before discussing a detailed procedure for testing the AGs, the following issues are to be considered:

1. It may be preferable to treat the control inputs of an AG differently from its data inputs and to use different testing-switch modules in conjunction with control and data inputs, as shown in Figure 8.

Before
BIT Insertion



(Figure 8 continued)

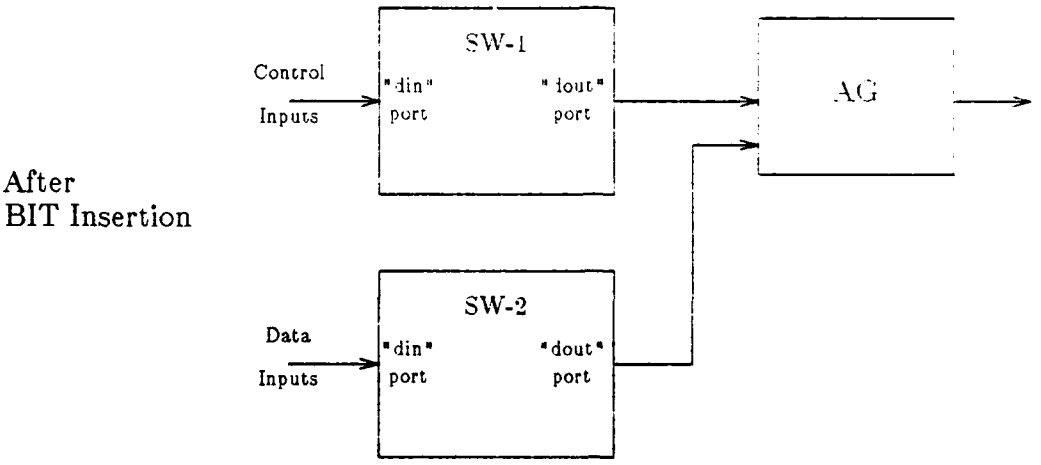


Figure 8. Example Illustrating the Use of Different Testing-switch Modules at Control and Data Inputs of an AG

This feature facilitates the application of deterministic test patterns at the control inputs of an AG, while allowing pseudorandom patterns to be applied at its data inputs. It is also conceivable to pass only the data inputs of an AG through a testing-switch module while allowing the control inputs to be directly connected, as shown in Figure 9.

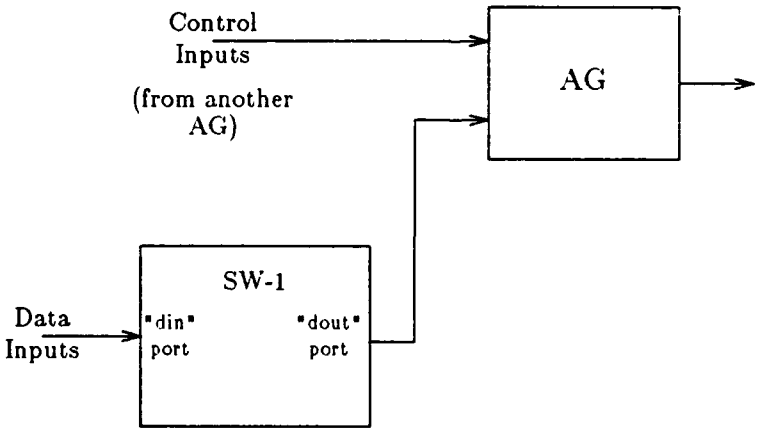


Figure 9. Example in which a Testing-switch Module is Used Only at the Data Inputs of an AG

In such a scheme, it is generally assumed that the AGs that generate the control inputs are tested first, and their outputs are set to desired values during the test of other AGs. The feasibility of properly implementing this scheme depends on factors such as the design of AGs in the system and required fault isolation resolution, and this scheme is not considered in the test procedure given subsequently.

2. If the number of inputs or outputs of an AG exceed the default word-size of the testing-switch, multiple testing-switch modules are required. However, the test pattern generators and the signature analyzers in different testing-switch modules need to be used independently, i.e., they cannot be used to form equivalent larger word-size modules. This is not a limitation in general, since the test pattern generator in Figure 3 has the programmable feedback feature and it can generate maximal length $(2^{16}-1)$ nonrepetitive test vectors. Furthermore, uncorrelated test vectors can be generated by initializing the various testing-switches in the system with different seeds (initial test patterns). Also, the error escape probability associated with the signature analyzer of Figure 4 is acceptably low, i.e., $\leq 2^{-16}$.

3. Proper setting of the testing-switches in the system plays an important role in the overall test process. Such setting of the testing-switches can be in a predetermined order, as shown in Figure 10, or in an arbitrary order, as shown in Figure 11. In Figure 10, "enabin" to "enabout" delay may have to be several tphi1/tphi2 clock periods for proper initialization of the testing-switches. With this requirement in perspective, "clk1" and "clk2" inputs of the enabin-enabout

flip-flop are separated from the "phi1" and "phi2" clocks. Arbitrary-order setting of the testing-switches, in general, requires a decoder as shown in Figure 11. In this scheme, the "enabout" output pin and the "clk1" and "clk2" input pins of the testing-switch are not utilized.

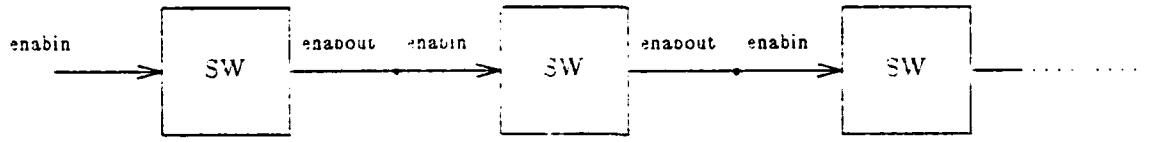


Figure 10. Scheme for Predetermined-order Setting of the Testing-switches

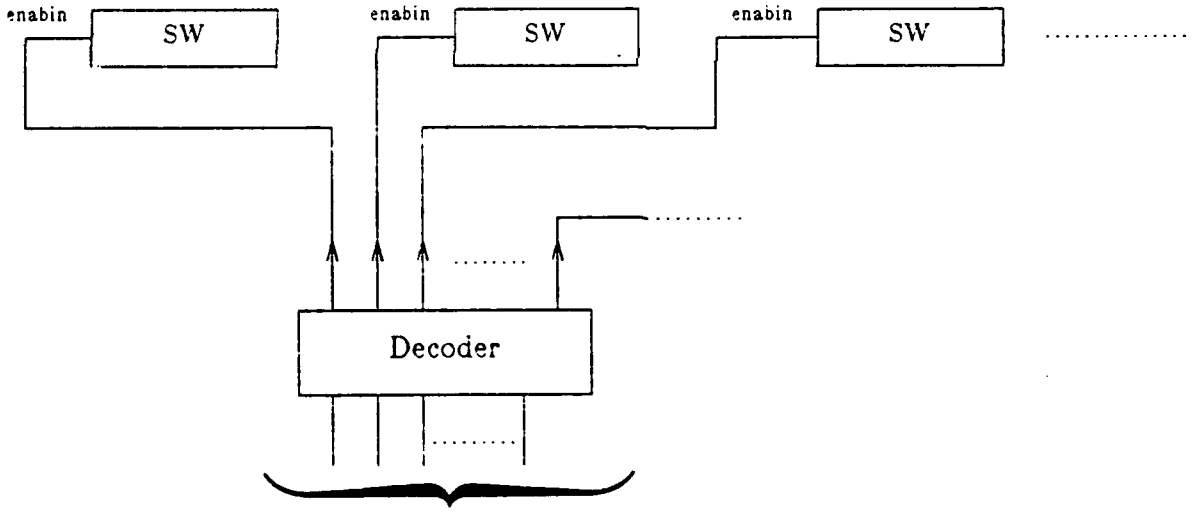


Figure 11. Scheme for Arbitrary-order Setting of the Testing-switches

Procedure for Testing AGs

Preliminary Steps

1. Reset all the testing-switch modules in the design by applying "reset," "reset1," and "reset2" signals.

2. Select each testing-switch by means of its "enabin" signal, initialize the test pattern generator and the signature analyzer, and enable the test pattern generator to signature analyzer path ("ctrl1" = 1, "ctrl0" = 1). After a predetermined number of clock cycles, shift out the signature through "satout" node and check against the expected signature. If an erroneous signature is noticed, the testing-switch module itself could be faulty and might have to be replaced.

3. Perform reset (preset) of all AGs and enable the normal data inputs to signature analyzer path in each relevant testing-switch ("ctrl1" = 1, "ctrl0" = 0 — or "ctrl1" = 0, "ctrl0" = 1). After one clock cycle (i.e., signature analyzer acts as a parallel-in/serial-out register), shift out the signature analyzer contents and check whether the AG outputs are indeed reset (preset). If this test fails, fault isolation is generally to an AG-testing-switch(es) combination under the single faulty AG assumption.

AG Testing

4. Identify the AGs in the design that can be simultaneously tested. For instance, the AGs 1 and 2 in Figure 6 can be simultaneously tested, whereas, the AGs of Figure 7 may not be simultaneously testable. Simultaneous testing of AGs is not always possible because of the requirement that certain AG outputs must be kept at predetermined states while testing other AGs. If simultaneously testable AGs do not exist or have already been tested, then pick a not-yet-tested AG (if one exists) based on a predetermined order.

5. Bring all the AGs into predetermined and well-defined states by means of global control signals, such as reset and preset, and then disable the AG clocks from further altering their state. Select the testing-switches associated with the AG(s) in step 4, one by one, and perform the required initialization of the associated test pattern generators and signature analyzers. The initialization process involves such steps as programming the feedback polynomial of the test pattern generators; shifting in the initial seed; and proper setting of the various control signals, such as "genctrlin," "genshiften," "ctrl1in," "ctrl0in," "sac1in," and "sac2in."

6. Enable the AG clocks, and ensure that they bear appropriate frequency and phase relationship with the test clocks tphi1/tphi2. The pseudorandom test patterns generated by the test pattern generators in the testing-switch modules will be

applied to the relevant AG(s), and their responses will be compressed by the signature analyzers in the testing-switches. Control inputs to AGs can be held at desired logic values by using the disable-shift capability of the test pattern generators in the testing-switch modules.

7. After a predetermined number of clock cycles, apply "reset1" signal and disable shifting in all the signature analyzers. Then, select each relevant signature analyzer by means of "enabin" signal of the associated testing-switch, and shift out the stored signature. Compare this signature with the expected one, obtained a priori from simulation, and determine the status of AG(s).

8. Repeat steps 5 through 7 for all the necessary control modes of the AG(s).

The above AG test procedure is repeated until all the AGs are tested. If the test of an AG fails, fault isolation is generally to the combination of an AG, the associated testing-switches, and the inherent interconnections.

Timing Considerations

The outputs "dout[16:1]" assume the values of "din[16:1]" (normal data inputs), or "a[16:1]" (test pattern generator outputs), or "1" (logic one) with one clock cycle delay after the "ctrl1in" and "ctrl0in" values change. Figure 12 illustrates this aspect.

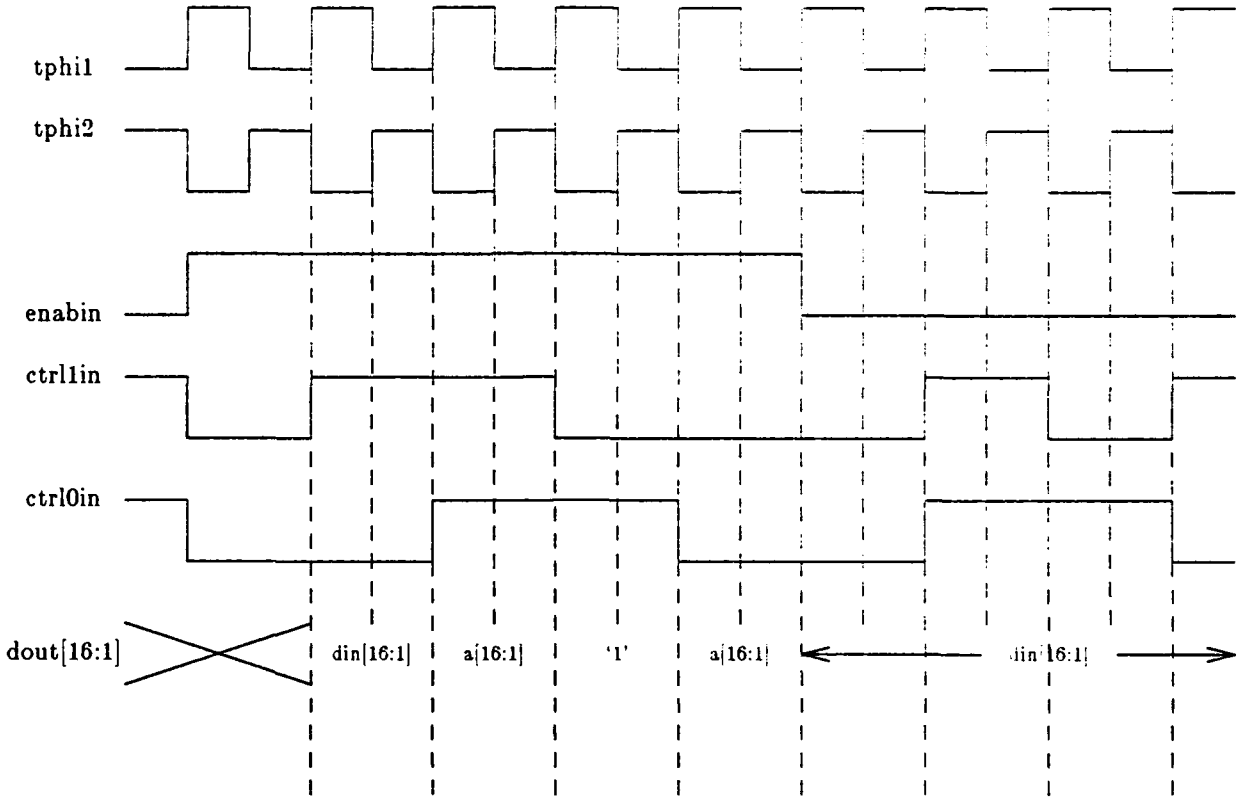


Figure 12. Timing Diagram Illustrating the Operation of the Data Exchange Network of the Testing-switch

ADDITIONAL LOGIC ASSOCIATED WITH THE VERSION-2 BIT MODULES

Objective

The purpose of this note is to describe the additional logic incorporated into the Version-2 BIT modules in order to make them directly compatible with the maintenance node unit (MN) described in the application note entitled *Maintenance Node*.

Block Diagram

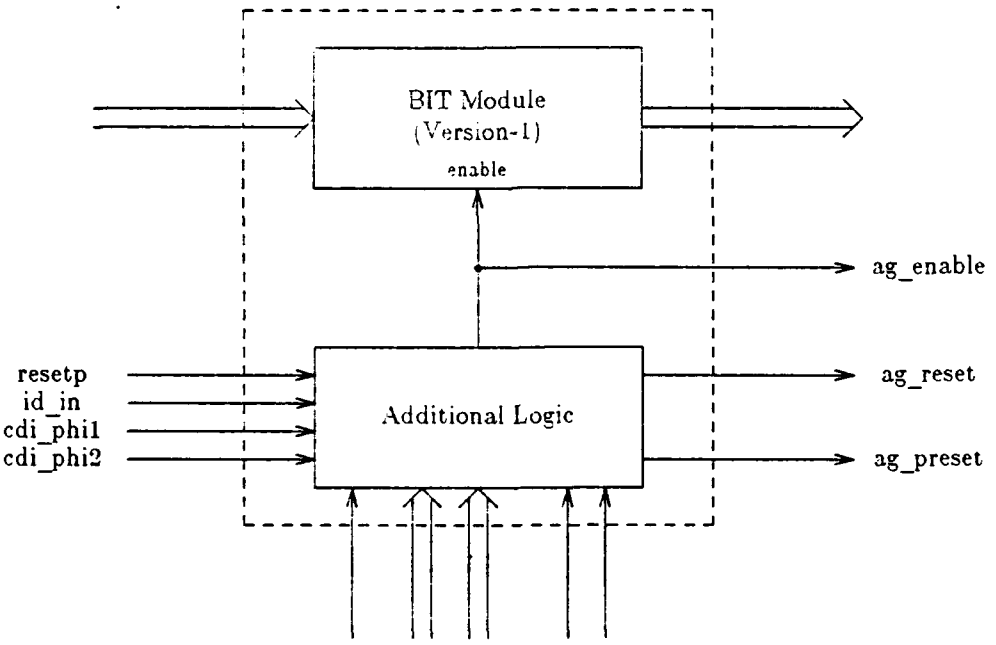


Figure 1. Block Diagram Showing the Inputs and Outputs of the Additional Logic Associated with the Version-2 BIT Modules

Pin Description (See also the logic diagram and the application note on the utilization of the MN)

Name	Description
ag[4:0]	5-bit ambiguity group (AG) id (generally hardwired)
ag_preset	Output signal to indicate that the "preset" of the AG outputs is required
ag_reset	Output signal to indicate that the "reset" of the AG outputs is required
bit_id [2:0]	3-bit BIT module id (generally hardwired)
breset	Board-reset signal generated by the MN
cdi_phi1, cdi_phi2	Non-overlapping phases of the CDI clock generated by the MN
comp	"Compare successful" signal generated by the MN
ag_enable	Output generated by the additional logic in the BIT module. It is connected to the "enable" input of the version-1 BIT module, as shown in figure 1. It may also be connected to the tristate control input(s) of the version-1 BIT module.
id_in	Serial identification-address input
id_load	The "identification_load" signal generated by the MN
resetp	The "power-on" reset of the additional logic block

Logic Description

Figure 2 illustrates the additional logic incorporated, while figure 3 shows a more detailed description used in the CADAT simulation. The CADAT description uses primitives such as "dff-l," "n1mos," "comparator," "decbin" (decoder), and "buf" in addition to the basic gates.

Timing Considerations

Figure 4 shows a data sequence required to be input through "id_in" in order to raise the "enable" signal to high level. Once the "enable" signal becomes high, the BIT module accepts data and control signals through its relevant inputs. The "comp" and "breset" signals are generated by the MN, and it is assumed that the "breset" signal is low for this example. The 2-bit AG preset/reset command code is chosen such that the "ag_reset" output is activated. Finally, all the hardwired id values are assumed to be all "1's" for simplicity. The "enable" signal may be brought to low level when needed by following the same sequence with a different 3-bit BIT module id.

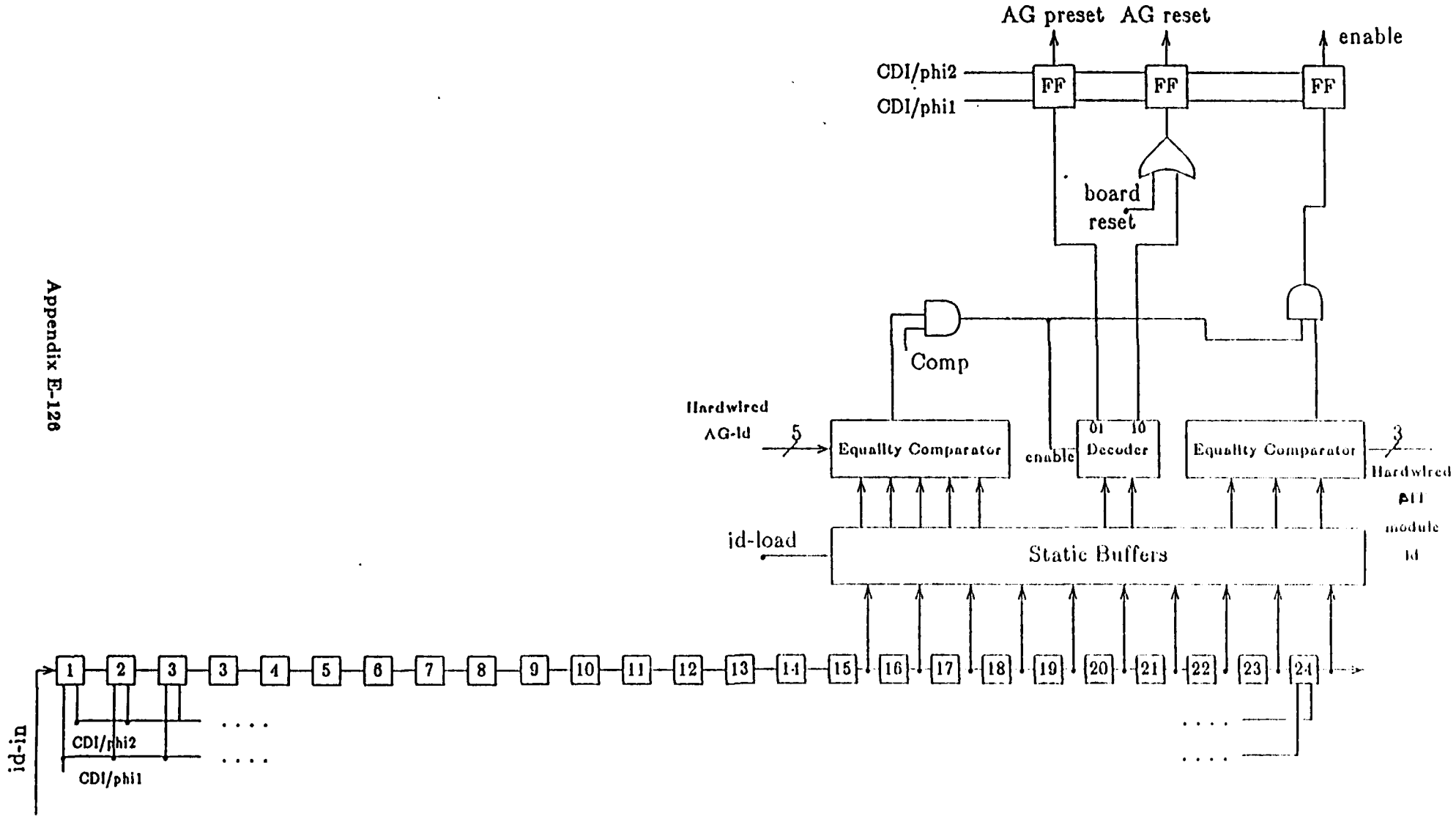


Figure 2. Additional Logic Associated with BIT Modules to Generate the Required Test Signals

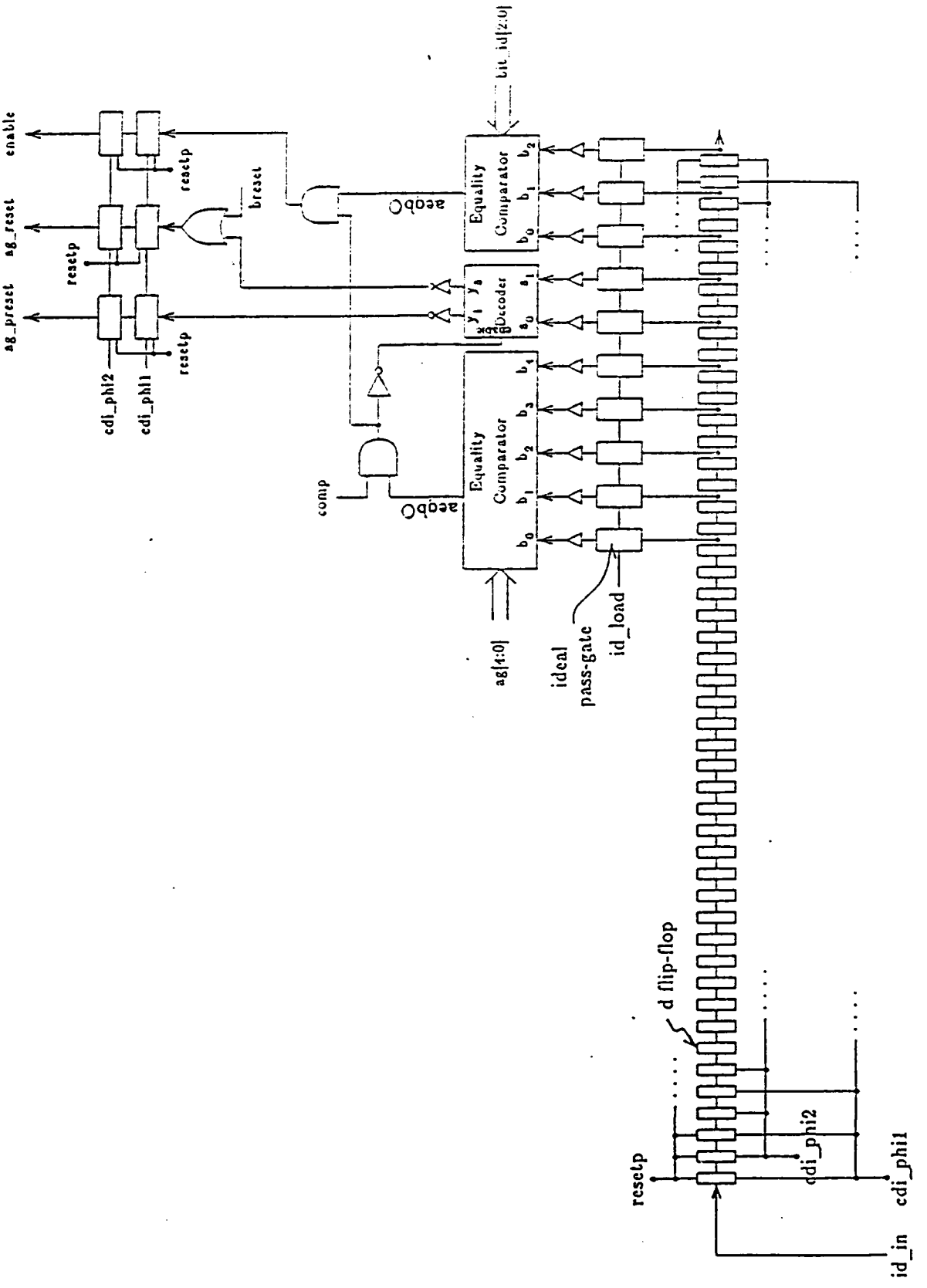


Figure 3. A Detailed Description of the "Additional Logic" Block Used in Version-2 BIT Modules

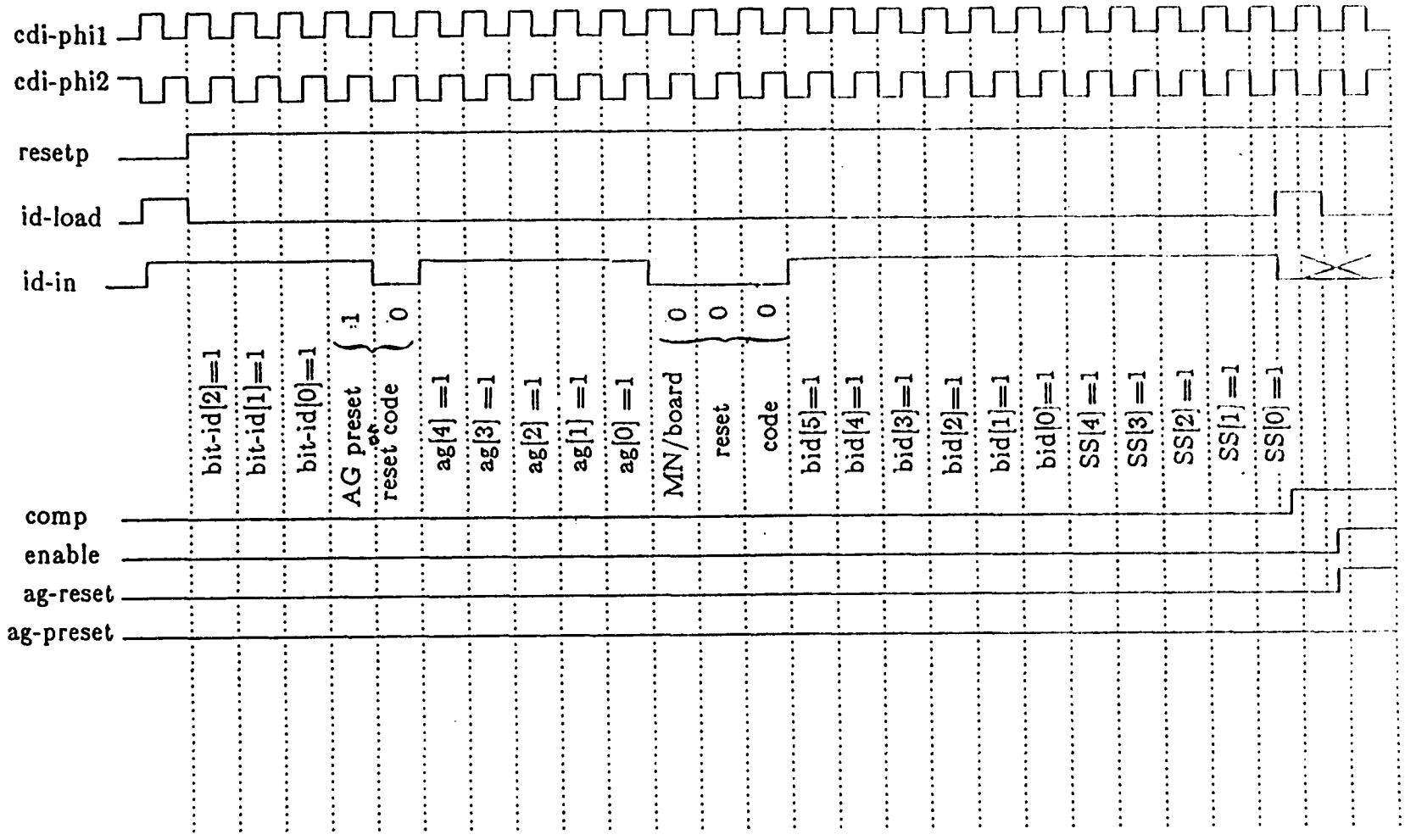


Figure 4. A Timing Diagram Illustrating the Operation of the Additional Logic Associated with the Version-2 BIT Modules

MISCELLANEOUS MODULES TO AID TESTING

Objective

The objective of this application note is to describe a collection of miscellaneous modules which are part of the BIT module library supplied with TEA.

Module 1 : MUX2×8

MUX2×8 is a multiplexor which allows the selection of one of two sets of 8 input lines. This module is shown in Figure 1 and is used in some BIT techniques as a means of selecting either "normal" input or "test" input.

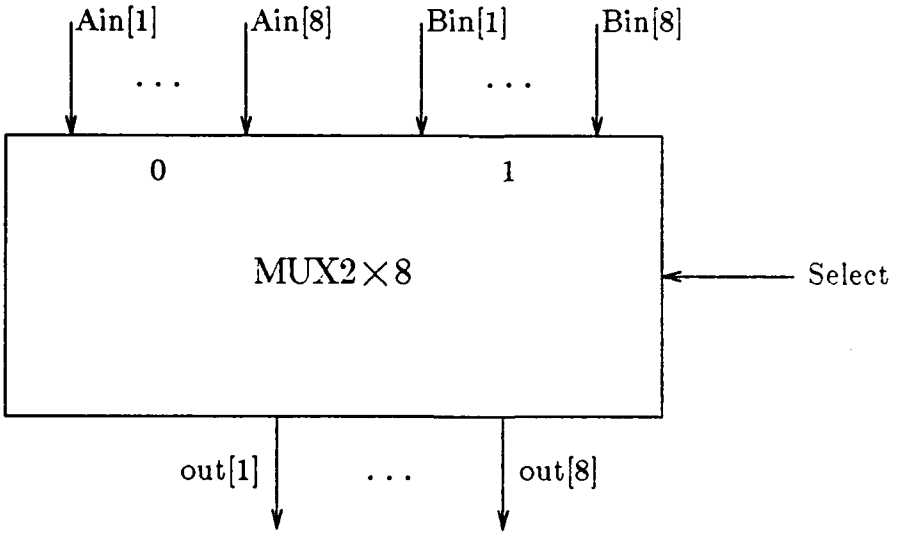


Figure 1. A Block Diagram of the MUX2×8 Module

Module 2 : MUX4×8

MUX4×8 is a multiplexor which allows the selection of one of four sets of 8 input lines. This module is shown in Figure 2 and is used in some BIT techniques as a means of arbitrating among the ambiguity group outputs ready for a signature analyzer.

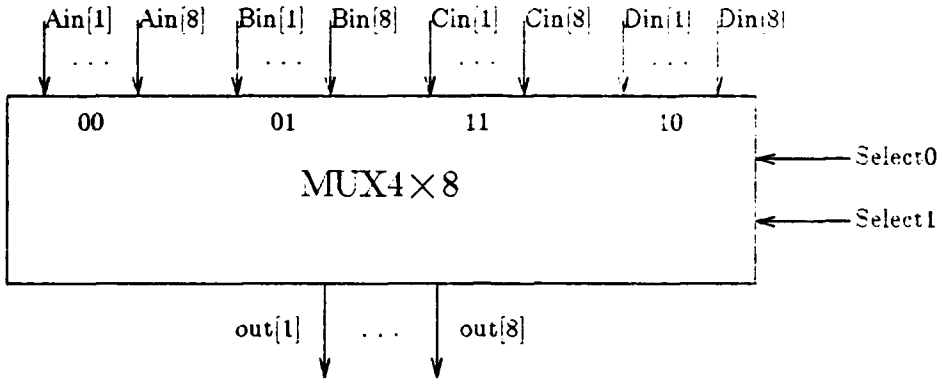


Figure 2. A Block Diagram of the MUX4×8 Module

Module 3 : MUX2

MUX2 is a multiplexor which allows the selection of one of two sets of 16 input lines. This module is shown in Figure 3 and is used in some BIT techniques as a means of selecting either "normal" input or "test" input.

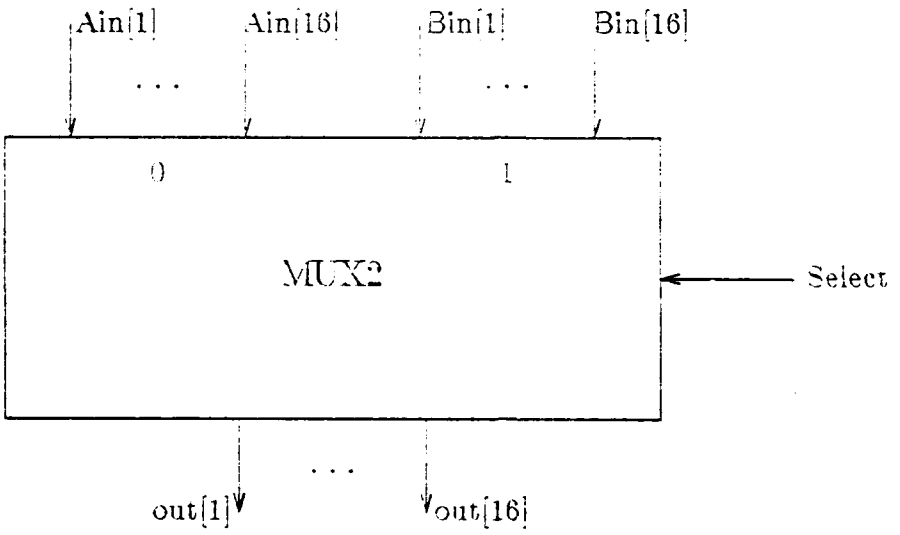


Figure 3. A Block Diagram of the MUX2 Module

Module 4: MUX4

MUX4 is a multiplexor which allows the selection of one of four sets of 16 input lines. This module is shown in Figure 4 and is used in BIT techniques to select a portion of output lines to be routed to a signature analyzer.

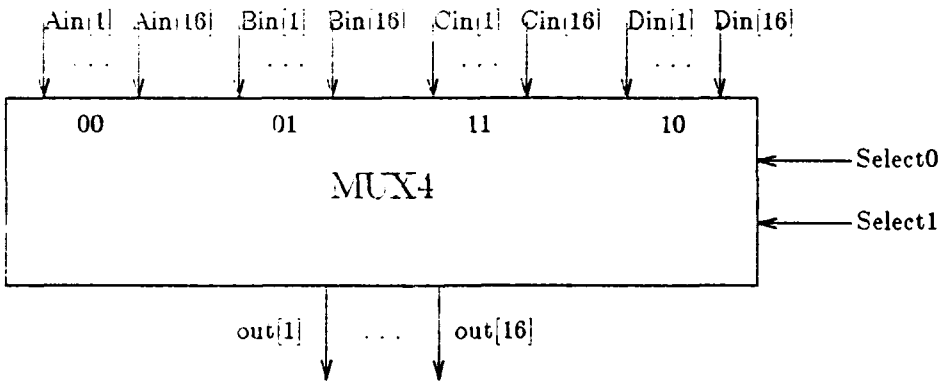


Figure 4. A Block Diagram of the MUX4 Module

Module 5 : ZEROS

ZEROS always has a low ('0') signal on its 16 output lines. There are no inputs to the module. This module is shown in Figure 5. The module is used to provide inputs to devices expecting inputs on all of its lines (e.g., 8-bit comparator).

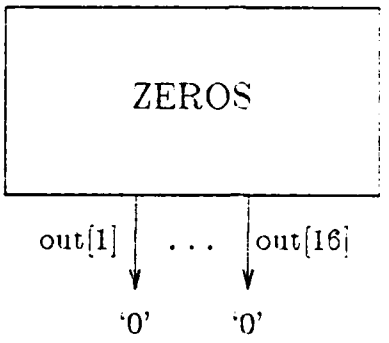


Figure 5. A Block Diagram of the ZEROS Module

Module 6 : ONES

ONES always has a high ('1') signal on its 16 output lines. There are no inputs to the module. This module is shown in Figure 6. The module is used to provide inputs to devices expecting inputs on all of its lines (e.g., 8-bit comparator).

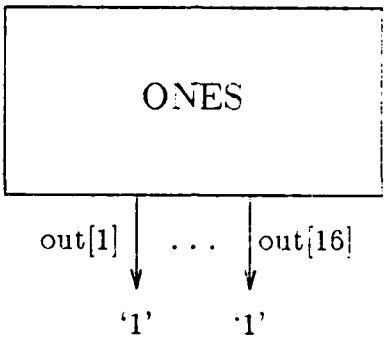


Figure 6. A Block Diagram of the ONES Module

INSTALLATION GUIDE
FOR THE
TEST ENGINEER'S ASSISTANT SYSTEM

ADAS is a registered trademark of the Research Triangle Institute.
VMS is a trademark of Digital Equipment Corporation.
VAX is a trademark of Digital Equipment Corporation.
VAXstation is a trademark of Digital Equipment Corporation.
VAXstation II/GPX is a trademark of Digital Equipment Corporation.
VAXstation II is a trademark of Digital Equipment Corporation.
VAXstation 2000 is a trademark of Digital Equipment Corporation.
MicroVAX is a trademark of Digital Equipment Corporation.
MicroVAX II is a trademark of Digital Equipment Corporation.
DEC is a trademark of Digital Equipment Corporation.
VWS is a trademark of Digital Equipment Corporation.
VAXC is a trademark of Digital Equipment Corporation.
Imagen is a trademark of Imagen Corporation.
Digi-Pad is a registered trademark of GTCO Corporation.
Bit Pad One is a registered trademark of SummaGraphics Corporation.
SummaMouse is a trademark of SummaGraphics Corporation.
M-1 Mouse is a trademark of Mouse Systems.
PostScript is a registered trademark of Adobe Systems, Inc.

TABLE OF CONTENTS

OVERVIEW

This installation guide describes the ADAS/TEA system installation. This guide is divided into several major sections, which are listed in the table of contents:

1. Setting Up Hardware — describes workstation switch settings and interconnections, mouse calibration, and plotter pen order.
2. The Installation Process — describes each step in the ADAS/TEA installation process for both new installations and system upgrades. Each installation step begins on a new page; these steps must be performed in the sequence presented to ensure the system is installed correctly.
3. Sample *Graphics.dat* and *Hardcopy.dat* Files — discusses the contents of these files and illustrates their interconnections.
4. *Graphics.dat* Device Flags — describes all flags associated with entries in the file and gives their value ranges.
5. Troubleshooting the Installation Process — provides guidelines for correcting problems during the installation process.

If you have problems installing ADAS/TEA, or questions that this guide does not address, contact Jill Hallenbeck at 919-541-7413.

NOTES

This is an installation guide for installation of the prototype TEA code on a MicroVAX II/GPX running VMS. It is our intent to minimally disturb regular ADAS installation procedures, so more information is provided than is actually needed.

Also, the version of EDIGRAF delivered with TEA is a version resulting from recent Government contracts, so some added features may be unfamiliar to the ADAS user.

THE ADAS/TEA ENVIRONMENT

The TEA user needs a VAXstation.

- Stand-alone workstations — follows with at least 10 Mbytes of disk storage capacity and at least 3 Mbytes of working memory:
    - VAXstation II/GPX, with VMS Operating system, Version 4.3 or later; VWS Version 3.0 or later; optional VAXC compiler Version 2.0 or later; and/or Ada compiler with either Version 1.3, 1.4, or 1.5.
    - VAXstation 2000, with VMS Operating system, Version 4.3 or later; VWS Version 3.0 or later; optional VAXC compiler Version 2.0 or later. The monitor can be either B/W, 4-color plane, or 8-color plane.
- Pen plotter — Hewlett-Packard Plotters, Model 7220-C, Model 7585-B, Model 7550, and Model 7475-A. All plotters must be connected to their own ports on the host computer.
- Laser printer — Imagen Imprint-10 and 8/300 and other PostScript-supported laser printers.

If the current version of VMS is not 4.7, then ADAS must be relinked. Other errors that may be encountered during the installation process are described later in this guide, in the section entitled *Troubleshooting the Installation Process.*

The VAXstation Environment

The connection of the VAX mouse to the VAXstation monitor is straightforward and is well-documented in the VAXstation manual.

By default, TEA creates a square graphics display window when an interactive TEA tool is invoked that is 13 cm by 13 cm. The TEA user can change the dimensions of this window by entering the command

```
$ define gterm_size  nn
```

where *nn* is the size of the window in centimeters (maximum dimension is 25 cm by 25 cm). The graphics window created by TEA can be repositioned like any other window.

SETTING UP THE PLOTTER

Plotter Switch Settings

1. Hewlett-Packard Model 7475-A

Set the switches on the back side of the plotter as follows:

	Switch Setting								
	S2	S1	Y	US	A3	B4	B3	B2	B1
1200 baud:	0	0	0	1	0	0	1	1	1
2400 baud:	0	0	0	1	0	1	0	0	0
4800 baud:	0	0	0	1	0	1	0	0	1
9600 baud:	0	0	0	1	0	1	0	1	0

2. Hewlett-Packard Model 7585-B

Set the interface mode switch (HP-IB/RS-232-C) on the back side of the plotter to the RS-232-C position.

Set the rotary switch at the bottom of the switch block to 2400 baud.

Set the switches to the right of the interface mode switch as follows:

1. Set EXPAND/NORMAL to NORMAL.
2. Set EMULATE/NORMAL to NORMAL.
3. Set STAND ALONE/EAVESDROP to STAND ALONE.
4. Set MONITOR MODE/NORMAL to NORMAL.
5. Set LOCAL/NORMAL to NORMAL.

Set the switches in the RS-232-C section as follows:

1. Set PARITY ON/OFF to OFF.
2. Set PARITY EVEN/ODD to ODD.
3. Set DUPLEX HALF/FULL to FULL.
4. Set HARDWIRE/MODEM to MODEM.
5. Set DTR BYPASS/NORM to NORM.

SETTING UP THE PLOTTER

Pen Color Order

For Hewlett Packard plotters that have *eight* pens (7550-A, 7220-C, 7585-B), insert the pens into the following carriage positions:

1 - red	5 - yellow
2 - blue	6 - purple
3 - green	7 - brown
4 - black	8 - orange

For Hewlett Packard plotters that have *six* pens (7475-A), insert the pens into the following carriage positions:

1 - red	4 - black
2 - blue	5 - yellow
3 - green	6 - purple

STEP ONE: READING THE ADAS/TEA INSTALLATION TAPE

A. Tape File Structure and Contents

The ADAS/TEA distribution tape is a nine-track, 1600 bpi or TK50 VAX/VMS backup format tape. The TEA installation process will create the following directory structure, where *dev:* is the device that holds the installation files:

dev:`[adas.config]`	Contains *graphics.dat* and *hardcopy.dat* configuration files. Sample files are generated during the installation; these will have to be edited using the information in this guide to customize the installation for your site.
dev:`[adas.dist.bin]`	Contains the ADAS/TEA program executable files.
dev:`[adas.dist.help]`	Contains the ADAS/TEA help files.
dev:`[adas.dist.include]`	Contains CSIMGEN include files for the CsimDb data base access routines.
dev:`[adas.dist.lib]`	Contains color map files and master graph component template files.
dev:`[adas.dist.obj]`	Contains object files for relinking the *bin* directory.

STEP ONE: READING THE ADAS INSTALLATION TAPE

B. Creating An ADAS Directory

Before performing this step, be sure to check for any installation instructions packed with the release tape, since these could affect the procedures below.

- Create a directory for the ADAS system files. The preferred name for this directory is *adas*, and its preferred location is directly in a device's master file directory. The following example creates directory *adas* in a master file directory named *dra2:*

```
$ create/dir dra2:[adas]
```

- If this is a new installation, or if the user "**adas**" does not already exist on the system, then add the user "**adas**" with a home directory of *dra2:[adas].*
- Define a systemwide logical named *adas* that will reference the ADAS root directory. For example,

```
$ define/system adas dra2:[adas.]/translation=concealed
```

This will allow you to access files by using the path names in the user manual.

STEP ONE: READING THE ADAS/TEA INSTALLATION TAPE

C. Extracting The Tape Contents

Login as "adas" and extract the contents of the tape into the new directory using the **backup** command. The following example assumes the tape drive is named *msa0:* and places all ADAS files and directories from the tape in directories *adas:[config]*, and *adas:[dist]*.

```
$ set default adas:[000000]
$ mount/foreign msa0: adas tape
$ backup/rewind/verify tape:adas.save [*...]
$ dismount tape
```

STEP TWO: CREATING FILE GRAPHICS.DAT

File *adas:[config]graphics.dat* contains information that associates input and output devices with logical workstations and assigns names to the workstations. The file is divided into two parts separated by a line containing two percent signs (**%%**). Lines beginning with a pound sign (#) are comment lines. Their contents are ignored. A sample *graphics.dat* file is supplied with the installation tape; it can be edited to customize it for your site's workstation configuration. Examine this file and the *graphics.dat* example later in this guide to become familiar with the structure and contents of the file.

The *first part* of the file describes logical workstations and their characteristics. Each line of the file describes a single graphics workstation or hardcopy device and contains several fields separated by blanks (the values of **out_type, in_type, and flagi-valuei** are listed later in this guide, in the section *Graphics.dat Device Flags and Their Values*):

work_name out_type out_name in_type in_name [flag1=value1 ...]

where:

work_name is the name of the workstation or hardcopy device as entered by a user when invoking an ADAS program; it can be any alphanumeric value, but workstation names *must* use lower case letters only and hardcopy device name must be no longer than 18 characters (e.g., `ras1`, `bwplot`).

out_type is the output device type; legal values are listed later in this guide (e.g., `rastech`, `imagen`).

out_name is the physical name of the output device described in the second half of the file; it can be any alphanumeric value, but it *must* use lower case letters only (e.g., `rast`, `hpplotlg`).

in_type is the input device type; legal values are listed later in this guide (e.g., `summa`, `vs11tab`).

in_name is the physical name of the input device described in the second half of the file; it can be any alphanumeric value, but it *must* use lower case letters only (e.g., `ras1`).

flagi=valuei is one or more device-dependent flags associated with the workstation or hardcopy device. There are flags for device type, device model, plot color and fill modes, laser printer resolution, and laser printer output disposition. These flags are described later in this guide.

The *second part* of the file describes paths to actual physical devices. Each line of the file describes a single physical device and contains four mandatory fields separated by blanks:

dev_name conn_type hard_name speed

where:

dev_name is the physical device name as specified in the first part of the file (see the descriptions of **out_name** and **in_name** above).

conn_type is the connection type; the only valid connection type at present is a capital N.

hard_name is the hardware name for the device; this will usually be the name of the terminal line to which the device is connected. Some devices may have their own controllers, however, and will not be attached to a terminal line.

speed is the baud rate for the line to which the device is connected. This field can be set to the value *ignore* if the speed does not have to be set for the device (i.e., for a device with its own controller).

STEP THREE: CREATING FILE HARDCOPY.DAT

File *adas:/config/hardcopy.dat* contains the names of all hardcopy devices described in file *graphics.dat* and is used to generate a menu of hardcopy devices in the ADAS user interface. A sample *hardcopy.dat* file is supplied with the installation tape: it can be edited to customize it for your site's workstation configuration. Examine this file and the *hardcopy.dat* example later in this guide to become familiar with the structure and contents of the file. All entries in the file must be alphanumeric, must begin with a letter, and can be up to 18 characters long. Each line of the file contains the name of a single hardcopy device.

STEP FOUR: MODIFYING THE SYSTEM START-UP PROCEDURE

Edit the system start-up file *sys$manager:systartup.com* to set the protection on the output terminal devices specified in the second part of file *graphics.dat* as follows:

```
$ set prot=w:rwlp hard_name/device
```

where *hard_name* is the hardware name for the device as specified in *graphics.dat.* For the sample *graphics.dat* file later in this guide, the entries in the system start-up file would look like the following:

```
$ set prot=w:rwlp ttb0:/device
$ set prot=w:rwlp ttb1:/device
$ set prot=w:rwlp ttb2:/device
$ set prot=w:rwlp ttb3:/device
```

Add the following line to the system start-up file to initialize the ADAS system:

```
$ @adas:[dist.bin]run_monitor
```

Edit the user's *login.com* file to execute the ADAS setup file:

```
$ @adas:[dist]setup
```

NOTE: It is not recommended that a graphics device be attached to Port 0 or Port 1 of a DMF-32 because of the complex communication requirements. Another recommendation is that the terminal line to each graphics device be set to "no broadcast."

STEP FIVE: CHECKING THE INSTALLATION

Go to the *ADAS User Manual* and try executing the *Add Command Tutorial* , page 7–8. If the following error message appears, the programs need to be relinked:

```
%DCL-W-ACTIMAGE, error activating image name
-CLI-E-IMGNAME, image file filename
-SYSTEM-F-SHRIDMISMAT, ident mismatch with shareable image
```

To relink the programs, execute the ADAS linkage command file as follows:

```
$ set def adas:[dist.obj]
$ @link
```

See the note on ADAS on page 2.

A SAMPLE GRAPHICS.DAT FILE

The following example, Figure 1, illustrates a complete *graphics.dat* file.

```
ras1 rastech ras1 rastab ras1 model=40
ras2 rastech ras2 mouse ras2tab type=standard model=10
rasT rastty rasT rasttytab rasT model=10
tek1 tek tek1 tektab tek1 model=4107 device=8
plot hpplotter plot null null type=7585 color=color
fill hpplotter plot null null type=7585 color=color fill=on
bwplot hpplotter plot null null type=7585 color=bw
imagen imagen null null null "command=print/que=laser/delete %s" resolution=240
imfile im_file null null null resolution=300
imfile2 imagen null null null "command=rename %s adas imagen_output" resolution=300
vs11 vs11 null vs11tab null color=color
vs2000 vs11 null vs11tab null color=color9
vs11bw vs11 null vs11tab null color=bw
hpfile hp_file null null null type=7220 color=color
postscript postscript null null null resolution=300
%%
ras1 N ttb0: 9600
ras2 N ttb1: 9600
ras2tab N ttb2: 1200
rasT N ttb3: 9600
plot N ttb4: 2400
tek1 N ttb5: 9600
```

Figure 1. Contents of graphics.dat

In Figure 1,

ras1 represents a Raster Technologies Model 1/40 graphics terminal that has an unspecified type of mouse or data tablet attached directly to it. The **ras1** terminal is connected to line **ttb0** and runs at 9600 baud.

ras2 represents a Raster Technologies Model 1/10 that has a Mouse Systems Model M-1 attached directly to the computer rather than into the graphics terminal. The **ras2** terminal is connected to line **ttb1** and runs at 9600 baud. The mouse is connected to line **ttb2** and runs at 1200 baud.

rasT represents a Raster Technologies Model 1/10 that will simultaneously display text (in the lower 8 rows) and graphics. This eliminates the need for an additional VT100 terminal. The **rasT** terminal is connected to line **ttb3** and runs at 9600.

vs11 represents a color VAXstation 11/GPX. Note that the logical input and output devices are null because there is only one software display and the software knows where to find it.

`vs2000` represents a color VAXstation with 4 color planes. Note the `color=color9` entry which is necessary for the 4 color plane feature.

`vsiibw` represents a black-and-white display for a VAXstation. Note the `color=bw` entry necessary for this feature.

`tek1` represents a Tektronix Model 4107 graphics terminal with a Tektronix 4957 data tablet attached directly to it. The `tek1` terminal is connected to line `ttb5` and runs at 9600 baud.

`plot` represents a Hewlett-Packard Model 7585-B pen plotter. The entry `color=color` will outline the nodes in color but will not fill them. The plotter is connected to line number `ttb4` and runs at 2400 baud.

`fill` represents a Hewlett-Packard Model 7585-B pen plotter. The entry `fill=fill` will fill the entire node in color. The plotter is connected to line number `ttb4` and runs at 2400 baud.

`bwplot` represents a Hewlett-Packard Model 7585-B pen plotter. The entry `color=bw` will outline the nodes in black but will not fill them. The plotter is connected to line number `ttb4` and runs at 2400 baud.

`imagen` represents an unspecified model Imagen laser printer with a resolution of 240 pixels per inch. The print commands are output directly to an Imagen printer.

`imfile` represents an interactive driver that allows the print commands, in impress format, to be output to a file instead of directly to the printer. The generated file, named *adas.impress* by default, must be incorporated into a larger document since it does not contain any header information. The entry `imfile` must be added to the *hardcopy.dat* file.

`imfile2` uses the Imagen driver to output the print commands, in impress format, to a file instead of an Imagen printer. The generated file, named *adas.imagen_output* by default, will contain the necessary header information for printing. The entry `imfile2` must be added to the *hardcopy.dat* file.

`hpfile` represents an interactive driver that will output the Hewlett-Packard pen plotter commands to a file instead of to a plotter. The generated file, named *adas.hpplot* by default, can be printed using your local plotter print commands. The entry `hpfile` must be added to the *hardcopy.dat* file.

`postscript` represents an interactive driver that will output print commands, in PostScript format, to a file named *adas.ps* by default. The resolution will differ depending on machine type. This file can be printed using your local PostScript print commands. The entry `postscript` must be added to the *hardcopy.dat* file.

HOW GRAPHICS.DAT INTERNAL POINTERS WORK

When an ADAS user invokes one of the interactive tools (see Figure 2. below), the graphics device name specified on the command line must match one of the graphics device names in file *graphics.dat.* Note that in Figure 2 the user has specified device `ras1`, which matches one of the workstation definition lines in the first part of file *graphics.dat.* The output and input device physical names (`rast`) in the first part of file *graphics.dat* connect the workstation description to the physical device in the second part of the file. `ras1` is a Raster Technologies Model 1/40 device connected to terminal line `ttb0:` running at 9600 baud.

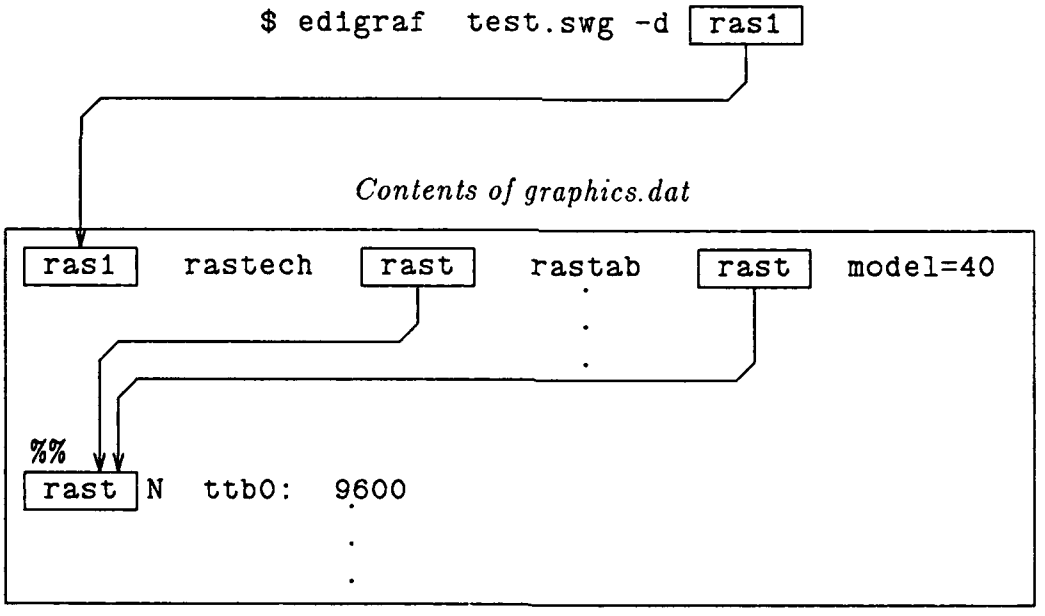


Figure 2. Internal graphics.dat Pointers

A SAMPLE HARDCOPY.DAT FILE

The following example, Figure 3, illustrates a complete *hardcopy.dat* file for the sample *graphics.dat* file. The three hardcopy devices are associated with hardcopy device names in the *graphics.dat* file entries; they are entered in their order of appearance in ADAS user interface menus. Changing their order in *hardcopy.dat* will change their order of appearance in ADAS menus.

```
plot
fill
bwplot
imagen
imfile
imfile2
hpfile
postscript
```

Figure 3. Contents of hardcopy.dat

HOW HARDCOPY.DAT INTERNAL POINTERS WORK

When an ADAS user selects the `hardcopy` option from the user interface main menu, a submenu of hardcopy devices is displayed on the terminal screen (see Figure 4. below). Device names are displayed in their order of occurrence in file *hardcopy.dat*. Note that in Figure 4 the user has selected device `bwplot`, which is the third entry in file *hardcopy.dat*. The *hardcopy.dat* entry matches one of the workstation/device definition lines in file *graphics.dat*. The output device physical name (`plot`) in the first part of file *graphics.dat* connects the hardcopy device description to the physical device in the second part of the file. `bwplot` is a Hewlett-Packard Model 7585 plotter operating in black/white mode; it is connected to terminal line `ttb3:` running at 2400 baud.

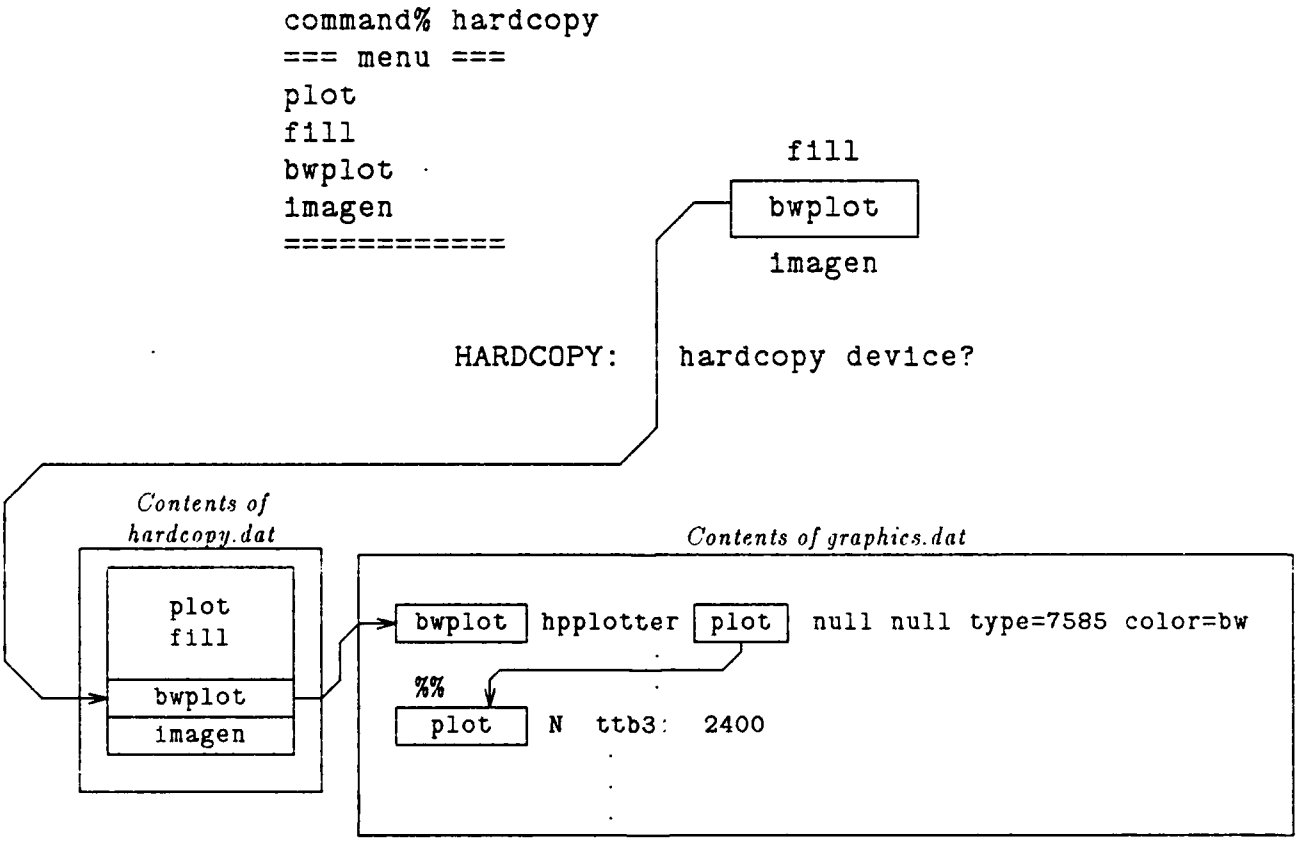


Figure 4. hardcopy.dat and Its Interaction with graphics.dat

GRAPHICS.DAT DEVICE FLAGS AND THEIR VALUES

1. Output Device Flags

Second field. All characters must be lower case. The following output device types are supported:

`hpplotter` Hewlett-Packard pen plotter. Use the `type` flag to select model, the `color` flag to select color mode, and the `fill` flag to select plot fill mode.

`imagen` Imagen laser printer. Third field (physical output device name) must be set to `null`. Use the `resolution` flag to set output resolution and the "`command=`*value*" flag to specify output disposition.

`rastech` Raster Technologies graphics device. Use the `model` flag to select model.

`tek` Tektronix terminals. Use the `model` flag to select model and the `device` flag to select input port number.

`vsii` VAXstation. The third field (physical output device name) must be set to `null`. Use the `color` flag to select either B/W, 16 colors, or 256 colors.

`im_file` Imagen print to file. Use the `resolution` flag to set output resolution.

`hp_file` Hewlett-Packard pen plotter commands to file. Use the `color` flag to select color mode and `fill` flag to select plot fill mode.

`postscript` Generates PostScript format file for printing on PostScript-supported printers. Use the `resolution` flag to set output resolution.

2. Input Device Flags

Fourth field. All characters must be lower case. The following input device types are supported:

`mouse` A Mouse Systems Model M-1 mouse or Summagraphics SummaMouse plugged directly into the computer. Use the `type` flag to identify the type of mouse.

`rastab` Any type of data tablet or mouse that is plugged directly into a Raster Technologies Model 1/10, 1/40, or 1/60 graphics device. No special flags are required.

`summa` A Summagraphics Bit Pad One. No special flags are required.

`null` Indicates there is no input device associated with the entry.

`tektab` Any type of data tablet or mouse that is plugged directly into a Tektronix terminal. No special flags are required.

`vsiitab` VAXstation mouse or tablet. The fifth field (physical input device name) must be set to `null`. No special flags are required.

3. Workstation Flags

Sixth and following fields. All characters must be lower case. One or more of the following flags may be specified as described above:

`color=`*value* When used as a flag for the Hewlett-Packard plotters, it specifies black-and-white plots (*value* is `bw`) or color plots (*value* is `color`). Default is `color`.

When used as a flag for the Hewlett-Packard plotters, it specifies black-and-white displays (*value* is `bw`), or for displays that support only 16 colors (*value* is `color9`), or for displays that support 256 colors (*value* is `color`).

`"command=`*value*`"` Specifies disposition of the graphics driver output file for an Imagen laser printer. The entire flag must be enclosed in double quotation marks as shown since the *value* field may contain blanks. Typically, the *value* field is a print command giving the disposition of the output; the file name is specified with a `%s` which the ADAS programs replace with the actual file name at execution time. For example:

```
"command=print/que=laser2/delete %s"
```

`device=`*value* Specifies an input device port number for a Tektronix workstation. *Value* is set to the integer number of the input port (default is `0`).

`fill=`*value* Specifies nonfilled nodes on plots (*value* is `off`) or filled nodes on plots (*value* is `on`). The default value is `off`. **NOTE:** If this option is provided at your site, it should be used with caution, since node filling wears out plotter pens quickly and greatly increases the time it takes to generate a plot.

`model=`*value* Specifies a graphics device model. *Value* may be set to
`10` for a Raster Technologies Model 1/10 (default for a Raster Technologies device)
`40` for a Raster Technologies Model 1/40 or 1/60
`4107` for a Tektronix Model 4107 (default for a Tektronix device) or 4207
`4113` for a Tektronix Model 4113 or 4115
`4125` for a Tektronix Model 4125

`resolution=`*value* Specifies pixels per inch resolution for an Imagen laser printer. Default is 240 pixels/inch.

`type=`*value* Specifies a device type. *Value* may be set to
7220 for a Hewlett-Packard Model 7220-C plotter
7475 for a Hewlett-Packard Model 7475 plotter
7550 for a Hewlett-Packard Model 7550 plotter
7585 for a Hewlett-Packard Model 7585-B plotter (default for a Hewlett-Packard plotter)
standard for a Mouse Systems Model M-1 mouse (default for a mouse)
summamouse for a Summagraphics SummaMouse

TROUBLESHOOTING THE INSTALLATION PROCESS

1. ADAS Cannot Initialize Graphics Device

Check file *graphics.dat* to make sure there is a device by that name. Make sure the line name is correct, the input and output types are valid, and that device flags are set correctly. Make sure the device name to port associations and baud rates are correct in the second half of the file.

If there are no identifiable problems in *graphics.dat*, check the terminal line and make sure it has read/write protection. If the graphics device is a terminal emulator (for example, the Raster Technologies Model 1/10 can emulate a VT-100 terminal), make sure it is cabled correctly by trying to log in using the graphics device as a terminal.

2. ADAS Monitor Generates An Error

If the error message reads

```
ADAS monitor not running. See your system manager.
```

activate the monitor as [system] by entering the command

```
$ @adas:[dist.bin]run_monitor
```

If the error message reads

```
ADAS monitor error. See your system manager.
```

you must stop the monitor that is running, then reactivate the monitor by entering the command

```
$ @adas:[dist.bin]run_monitor
```

3. Licensed Simultaneous ADAS Use Exceeded

If this message appears, and the number of active workstations does not exceed the number specified in your license, you must stop the monitor that is running, then reactivate the monitor by entering the command

```
$ @adas:[dist.bin]run_monitor
```

4. Mouse Will Not Track Or Tracks Incorrectly

Try powering down the mouse and then check the connections. Power the mouse back up again and recalibrate. If you are using a Tektronix device, make sure the `device` flag is set correctly in file *graphics.dat*.

5. Other Communication-Related Problems

Check device speeds and switch settings: check the cabling between the mouse or data tablet and the graphics device.