cadence®

Encounter® Test: Guide 2: Testmodes

Product Version 12.1.101
February 2013

Printed in the United States of America.

Cadence Design Systems, Inc. (Cadence), 2655 Seely Ave., San Jose, CA 95134, USA.

Contents

List of Figures 9

Preface 13
About Encounter Test and Diagnostics 13
Typographic and Syntax Conventions 13
Encounter Test Documentation Roadmap 14
Getting Help for Encounter Test and Diagnostics 15
Contacting Customer Service 15
Encounter Test And Diagnostics Licenses 16
What We Changed for This Edition 16
Revisions for Version 12.1.100 16

1
Test Mode Overview 17

2
Build Test Mode 19
Building a Test Mode 19
Build Test Mode Command Examples 20
Build Test Mode Inputs 21
Mode Definition File (Required) 21
Test Function Pin Assignments (Usually Required) 22
Assign File 22
Test Function Pins in Design Source 23
Tester Description Rule (Required) 24
Sequence Definition File (Optional) 25
STIL Pattern File (Optional) 26
Partition File (Optional) 26
BSDL File (for 1149.1 Testmodes) 27
Build Test Mode Outputs 27

Active and Inactive Logic . . . 27
Design States . . . 27
Build Test Mode Restrictions . . . 28
Identifying Test Function Pins (Advanced) . . . 28
Specifying Test Function Pin Attributes in Design Source . . . 28
Assignfile Examples . . . 32
Test Function Attribute Definition . . . 32
Determining Which Test Functions to Use . . . 52
Valid Combinations of Test Functions . . . 54
Mode Initialization Sequences (Advanced) . . . 66
Requirements of Initialization Sequences . . . 66
Automatically Generated Initialization Sequence . . . 67
Coding an Externally Specified Initialization Sequence . . . 68
Custom Scan Sequences (Advanced) . . . 70
Defining a Custom Scan Sequence . . . 71
Automatically Generated Initialization Sequence . . . 84

3
Analyze Test Mode Results . . . 87
Analyzing Build Testmode Log Results . . . 87
Mode Init and Scan Sequences . . . 87
Active Logic . . . 87
Faults Not Active in any Testmode . . . 89
Troubleshooting Build Test Mode Problems . . . 90
Unit or Zero Delay Simulations for Test Mode . . . 92
Reporting Test Mode Information . . . 93
Basic Command Syntax for report_test_structures . . . 93
Scan Chain Information Report . . . 94
Embedded Pipeline Report . . . 96

4
Multiple Test Modes . . . 99
Building Multiple Test Modes . . . 99
Cross-Mode Markoff . . . 100
Building a Hierarchy of Test Modes (Parent Testmodes) . . . 101

5
Delete Test Mode Information 103

Deleting a Test Mode 103
Deleting Committed Data from a Test Mode 103

A
Design Structures and Testmode Details 105

Delay Test Modes 105
- Customizing a Mode Definition for Delay Test 105

Assumed Scan Test Modes 106
- Excluding Specific Flops from a Scan Chain 107
- Assumed Scan Mode Limitations 108

Pipelined Scan Test Modes 109
OPMISR Test Modes 111
- Introduction to OPMISR and OPMISR+ Compression 111
- Generating OPMISR TestMode 116
- OPMISR Building Blocks 118
- Modes of Operation 120
- Channel Masking Logic 124
- Implementing Channel Masking Logic in a Design 127
- Channel Masking on Pad Cycles 128
- Building a Test Mode for OPMISR, OPMISR+ 135
- Specifying Attributes for a MISR Test Mode 140
- Inserting Block-Level Test Compression 141
- Commands for Block Level Test Compression 144

1149.1 Test Mode 145
- IEEE 1149.1 Boundary Scan Features 146
- Boundary Scan Design Requirements 150
- 1149.1 Test Mode Input Files 151

1149.6 Test Mode 152
- The 1149.6 Test Receiver 152
- Changes to the 1149.6 TAP Controller 155
- Changes to the Output Boundary Scan Cell 159
- IEEE 11496 Boundary Cells 160

Verifying the Boundary Scan Implementation . . . 172
On-Product XOR Compression . . . 173
The XOR Compression Macro . . . 175
Modes of Operation . . . 177
XOR Compression Design Flow . . . 180
XOR Compression Limitations . . . 181
SmartScan Compression . . . 183
Compression Serial and Parallel Interfaces . . . 184
Compression with Serial Only Interface . . . 192

B
Mode Definition File Syntax . . . 195
Mode Definition File . . . 195
Mode Definition Syntax . . . 195
Mode Definition Statements . . . 196

C
Tester Description Rule (TDR) File Syntax . . . 259
Introduction to Tester Description Rules (TDRs) . . . 259
TDR Statement Syntax . . . 259
TDR_DEFINITION . . . 260
TEST_PINS . . . 262
OSCILLATOR_PULSES_PER_TESTER_CYCLE . . . 265
TERMINATION . . . 266
MEASURE . . . 267
PIN_TIMING . . . 269
DELAY_PROCESSING . . . 281
SCAN_REQUIREMENTS . . . 284
Sample TDRs . . . 286

D
Identifying Inactive Logic . . . 289
Reporting Inactive Logic . . . 290
Debugging Inactive Logic . . . 290

E
Test Mode Design States 295
Design States 295
Tie Only 295
Test Inhibit 295
Test Constraint 295
Test Constraint and Clocks Off 296
Mode Initialization 296
Scan States 296
Flush Scan Chain 298
Nonscanflush Design State 298
MISR Observe 298
MISR Reset 299
PRPG Load 299
Channel Mask Load 299
OPCG Load State 299

F
Boundary Scan Design Language 301
BSDL Extension - Port Alias 301
BSDL Extension for Identifying Image Unwired Ports 303
Parsing BSDL 304
Writing BSDL 307
Creating Initial BSDL File for Custom 1149.1 Design 310

Index 313

List of Figures

Figure 2-1 Waveforms for Test Function Pins 24
Figure 2-2 An Example Test Mode Initialization Sequence in TBDpatt Format 69
Figure 2-3 Design that Requires Multiple Scan Sections 73
Figure 2-4 Basic Scanop Structure 76
Figure 2-5 Advanced Scanop Structure 77
Figure 2-6 TAP Controller State Diagram 78
Figure 2-7 Example scanprecond sequence for TAP scan SPTG, TAP_TG_STATE=TLR 81
Figure 2-8 Example scansequence and scanlastbit sequences, TAP_TG_STATE=TLR . . 82
Figure 2-9 Example scansectionexit and scanexit sequences, TAP_TG_STATE=TLR . . . 82
Figure 2-10 Example misrobserve, misrreset and channelmaskload Sequence 82
Figure A-1 Normal FullScan ATPG Configuration 112
Figure A-2 OPMISR Configuration 113
Figure A-3 OPMISR+ Configuration 114
Figure A-4 OPMISR using Scan Fan Out 115
Figure A-5 On-Product MISR Configuration Example 116
Figure A-6 OPMISR in Functional Mode 121
Figure A-7 OPMISR Scan Mode 122
Figure A-8 OPMISR Read Operation 123
Figure A-9 Normal ATPG Scan Test using OPMISR 124
Figure A-10 WIDE2 Channel Masking Logic for One Channel 125
Figure A-11 WIDE1 Channel Masking with a Single CME Signal 126
Figure A-12 WIDE0 Channel Masking without Mask Bits 126
Figure A-13 Mask Bits Loaded from the Scan-in Pin 127
Figure A-14 Variable Length Overlapped Scans 129
Figure A-15 Exposure from Overlapped Scans without Masking 130
Figure A-16 Scan Overlap 131
Figure A-17 X-Mask Padding Problem 132
Figure A-18 X-Mask 0 Padding Solution 133

Figure A-19 X-Mask Channel Masking on Pad Cycles 134
Figure A-20 Test Mode Assign File for Full-Scan ATPG 137
Figure A-21 Test Mode Assign File for OPMISR Mode 138
Figure A-22 Test Mode Assign File for OPMISR+ Mode 139
Figure A-23 OPMISR Waveform Example 140
Figure A-24 Top-Level OPMISR+ 143
Figure A-25 One OPMISR+ Embedded Within a core and Another Within the UDL 144
Figure A-26 Scan Chain with Multiple Boundary Scan Devices 146
Figure A-27 IEEE 1149.1 Boundary Scan Architecture 148
Figure A-28 Typical Boundary Scan Structure 150
Figure A-29 Test Receiver Configured with Input Boundary Cells for Observability and Controllability 154
Figure A-30 Changes to Output Boundary Scan Cells 155
Figure A-31 Generation of an AC Test Signal 156
Figure A-32 Generation of an Initialization Clock 157
Figure A-33 JTAG_ACPSCLK, JTAG_ACPSEN, and JTAG_ACPULSE Implementation Details 158
Figure A-34 Augmented Output Boundary Cell with 1149.6 Support 160
Figure A-35 BC_11496_ACTR 163
Figure A-36 BC_11496_BIDIR 164
Figure A-37 BC_11496_BIDIR_TI 165
Figure A-38 BC_11496_BIDIR_TO 166
Figure A-39 BC_11496_BIDIR_TO_OO 167
Figure A-40 BC_11496_OUT 168
Figure A-41 BC_11496_OUT_NT 169
Figure A-42 BC_11496_OUT_TI 170
Figure A-43 BC_11496_OUT_TO 171
Figure A-44 BC_11496_OUT_TO_OO 172
Figure A-45 On-Product Test Data Compression Architecture 173
Figure A-46 Encounter Test Compression Options 174
Figure A-47 Internal View of XOR-Compression Macro 177

Figure A-48 XOR Compression Macro Connection to I/O Pins and Scan Channels of Design 178
Figure A-49 Compression Mode with Both Spreader and Compactor Active 179
Figure A-50 Compression Mode with Scan Fanout and Compactor Active 179
Figure A-51 XOR Compression in Full Scan Mode 180
Figure A-52 Design Flow for XOR Compression 181
Figure A-53 SmartScan Compression Architecture 183
Figure A-54 Design Flow for SmartScan Compression with Parallel and Serial Interface 185
Figure A-55 Update to ATPG Pattern by convert_vectors_to_smartscan 188
Figure A-56 Design Flow for SmartScan Compression with Serial Only Interface 193
Figure C-1 TDR_DEFINITION Statement Syntax 261
Figure C-2 TEST_PINS Statement Syntax 262
Figure C-3 OSCILLATOR_PULSES_PER_TESTER_CYCLE Statement Syntax 265
Figure C-4 TERMINATION Statement Syntax 266
Figure C-5 MEASURE Statement Syntax 268
Figure C-6 PIN_TIMING Statement Syntax 270
Figure C-7 DELAY_PROCESSING Statement Syntax 282
Figure C-8 SCAN_REQUIREMENTS Statement Syntax 285

Preface

About Encounter Test and Diagnostics

Encounter® Test uses breakthrough timing-aware and power -aware technologies to enable customers to manufacture higher-quality power-efficient silicon, faster and at lower cost. Encounter Diagnostics identifies critical yield-limiting issues and locates their root causes to speed yield ramp.

Encounter Test is integrated with Encounter RTL Compiler global synthesis and inserts a complete test infrastructure to assure high testability while reducing the cost-of-test with on-chip test data compression.

Encounter Test also supports manufacturing test of low-power devices by using power intent information to automatically create distinct test modes for power domains and shut-off requirements. It also inserts design-for-test (DFT) structures to enable control of power shut-off during test. The power-aware ATPG engine targets low-power structures, such as level shifters and isolation cells, and generates low-power scan vectors that significantly reduce power consumption during test. Cumulatively, these capabilities minimize power consumption during test while still delivering the high quality of test for low-power devices.

Encounter Test uses XOR-based compression architecture to allow a mixed-vendor flow, giving flexibility and options to control test costs. It works with all popular design libraries and automatic test equipment (ATE).

Typographic and Syntax Conventions

The Encounter Test library set uses the following typographic and syntax conventions.

- Text that you type, such as commands, filenames, and dialog values, appears in Courier type.

  Example: Type `build_model -h` to display help for the command.

- Variables appear in Courier italic type.

  Example: Use `TB_SPACE_SCRIPT=`*`input_filename`* to specify the name of the script that determines where Encounter Test results files are stored.

- Optional arguments are enclosed in brackets.

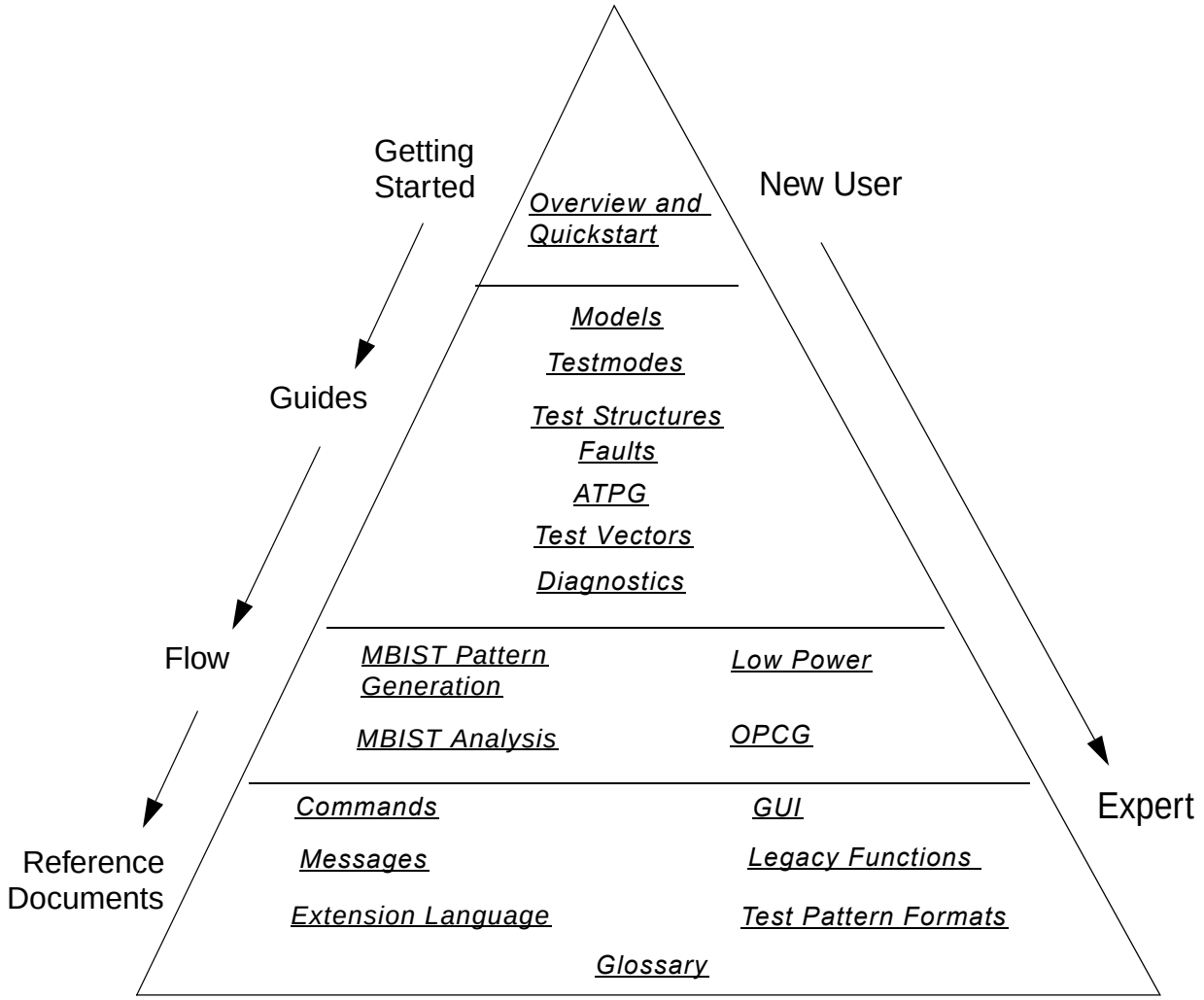
Example: `[simulation=gp|hsscan]`

- User interface elements, such as field names, button names, menus, menu commands, and items in clickable list boxes, appear in Helvetica italic type.

  Example: Select *File - Delete - Model* and fill in the information about the model.

Encounter Test Documentation Roadmap

The following figure depicts a recommended flow for traversing the documentation structure.



Getting Help for Encounter Test and Diagnostics

Use the following methods to obtain help information:

1. From the <*installation_dir*>`/tools/bin` directory, type `cdnshelp` at the command prompt.
2. To view a book, double-click the desired product book collection and double-click the desired book title in the lower pane to open the book.

Click the *Help* or *?* buttons on Encounter Test forms to navigate to help for the form and its related topics.

Refer to the following in the *Encounter Test: Reference: GUI* for additional details:

- "Help Pull-down" describes the *Help* selections for the Encounter Test main window.
- "View Schematic Help Pull-down" describes the Help selections for the Encounter Test View Schematic window.

Contacting Customer Service

Use the following methods to get help for your Cadence product.

- Cadence Online Customer Support

  Cadence online customer support offers answers to your most common technical questions. It lets you search more than 40,000 FAQs, notifications, software updates, and technical solutions documents that give step-by-step instructions on how to solve known problems. It also gives you product-specific e-mail notifications, software updates, service request tracking, up-to-date release information, full site search capabilities, software update ordering, and much more. Go to http://www.cadence.com/support/pages/default.aspx for more information on Cadence Online Customer Support.

- Cadence Customer Response Center (CRC)

  A qualified Applications Engineer is ready to answer all your technical questions on the use of this product through the Cadence Customer Response Center (CRC). Contact the CRC through Cadence Online Support. Go to http://support.cadence.com and click Contact Customer Support link to view contact information for your region.

- IBM Field Design Center Customers

  Contact IBM EDA Customer Services at 1-802-769-6753, FAX 1-802-769-7226. From outside the United States call 001-1-802-769-6753, FAX 001-1-802-769-7226. The e-mail address is edahelp@us.ibm.com.

Encounter Test And Diagnostics Licenses

Refer to "Encounter Test and Diagnostics Product License Configuration" in *What's New for Encounter Test and Diagnostics* for details on product license structure and requirements.

What We Changed for This Edition

■ "Revisions for Version 12.1.100"

Revisions for Version 12.1.100

This is new guide for this release.

1

Test Mode Overview

A test mode is a specific test configuration of the design. The configuration defines how the test structures (such as parallel scan or IEEE 1149.1) are accessed and how clocking is controlled. Each configuration is defined by the test function pins, clocking, and the current test methodology. When you build a test mode, you can specify the following information:

- The type of processing (test methodology) to be performed. This can include:
  - Special initialization sequences
  - Custom scan operations
  - Unique test access blocks
- Tester limitations
- Test function and Clock pins
- On-Product Clock (OPC) circuitry (PLLs, Clock Domains, cut points, and pseudo primary inputs)

The following information is identified by build_testmode:

- The scan structures (such as scan chains and compression structures)
- The scan sequence
- The test mode initialization sequence
- Inactive logic (cannot be observed in the test mode)
- Active logic (observable and testable via ATPG)
- OPC logic (can be observed only through a cutpoint)

In a typical ATPG flow, build_testmode occurs after the logic model has been built. Refer to Static ATPG Use Model in the *Getting Started Guide* for more information.

2

Build Test Mode

Building a Test Mode

The following is the basic command syntax for building a test mode:

```
build_testmode workdir=<directory> testmode=<modename>  \
[assignfile=<file>] \
[modedef=<modename>] [modedefpath=<directory>] \
[seqdef=<file>] [tdrpath=<directory>]
```

where:

- `workdir` is the name of the working directory.
- `testmode` is the name for the testmode being examined.
- `assignfile` is the name of the file that contains test function pin assignment statements. If an assignfile is not used, test functions must be specified as assign statements in the mode definition file or as attributes in the design netlist.

  Test function information is a required input. When the Cadence RTL Compiler (synthesis) is used to insert the test structures, this information is created automatically.
- `modedef` is the name of the file that contains the test mode definition. If the testmode name is the same as the mode definition name, mode definition is not required.
  - By default, Encounter Test searches for a mode definition file that has the same name as the test mode.
  - Typically, the predefined mode definition files cover most applications. However, you may define your own files, if required.
  - Sample mode definition files are located in the `defaults/rules/modedef` directory within the installation.
- `modedefpath` is the directory path where the modedef file is located. Multiple directories should be separated by a colon (:). This keyword is required only if you are using a customized modedef.

- `seqdef` is the file containing a custom mode initialization sequence or a custom scan sequence. In many cases, a seqdef is not required.
  - A nonscanflush custom sequence may NOT contain the loop construct:

    ```
    (pattern_type = begin_loop);
    ```

  - A `seqpath` might also be required to specify the directory of the sequence definition file.
- `tdrpath` is the directory path where the TDR file is located. This keyword is required only if you are using a custom TDR.
  - Encounter Test supplies a set of standard TDR files that are sufficient for most designs.
  - TDR is referenced within the mode definition file. The default set of mode definition files refer to the standard TDR file set. If using a default mode definition file, you will not require this parameter.
  - Sample tester description rules are located in the `defaults/rules/tdr` directory within the installation.

Build Test Mode Command Examples

Example 1: Defining a standard FULLSCAN test mode

```
build_testmode workdir=/my/work/dir testmode=FULLSCAN   \
     assignfile=/my/AssignFile
```

When using the FULLSCAN mode definition file, Encounter Test automatically searches for the file in `defaults/rules/modedef`. The `modedefpath` parameter need not be specified. The FULLSCAN mode definition file references a standard TDR, therefore, there is no need to specify the `tdrpath` keyword.

Example 2: Defining a standard FULLSCAN test mode with a different testmode name

```
build_testmode workdir=/my/work/dir testmode=testmode2   \
     modedef=FULLSCAN assignfile=/my/AssignFile
```

The resultant testmode will be called `testmode2`.

Example 3: Defining a custom test mode with mode initialization sequence file

```
build_testmode workdir=/my/work/dir testmode=dtest modedef=dtest_mode \
      modedefpath=/my/modedef assignfile=/my/AssignFile \
      seqdef=/my/sequences/seqdef
```

In this example, the mode definition file (`dtest_mode`) has been customized. The test mode name is different from the mode definition name, therefore, both `modedef` and `testmode` must be specified. The mode initialization sequence file is specified with the `seqdef` keyword. The custom mode definition references a standard TDR file, therefore, the `tdrpath` parameter need not be specified.

Build Test Mode Inputs

The following input files are used to build a test mode:

Mode Definition File (Required)

The mode definition file (modedef) contains statements that characterize the test mode, including:

- The type of tests expected to be generated (such as static, dynamic, or signature based)
- The type of scan implemented in the test mode (such as general scan design, lssd, or boundary scan)
- OPCG test structures expected in the test mode (such as PLL registers, clock domains, cutpoints, or PPIs)
- Tester Description Rule (TDR) for the test mode (describes tester capabilities and limitations)
- Test functions pin assignments for the Primary Inputs and Primary Outputs

Encounter Test includes a number of predefined Mode Definition Files in `$Install_Dir/defaults/rules/modedef`. These include `FULLSCAN`, `COMPRESSION`, `OPMISR`, `OPMISRPLUS`, and `1149`.

Some of the common mode definitions are:

`Test Mode`	Purpose
`FULLSCAN`	Static or Delay Test/Diagnostics using no compression
`COMPRESSION`	Static or Delay Test/Diagnostics using XOR Compression
`OPMISR`	Static or Delay Test/Diagnostic using OPMISR compression
`OPMISRPLUS`	Static or Delay Test/Diagnostics using OPMISR+ compression
`1149`	JTAG testmode for 1149.1 verification

Test Mode	Purpose
1149.mbist	MBIST related testmode

For details on how to customize a mode definition file, refer to "Mode Definition File Syntax" on page 195.

Refer to "Design Structures and Testmode Details" on page 105 for details on specialized testmodes including On-product clocking concepts, Assumed scan, 1149, Reduced Power, and OPMISR concepts.

Test Function Pin Assignments (Usually Required)

The test function of Primary Input/Output pins and optionally flops/latches are almost always required for testing. These are the clocks, scan pins, tied pins, etc. that control the test function of the design. There are several options available for specifying this information which are described in the following sections. If you use more than once source, and the same pin is specified differently, the function in the mode definition file wins over the function in the design source, and the function in the assign file wins over the mode definition file. It is also possible to identify pins in the design source (netlist) so that an assign file is not required. This capability is described in "Identifying Test Function Pins (Advanced)" on page 28.

Encounter Test automatically creates the preceding test sequence without requiring any input. The automatic support is also available for 1149.1 and test compression. In some cases, you will need to define your own mode initialization and clocking sequences. Refer to the section "Sequence Definition File (Optional)" on page 25.

Assign File

The assign file (optional but frequently used) specifies test function pin assignments for the `build_testmode` command. This information can also be placed into the mode definition file, but this is not recommended. If there is pin assignment information within your mode definition file, a pin assignment file can be used to augment or override previously defined test function pin assignments and other design-specific data in the mode definition file. When using one of the predefined mode definition files included with Encounter Test, an assign file is the most likely place to add the test function pin assignments.

The syntax of the assign statement is:

```
assign pin "<pinname>" test_function tf[, tf?] ;
```

where:

- *pinname* is the name of the pin being assigned a test function
- *tf* is the test function of the pin. Refer to "Test Function Pin Attributes" on page 23 for more information.
- In many cases, the quotes around the pin name are not required, but it is recommended to always quote the pin names.
- One assign statement is needed for each PI or PO that has a test function.
- A pin may have multiple test functions, but there are restrictions on how the test functions may be combined. For more information on combining test functions, see "Identifying Test Function Pins (Advanced)" on page 28.

Test Function Pins in Design Source

The following table lists the common test function pin attributes:

Table 2-1 Test Function Pin Attributes
:

Test Function	Value	Description
SC	0, 1, -, +	System clock (value is off state of clock)
EC	0, 1, -, +	Edge Triggered Scan Clock (value is off state of clock)
ES	0, 1, -, +	System Clock and Scan Clock (value is off state of clock)
SI		Scan Input
SO		Scan Output
SE	0, 1, -, + or Z	Scan Enable (constant during scanning)
TI	0, 1, -, +, X or Z	Test Inhibit (constant during scanning and testing)

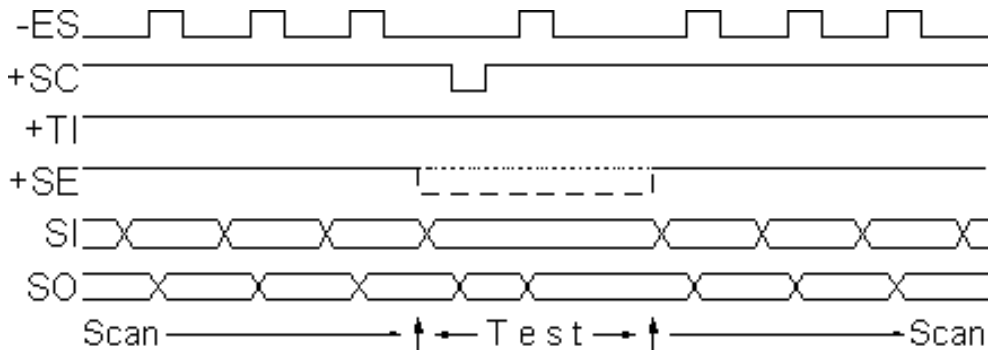
Test Function	Value	Description
TC	0, 1, -, +, X or Z	Test Constraint (constant only during testing)

Example:

```
assign pin "scanin1"   test function SI;  // scanin for chain 1
assign pin "scanout1"  test function SO;  // scanout for chain 1
assign pin "scanin2"   test function SI;  // scanin for chain 2
assign pin "scanout2"  test function SO;  // scanout for chain 2
assign pin "clock"     test function 0ES; // scan and system clock
assign pin "reset"     test function -SC; // asynchronous reset (clock)
assign pin "test_en"   test function 1TI; // test enable (constant)
assign pin "scan_en"   test function +SE; // scan enable (constant during scan)
```

The following figure illustrates the behavior of various test function pins:

Figure 2-1 Waveforms for Test Function Pins



Refer to "Assignfile Examples" on page 32 for more information on commonly-used assign file configurations.

Tester Description Rule (Required)

TDR is used to identify capabilities of the manufacturing tester. The Test Mode definition file contains the TDR name.

TDR can also identify the tester features such as number of pins, size of scan buffer, tester termination of outputs, and bidirectional pins. These characteristics are observed by

Encounter Test when building the test patterns from test generation. This tester-specific knowledge allows Encounter Test to both effectively utilize the capabilities of the test equipment and to supply some of the necessary tester setup information.

The TDR name is specified in the mode definition file. If using a standard mode definition file, check the specified TDR to ensure that it matches your tester capabilities. The standard TDRs are found in `$Install_Dir/defaults/rules/tdr` and are generally applicable to almost all testers, but may not take full advantage of any one tester's capabilities.

For details on how to customize a TDR, refer to "Introduction to Tester Description Rules (TDRs)" on page 259.

Sequence Definition File (Optional)

In some cases, Encounter Test is unable to automatically determine the proper test mode initialization sequence or scan sequence for a design. When this occurs, a mode initialization and/or scan sequence must be provided in the sequence definition file (seqdef).

Possible reasons for creating custom sequences are.

- Create a custom mode initialization sequence if:
  - RAMs must be initialized
  - Fixed value latches must be initialized
  - OPC Logic (PPIs) have TI or TC test function attributes

  For details on creating a custom mode initialization sequence see: "Coding an Externally Specified Initialization Sequence" on page 68.
- Create a custom scan sequence if:
  - No single PI stim vector enables all scan paths
  - The scan shift sequence is something other than pulsing in order all (AC, EC, BC) clocks
  - OPC Logic (PPIs) require values for scanning

For details on creating a custom scan sequence see: "Defining a Custom Scan Sequence" on page 71.

Refer to "Mode Initialization Sequences (Advanced)" on page 66 and "Custom Scan Sequences (Advanced)" on page 70 for more information.

STIL Pattern File (Optional)

Build Test Mode accepts the Standard Test Interface Language (STIL) pattern file, from which test mode information can be derived, as input. STIL files can be used as input to Build Test Mode, automatically creating scan protocol sequences and test function pin assignments, to produce a test mode based on the STIL data. In many cases, using a STIL pattern file eliminates the need for an Assign File or Sequence File. You can also convert a STIL file to an Assign File and Sequence File in a separate step so these files can be modified prior to running build_testmode.

To use a STIL file as input, add `stilmodefile=<file name>` to the command line. When importing a STIL file, you may also need to add `macrodomainname` and `macroname` parameters to help Encounter Test correctly decipher the STIL data.

Partition File (Optional)

Use a partition file as input to reduce, or partition, portions of active logic to accelerate ABIST simulation. Partitioning of ABIST logic causes Encounter Test fault simulation applications to simulate only the active logic defined by the partition file. Either hierarchical blocks of interest or starting/stopping points for a backtrace may be specified in the partition file. Refer to Active and Inactive Logic on page 27 for additional information.

The file must contain specific keywords to execute properly during mode definition. The available keywords (upper case required) are:

- `INCLUDE` - marks all pins/nets in the specified block as active. Tracing is not performed, therefore any internal latches that are not affected during backtracing do not have their clocks backtraced as active.
- `REMOVE` - unmarks all pins/nets in the specified block and identifies them as inactive.
- `STOP` - tracing is stopped at the specified points. If a pin/net is a STOPPING point, the pin/net is marked active, but no tracing is performed beyond that point. If a block is marked STOP, the output of the block is marked active and tracing into the block is discontinued.
- `BACKTRACE` - backtraces the specified net/pin or all inputs of the specified block.
- `BACKTRACE INPUT` - queues all inputs of the specified block for backtracing.
- `BACKTRACE OUTPUT` - queues all outputs of the specified block for backtracing. If there is logic in the block that does not feed the outputs, then this logic remains inactive. Also, backtracing stops at scan latches; if there are scan latches inside the block, the tracing stops there (clocks are fully backtraced).

Specify blocks, pins, and nets in the following format:

- `Block.f.l.`*`blockName`*
- `Pin.f.l.`*`pinName`*
- `Net.f.l.`*`netName`*

Multiple blocks, pins, and nets maybe be specified as follows:

```
INCLUDE Block.f.l.blk.nl.xx.yy Block.f.l.blk.nl.xx.yz
    BACKTRACE Pin.f.l.blk.nl.tt.rr
    BACKTRACE Pin.f.l.blk.nl.tt.rz
    BACKTRACE Pin.f.l.blk.nl.tz Net.f.l.blk.nl.ee.ss Block.f.l.blk.nl.yy.uu
        Net.f.l.blk.nl.ee.sa Block.f.l.blk.nl.yy.ua
```

BSDL File (for 1149.1 Testmodes)

A BSDL (Boundary Scan Design Language) file is recommended by the IEEE 1149.1 standard to specify boundary scan. Refer to "Boundary Scan Design Language" on page 301 for details.

Build Test Mode Outputs

In addition to identifying scan chains, compression structures, and other DFT-related logic, build_testmode creates additional information that is used not only by ATPG and simulation, but also during interactive analysis. This information includes:

Active and Inactive Logic

Inactive logic is the logic that cannot be observed in the test mode. Since it cannot be observed, the faults in inactive logic cannot be tested and are not included in the fault count for that specific test mode. See "Identifying Inactive Logic" on page 289 for a description of how inactive logic is determined.

Design States

At some points during a test, certain primary inputs, flops, and latches take on known values. The collection of these values represents a design state. Encounter Test uses these known design states to aid test generation and simulation. These design states can be accessed during interactive analysis. See "Test Mode Design States" on page 295 for description of the various available design states.

Build Test Mode Restrictions

A test mode cannot have a grandparent mode if the patterns are intended for conversion to WGL, STIL, or Verilog.

The following are additional limitations:

- More than 256 Scan I/Os are not supported if the scan in/out pins are declared with indexes in the assign file. Removal of pin indexes removes this restriction.
- More than 128 clock pins in the TDR are not supported. To circumvent this condition, correlate your clock inputs in the test mode definition. Refer to "CORRELATE" on page 239 for details.

Identifying Test Function Pins (Advanced)

Encounter Test supports a comprehensive collection of Test Function Pin Attributes, covering a wide variety of test methodologies and scan styles. This section discusses the test function pin attributes, their use, and limitations.

Test function attributes may be specified in:

- The design source (netlist)
- Mode Definition File or Assign File (through ASSIGN statements). See "ASSIGN" on page 210

BSDL attributes override design source attributes. Attributes in the Mode Definition file override attributes in the BSDL or design source. Assign files override the pin assignment specified in the mode definition file.

Specifying Test Function Pin Attributes in Design Source

Test Function Pin Attributes apply to:

- Primary Inputs (PIs) and Primary Outputs (POs)
- Flip/Flops and Latches
- Certain Hierarchical Blocks

The method of specifying test function pin attributes in a design source is similar to adding any other model attribute. The attribute name is defined by you and the attribute value is one of the Test Function Attribute Definitions.

The attribute is selected during `build_testmode` if the mode definition file contains the following statement:

```
TEST_FUNCTION_PIN_ATTRIBUTES = <attribute name>
```

Note: Use attribute names of the form TB_*<something>* or ET_*<something>*. Encounter Test automatically saves any attributes that have this naming format. If a different name is used, you may have to list the attribute name in an Attribute Control file on the build_model command.

Refer to Modeling Logic Structures and Attributes in *Encounter Test: Guide 1: Models* for more information

Example 1: A Single Test Mode

Verilog Design Source

```
module top_level (...);
  ...
(* ET_fullscan = "-SC" ,
   ET_opmisr = "-TI" *) input reset ;
  ...
endmodule
```

Mode Definition File : myfullscan

```
TEST_FUNCTION_PIN_ATTRIBUTES = ET_fullscan ;
```

Command Example for myfullscan

```
build_testmode workdir=<directory> modedefpath=<directory>
                   modedef=myfullscan  testmode=fullscan ...
```

When the test mode is built, primary input reset will have the test function -SC or (system clock).

Example 2: A Second Test Mode

Using the same design source, another test mode can be built using a different mode definition file:

Verilog Design Source

```
module top_level (...);
   ...
   (* ET_fullscan = "-SC" ,
      ET_opmisr = "-TI" *) input reset ;
   ...
endmodule
```

Mode Definition File: myopmisr

```
TEST_FUNCTION_PIN_ATTRIBUTES = ET_opmisr ;
```

Command Example for myopmisr

```
build_testmode workdir=<directory> modedefpath=<directory>
                modedef=myopmisr  testmode=opmisr ...
```

When the test mode is built, primary input reset will have the test function -TI or (test inhibit).

Example 3: Multiple Attribute Names

It is possible to specify multiple attribute names in the `TEST_FUNCTION_PIN_ATTRIBUTES` statement. If multiple attributes are specified, test functions associated with the first listed attribute for a given pin are used to build the testmode. The `TEST_FUNCTION_PIN_ATTRIBUTES` statement determines the order in which to search the model for the attributes with matching names. For any pin, only the function pin attributes of the first attribute name as identified by this search order will be applied.

However, this process has the following exceptions:

- If a `+/-BI` or `BDY` is specified for a pin that occurs later in the search list, then that test function is added to the pin
- If `TCK` or `TMS` is specified after another clock test function, it overrides that clock test function or `TC`
- If `TCK` or `TMS` is specified before a clock test function, the clock test function overrides the `TCK`
- If `TCK` or `TMS` is specified before or after a `TC` or `TI` function, it is added to the test functions for that pin
- If `TCK` or `TMS` is specified after non-clock test functions, it overrides those test functions except for `BI` and `BDY` which, in contradiction to exception #1, override the `TMS` or `TCK` designation
- Any `TRST` or `TDI` specified before `TCK` or `TMS` override them

- If `IS` is specified before or after `TMS` or `TCK`, both the `IS` and `TMS/TCK` functions are retained
- `RS` is not a recognized function
- `CMI,CME, CMLE, CML_A, CML_B,` and `CML` are not supported for the model source

Verilog Design Source

```
module top_level (...);
  ...
  (* ET_fullscan = "-SC",
     ET_opmisr = "-TI" *) input reset ;

  (* ET_fullscan = "-SC" *) input system_clock ;
  ...
endmodule
```

Mode Definition File: myopmisr

```
TEST_FUNCTION_PIN_ATTRIBUTES = ET_fullscan, ET_opmisr ;
```

Command Example for myopmisr

```
build_testmode workdir=<directory> modedefpath=<directory>
               modedef=myopmisr  testmode=opmisr ...
```

When the test mode is built, primary input reset will have the test function -TI, and primary input system_clock will have the test function -SC.

Example 4: Multiple Test Functions on One Pin

It is also possible to assign multiple attribute test functions to a single pin.

Verilog Design Source

```
module top_level (...);
  ...
  (* ET_fullscan = "-SE, +TC" *) input scan_enable ;

  ...
endmodule
```

Mode Definition File: myfullscan

```
TEST_FUNCTION_PIN_ATTRIBUTES = ET_fullscan ;
```

Command Example for myfullscan

```
build_testmode workdir=<directory> modedefpath=<directory>
                    modedef=myfullscan  testmode=fullscan ...
```

When the test mode is built, primary input scan_enable will have the test functions -SE and +TC. During scanning, the pin will function as a scan enable with a value of 0. During test generation, the pin will function as a test constraint held at a constant value of 1.

Assignfile Examples

Refer to "ASSIGN" on page 210 for more information on the ASSIGN statement syntax.

Example 1: Correlated Input Pins

```
assign pin=D4 test_function=-TI;
CORRELATE D2 + D4;
```

The above-mentioned example correlates pin D2 to the representative pin D4. The statement `CORRELATE D2 - D4;` will be used if the pins are out of phase.

Example 2: Scan Enable and Test Constrained Pin

```
assign pin=SE test_function=-SG,+TC;
```

Consider this pin as a 0 during scan, but as 1 during the release/capture of ATPG.

Test Function Attribute Definition

The following sections summarize test functions and their usages.

Scan Inputs and Outputs

`SCAN_IN` (SI)	Used on all scan data primary inputs in the design. There is one SI primary input for each register that can be scanned into (controllable register) on the design. This attribute has no associated value. **Note:** Placing the SI attribute on a PPI is not supported.

SCAN_IN_SELECTOR (SIS)	Used on all scan-in selector primary inputs in the design. This attribute has no associated value. **Note:** Placing the SIS attribute on a PPI is not supported.
SCAN_OUT_SELECTOR (SOS)	Used on any primary inputs that are required at a specific value to force the network into the scan state with all internal scan chains observable at scan out. **Note:** Placing the SOS attribute on a PPI is not supported.
SCAN_OUT (SO)	Used on all scan data primary outputs in the design. There is one SO primary output for each register that can be scanned out of (observable scan chain) on the design. This attribute has no associated value.
CHANNEL_INPUT (CHI)	Used to denote a block or a pin that is to be considered as the starting point of an internal scan chain (channel). When specified on a block, this attribute is propagated to all output and bidirectional pins of the block. Note that the CHI attribute is disallowed on a PRPG latch.
CHANNEL_OUTPUT (CHO)	Used to denote a block or a pin that is to be considered as the ending point of an internal scan chain (channel). When specified on a block, this attribute is propagated to all output and bidirectional pins of the block.

Scan Chain Sequence Order

The order of the scan chains in the design can be determined by assigning sequence numbers to the SCAN_IN test function. Controllable registers are numbered based on scan_in. Observable scan chains are numbered based on scan_out. BIST channels are numbered based on the MISR bit positions they feed. If a sequence number is not specified with a SCAN_IN, Encounter Test assumes a sequence order of zero. If a number is specified on multiple scan ins, these scan chains are numbered according to an alphanumeric sort of the respective pin names. Using scan chain sequencing offers the following benefits:

- You can ensure the order of the scan chains in Encounter Test matches the order they are referenced in the design.
- For stored pattern tests, if manual patterns are written in vector format and a subsequent name change is made in the design, the manual patterns match the new scan chain order without the task of reordering.

The following examples show some sample SCAN_IN test function assignments and indicate the result of processing. The syntax shown here is the syntax of assignment statements which may be coded in the mode definition file or test function pin assignment file.

- Example 1:

```
assign pin1 test_function=SI1;
assign pin2 test_function=SI3;
assign pin3 test_function=SI4;
assign pin4 test_function=SI2;
```

  The result is the scan chain fed by pin1 is #1; the one fed by pin4 is #2; the one fed by pin2 is #3; and the one fed by pin3 is #4.

- Example 2:

```
assign pin=pin1 test_function=SI;
assign pin=pin2 test_function=SI;
assign pin=pin3 test_function=SI;
assign pin=pin4 test_function=SI;
```

  All the scan inputs have the same number, SI0, therefore, the sequencing is alphanumeric by pin name.

- Example 3:

```
assign pin=pin1 test_function=SI1;
assign pin=pin2 test_function=SI3;
assign pin=pin3 test_function=SI;
assign pin=pin4 test_function=SI2;
assign pin=pin5 test_function=SI2;
```

  If a sequence number is not specified with a SCAN_IN, Encounter Test assumes a sequence order of zero.

  In this example, the scan chain fed by pin3 becomes the first scan chain. The chain fed by pin1 is #2; the one fed by pin4 is #3; the one fed by pin5 is #4; and the one fed by pin2 is #5.

System and Scan Clocks

Clock Functions

SYSTEM_CLOCK (SC)	Used on all primary inputs that are intended to be used as clocks for the system data function of latches, direct set or reset of latches, or for controlling the write operation of memory arrays. These primary inputs are not used for the scan operation. The test function value (+, 1, -, or 0) sets the clock off at the latches it feeds.

`A_SHIFT_CLOCK` (AC)	Used on all primary inputs intended to be used as the master (LSSD L1) scan clock for the scan operation. The test function value (+, 1, -, or 0) sets the clock off at the latches it feeds.
`A_SHIFT_SYSTEM_CLOCK` (AS)	Used on all primary inputs intended to be used as both a system clock and an A_SHIFT_CLOCK (for example, in a multiplexed scan data implementation). The test function value (+, 1, -, or 0) sets the clocks off at the latches they feed.
`B_SHIFT_CLOCK` (BC)	Used on all primary inputs intended to be used as the slave (LSSD L2) scan clock for the scan operation. The test function value (+, 1, -, or 0) sets the clock off at the latches it feeds.
`B_SHIFT_SYSTEM_CLOCK` (BS)	Used on all primary inputs intended to be used as both a system clock and a B_SHIFT_CLOCK (for example, in a multiplexed scan data implementation). The test function value (+, 1, -, or 0) sets the clocks off at the latches they feed.
`P_SHIFT_CLOCK` (PC)	Used on all primary inputs intended to be used to load data into LSSD L3 type latches following the scan operation. The test function value (+, 1, -, or 0) sets the clock off at the latches it feeds. There can be more than one `P_SHIFT_CLOCK`.
`P_SHIFT_SYSTEM_CLOCK` (PS)	Used on all primary inputs intended to be used as both a system clock and a P_SHIFT_CLOCK.The test function value (+, 1, -, or 0) sets the clocks off at the latches they feed.

`E_SHIFT_CLOCK` (EC)	Used on all primary inputs intended to be used to scan data into edge-sensitive flip-flops during the scan operation. There can be more than one `E_SHIFT_CLOCK`. There is typically no value that would turn the EC clock "off" at all the flip-flops it feeds, so here are some hints for setting the value (+, 1, -, or 0): ■ The value on the clock input should flush the slave (of the master-slave latch pair). This means the master is off. This is especially important when the master is a multi-port latch or when there is clock gating. ■ If there is clock gating, the clock input to the gating logic should be at the controlling value (for example, at zero for an AND).
`E_SHIFT_SYSTEM_CLOCK` (ES)	Used on all primary inputs intended to be used as both a system clock and a `scan_E_clock`. (for example, in a multiplexed edge-triggered scan design). The value (+, 1, -, or 0) is set using the hints shown for the `E_SHIFT_CLOCK`.

Scan Clock Sequence Number

To allow some control over the order in which scan clocks of a given type are pulsed during the scan operation, Encounter Test allows a sequence number to be specified with the scan clock test functions (for example, AC1). Two clock primary inputs that have identical scan clock test functions and sequence numbers are pulsed simultaneously. Thus in Encounter Test, the scan sequence (the sequence of clock pulses, and possibly primary input stimulus, necessary to shift data one bit position from scan-in to scan-out of a scan chain) is defined as follows:

- Apply scan state
- Pulse all `A_SHIFT_CLOCKs` in user-defined order (AC*n*/AS*n*)
- Pulse all `E_SHIFT_CLOCKs` in user-defined order (EC*n*/ES*n*)
- Pulse all `B_SHIFT_CLOCKs` in user-defined order (BC*n*/BS*n*)

Where *n* is the sequential order number. For example, if your test function specifications for different pins are:

```
AC, EC, BC1, AC1, AC4, EC5, ES15, AS7, BC10, BS10
```

then the scan clock sequence is:

```
Event #
-------
1           Pulse AC
2           Pulse AC1
3           Pulse AC4
4           Pulse AS7
5           Pulse EC
6           Pulse EC5
7           Pulse ES15
8           Pulse BC1
9           Pulse BC10 and BS10
```

Notes:

1. There is no checking for race-free operation when multiple clocks are pulsed simultaneously, as in event 9. If you cannot guarantee that such an operation is race-free, you should assign a unique number to each clock so that only a single clock is pulsed at a time.
2. There must be no blank between the flag and the number.
3. A blank following the flag is equivalent to a zero.

Oscillators

`OSCILLATOR` (OSC)	Used on a primary input that is to be connected to a tester-supplied oscillator signal, where the tester applies some constant number of pulses on the pin in each tester cycle. This attribute identifies the pin as a synchronous (tester-supplied) oscillator and specifies a stability state (i.e. an OFF value). This attribute can be specified only with either a + or - sign preceding it. This attribute is also allowed on PPIs.
`oTI`	This is a special case of the TEST_INHIBIT attribute, where the pin is permanently connected to a free-running (asynchronous) oscillator. In this case, the “value” the pin is tied to is coded as “o”.

Clock Gates

Note: These test function attributes are supported on pseudo primary inputs as well as primary inputs.

CLOCK_ISOLATION (CI)	Used on a primary input that controls the gating of combined system and scan clocks (AS, BS, PS, ES) on the design. The test function value (+, 1, -, or 0) causes the combined clocks to act as scan clocks. The opposite of the specified value causes the combined clocks to act as system clocks.
SCAN_ENABLE (SE)	Used on any primary inputs that are required at a specific value to force the network into the scan state. That is, they provide pre-conditioning required to make the scan operation work. The test function value (+, 1, -. 0, or Z) is the value the primary input is set to before, and held at throughout, subsequent scan operations. This is usually for the purpose of gating either a shift clock or the scan data. After the scan operation is completed, the scan_enable primary inputs can be switched to a different value. **Note:** Encounter Test supports the use of SCAN_GATE (SG) as a synonym for SCAN_ENABLE.

Note: If you have a bi-directional (inout) pin that is sourced by a three-state device you need to ensure that the TEST_FUNCTION_PIN_ATTRIBUTE of the pin doesn't cause three-state burnout during scan. This can be accomplished by using a SCAN_ENABLE with a value of high impedance (ZSE) on the inout pin. If the inout pin has a test function attribute of SCAN_OUT. Encounter Test automatically adds a TEST_FUNCTION_PIN_ATTRIBUTE of SCAN_ENABLE with a value of Z (ZSE) to this pin.

If a custom scan sequence is used, the scan preconditioning sequence must stimulate this pin to Z, and this is the responsibility of the user. Encounter Test does not modify the custom scan sequence definitions.

Test Inhibits, Test Constraints and Lineholds

`TEST_INHIBIT` (TI) Used on all primary inputs, pseudo primary inputs, and fixed value latches that you want forced to a fixed value throughout test generation (not only during scan, like the SE does). The test function value (+, 1, -, 0, Z, X, or o) is forced on the primary input. Pseudo primary inputs (PPIs) are forbidden to be designated as oTI (which would be interpreted as permanently connected to an oscillator). All other values listed are acceptable on a TI pseudo primary input. Z, X, and o are not valid for fixed value latches. For fixed value latches, this test function serves only for checking; it allows Test Structure Verification to verify that all latches having a +TI test function are, in fact, fixed value latches and were initialized to the specified state. For additional information, refer to:

- "Logic Test Structure Verification (TSV)" in the *Encounter Test: Guide 3: Test Structures*.
- "Guideline TB.9 - Fixed Value Latches" in the *Encounter Test: Reference: Legacy Functions*.

`TEST_INHIBIT` can be used to:

- Break feedback loops (for example, ring oscillators) that would cause test generation problems.
- Disable signals that would cause clock races.
- Hold fixed value latches at the initialized state (for example, by using a TI to force the clock inputs off at the latch).
- Remove a section of logic from consideration during testing in this test mode (for example, inhibit external logic during boundary scan internal testing).

Additional TI Considerations:

- Note that there is a difference between values asserted by `TEST_INHIBIT` pins and those forced by TIE primitive blocks. The value provided by the TI pin or latch applies only in the test mode being processed. This primary input or latch probably will be switched to the opposite value during some functional mode of operation (or even in a different test mode). A TIE primitive block is assumed to represent a permanent physical connection to, say, a power rail, and has the same value in all test modes and functional modes.
- A TI flag (or a Clock flag) can be assigned to a CIO only if the chip driver is guaranteed to be in a safe (HIGH-Z) state. A safe chip driver can be guaranteed in one of two ways:

- Assign TI flags to other chip PIs (not CIOs) that will force the chip driver to a Z state.
- Provide a mode initialization sequence that first puts a Z on the CIO input, then sets the chip driver to a Z state, then finally sets the TI value on the CIO. The internal chip state that forces the chip driver to Z must be maintained throughout all testing in this mode. Therefore, all other PIs that have to be held steady to maintain this chip internal state would also have to be assigned TI flags. Note that this chip internal state can not be influenced by any scannable latches in this mode.

See "Coding an Externally Specified Initialization Sequence" on page 68 for additional information.

`TEST_CONSTRAINT` (TC)	Used on all primary inputs and pseudo primary inputs that you want forced to a fixed value for test generation but not for scanning. The test function value (+, 1, -, 0, or Z) is forced during test generation, and may in turn cause certain latches to then take on the TC characteristic.
`LINEHOLD` (LH)	Used to hold a primary input pin or pseudo primary input to value (+, 1, -, 0, or Z) during test generation. It has no test significance other than that it has been determined that the automatic test generation process would be most effective with this pin held to the specified logic value. Lineholds can be overridden or removed during individual test generation, fault analysis, or macro structure verification runs. Lineholds are ignored during other Encounter Test applications.

Test Constraint Sequence Order

Turning a clock ON may cause some flip-flops to change state, and thus, turning ON multiple clocks simultaneously increases the chance of races. The exposure to races is especially great if the design responds to the clocks in question in a level-sensitive manner. For some designs, one clock being ON may block the propagation of some other clock, and thus prevent flip-flops from changing at all when the clocks are switched in the correct order.

To control the order of switching, append a number to the `TC` attribute to indicate the order in which they are set to their value within the scan exit sequence. A `TC` pin with no number is assigned a sequence number of `0`, i.e. `TC0`. Subsequent ordering of `TC` pins can be specified in the following manner: `TC1`, `TC2` etc. The `TC` pins will be set to their designated `TC` states in increasing order when leaving the scan state. When re-entering the scan state, those `TC` pins that must be switched are set to their designated scan states in the reverse order (in decreasing order).

A warning message is issued if there are multiple clocks that are TC'd ON, and that have no `TC` number or have the same `TC` number.

Latches

`FINITE_STATE_MACHINE` (FSM)	Used on latches only, this test function identifies a latch which assumes a specific state after the application of a homing sequence (which must be included in the modeinit sequence) but that cannot be brought to that state from an initial unknown state by standard three-valued simulation. The logic value assigned with this test function may be either unspecified or else 1, +, 0 or -. If unspecified then Encounter Test will automatically determine the state that this latch achieves through application of the modeinit sequence. Refer to "Mode Initialization Sequences (Advanced)" on page 66 for additional information.
`FIXED_VALUE_LATCH_LINE_HOLD` (FLH)	Used on latches only, this test function works like an LH on a primary input. The difference between FLH and TI on a latch is that, as on primary inputs, the linehold value can be overridden by the user for specific test generation runs and the TI cannot. Test Structure Verification verifies that all latches having a +FLH test function are, in fact, fixed value latches and were initialized to the specified state. Refer to "Logic Test Structure Verification (TSV) in *Verification and Analysis Reference* and "Guideline TB.9 - Fixed Value Latches" in *Encounter Test: Reference: Legacy Functions* for additional information.
`TEST_INHIBIT` (TI)	Refer to "Test Inhibit (constant during scanning and testing)" on page 23.

Three-State Driver Control

`BIDI_INHIBIT` (BI)
: Used on a primary input that forces primary outputs fed by three-state drivers (TSDs) to high-impedance (Z). The purpose of this primary input is to enable manufacturing to easily generate a parametric test to confirm that TSDs can achieve the high impedance state. If the product contains TSDs, then Test Structure Verification checks that such a pin exists and performs the inhibiting function for all TSDs that directly drive primary outputs. The value (+, 1, -, or 0) is the value that forces the outputs to Z. Refer to Three State Driver (TSD) Primitive for additional information in *Encounter Test: Guide 1: Models*.

Simultaneous Output Switching Control

`OUTPUT_INHIBIT` (OI)
: Used on a primary input or pseudo PI that disables output drivers for the purpose of avoiding noise induced by crosstalk or excessive power supply drain during switching. The test function value (+, 1, -, or 0) is the value that inhibits the output drivers and is the state at which the pin will be set during scanning.

Logic Built-In Self Test (LBIST) Controls

`MISR_ENABLE` (ME)
: Used on primary inputs and pseudo primary inputs that gate the operation of an on-board signature register (MISR). The test function value (+, 1, -, or 0) is the value the primary input is set to during a normal scan operation, when the MISR is enabled. Performing the scan operation with all MEs set opposite to their respective test function values forces all MISR latches to be held to their initial values (their values at the beginning of the scan).

`WEIGHT_SELECT` (WS)	Used on primary inputs and pseudo primary inputs that control the channel input signal weight selection for an LBIST structure that implements weighting. The test function value (+, 1, -, or 0) is the value that causes the scan chain inputs to have a signal probability of 0.5 -- equiprobable 1s and 0s -- when all WS pins are set to their test function values. Refer to the description for *Channel input signal weighting* in the "LBIST Concepts" section of *Automatic Test Pattern Test Generation User Guide*.
`PRPG_SAVE` (PV)	Used on primary inputs and pseudo primary inputs that cause the current PRPG state to be saved during a channel scan. This is used for the "fast forward with pins" implementation. The test function value (+, 1, -, or 0) causes the current PRPG state to be saved into the "PRPGSAVE register". If a PV pin is identified and a PRPGSAVE sequence is also defined, the PV pin is held to its inactive state (is ignored). Refer to the description of Fast Forward in the "LBIST Concepts" section of *Automatic Test Pattern Test Generation User Guide*. **Note:** In the TI state, the PV pin is set opposite to its assigned PRPG restore or save states; that is, treated as an implicit TI pin.
`PRPG_RESTORE` (PR)	Used on primary inputs and pseudo primary inputs that cause the current PRPG state to be restored during a channel scan. This is used for the "fast forward with pins" implementation. The test function value (+, 1, -, or 0) causes the current PRPG state to be restored from the "PRPGSAVE register". If a PR pin is identified and a PRPGRESTORE sequence is also defined, the PR pin is held to its inactive state (is ignored). Refer to PRPG save and restore operations and fast forward with pins in the "LBIST Concepts" section of the *Encounter Test: Guide 5: ATPG* for additional information. **Note:** In the TI state, the PR pin is set opposite to its assigned PRPG restore or save states; that is, treated as an implicit TI pin.
`PRPG_LOAD` (PLD)	Identifies a clock which loads the PRPG from the scan inputs.
`PRPG_LOAD_ENABLE` (PGE)	Identifies a signal that establishes the design state so that the `PRPG_LOAD` clocks can be pulsed without affecting the contents of the channel latches.

On-Product MISR Controls

These test function pins are specified when creating an On-Product MISR test mode. Prior to creating this test mode, you must first create a test mode for diagnostics scan out. Refer to On-product MISR (OPMISR) Test Mode and Appendix B, “DIAGNOSTIC_MODE” for details.

MISR_OBSERVE (MO)	Used to identify the output pins where the contents of a MISR will be observed.

Tips:

- MO pins do not have to also be SI pins other than to help keep the number of test function pins to a minimum.
- MISR bits can feed to a MO pin through an XOR or XNOR network. This is useful when there are more MISR bits than MO pins. See XOR Primitive Function and XNOR Primitive Function in *Encounter Test: Guide 1: Models.*
- There is some increased chance of signature aliasing when MISR bits are XORed together. It is suggested that no less than 16 MO pins be used. It is also suggested to use multiple MISRs with different polynomials to help reduce the chance for aliasing.

MISR_RESET (MRST)	Used to identify a clock that resets the MISR L1 latches to logic zero when pulsed.
MISR_RESET_ENABLE (MRE)	Allows the MISR_RESET clock to be shared with some other clock function by shutting of the MSR_RESET clock during all other functions. The ZMRE test function pin may also be specified. ZMRE indicates the MRE will have a value of Z when its test function is applied.
MISR_READ (MRD)	Identifies a signal that causes the product MISRs to be seen at the MISR_OBSERVE pins when all MRD pins are asserted while the design is in the scan state.

SCAN_IN_GATE (SIG)	Identifies a primary input which must be at the specified value when scanning into the chip. This pin is necessary only if data is not being scanned out to the tester at the same time a new test is being scanned in. During scan out, this pin is not required to be at any specified value unless it also has another attribute such as SCAN_OUT_GATE.
SCAN_OUT_GATE (SOG)	Identifies a primary input which must be at the specified value when scanning data out of the chip to the tester. When data is being scanned in, this pin is not required to be at any specified value unless it also has an attribute such as SCAN_IN_GATE.
SCAN_OUT_FILL (SOF)	If a test mode has been created where scan-in and scan-out are not overlapped, this pin function identifies the value to be placed at the top of the scan chains during the scan-out operation. This serves two purposes; first, to ensure that known values are placed in the scan chains so that predictable results appear on the scanouts of the short scan chains, and second, to enable an LSSD flush test.
Channel_Mask_Input (CMI)	These are optional in that SI pins will be assumed to be CMI pins. If there are no CMI pins defined and there is a CML (or CML_A or CML_B) pin defined, a CMI specification is generated for each SI pin.
Channel_Mask_Enable (CME)	This pin defines how the channel masks are applied based on scan cycle. The specified polarity identifies the CME state that does not gate out any channel. A maximum of four CME pins (excluding any pins that may be correlated to a CME pin) may be specified. The default is to add a TC to the same value for each CME pin. Specify a number (e.g. CME0 and CME1) to denote an ordering for these pins when treated as a bus signal (e.g. CME0,CME1 = 00). Digits 0-3 are supported. If a digit is not specified, the test mode build process assigns one automatically.

Channel_Mask_Load_Enable (CMLE)	The polarity of the CMLE pins defines the design state on top of the scan design state in which the channel masks are to be loaded. CMLE pins are needed only if the CML clock is also a scan clock such that the application of the CMLE pins forces the CML clock to perform its mask bit loading function rather than the scan loading function. A CMLE pin should also be an SG/SE pin of the opposite polarity; the test mode define process adds this if has not been specified as such. If an SE is specified to the same polarity on the CMLE pin, a warning message is issued stating that it is unlikely to work correctly. Opposite polarities are most likely to work.
Channel_Mask_Load (CML)	This clock, when pulsed in the CMLE design state, will serially load in the mask bits from the CMI/SI pins. If there is a CML pin defined, then there must be at least one CME pin defined. This is assumed to be an edge-based clock for the purpose of scan loading the channel mask registers.
Channel_Mask_Load_A (CML_A)	This clock, when pulsed in the CMLE design state, combines with the CML_B clock to serially load in the mask bits from the CMI/SI pins using an LSSD style of scan clocking.
Channel_Mask_Load_B (CML_B)	This clock, when pulsed in the CMLE design state, combines with the CML_A clock to serially load in the mask bits from the CMI/SI pins using an LSSD style of scan clocking.

All of these preceding signals with the exception of SCAN_IN_GATE, SCAN_OUT_GATE, and SCAN_OUT_FILL may be specified only in the On-Product MISR test modes.

OPCG Controls

Specify these test function pins when creating an OPCG test mode. Refer to OPCG Test Modes and “OPCG” on page 242 for more information.

Oscillator (OSC)

The specified stability value represents the OFF state of the signal with regard to simulation of pulses along that signal.

Refer to "OSCILLATOR (OSC)" on page 37 for additional information.

GO

The specified stability value identifies the state of the GO signal that holds the OPCG logic quiescent. Switching the GO signal to its opposite value causes the launching of internally generated clocks. The specified stability value is held on the GO signal at all times except when launching an OPCG clocking sequence.

OPCG_Load_Input (OLI)

This test function identifies a scan data input pin used to serially load an OPCG register. If an OLC (OPCG Load Clock) is defined without any OLI pins being defined, all SI pins are presumed to also be OLI pins. If Encounter Test reports that it cannot verify the loading of the OPCG Register Flip Flop or if a subset of SI pins are the OLI pins then you must provide the OLI test function as input.

OPCG_Load_Enable (OLE)

The polarity of these pins defines the design state on top of the TG design state in which the OPCG control registers are loaded. OLE pins are used similarly to CMLE pins for loading mask registers except that control OPCG registers are loaded directly from the TG state and not during scan operations.

OPCG_Load_Clock (OLC)

This clock, when pulsed in the OLE design state, serially loads the control register bits from the OLI/SI pins. This is assumed to be an edge-based clock for the purpose of scan loading the OPCG control registers.

OPCG_Load_Clock_A (OLC_A)

This test function pin is the LSSD A clock equivalent to OLC.

OPCG_Load_Clock_B (OLC_B)

This test function pin is the LSSD B clock equivalent to OLC.

RPCT Boundary Controls

`BOUNDARY_DATA_PIN` (`BDY`)	Used on any primary input pin or primary output pin that is to be contacted during RPCT boundary scan for internal testing, but has no other test function. There is no test function value for BDY. For mixed analog/digital designs, use the BDY value to identify the pins that can be controlled and/or observed during ATPG.
`CONTROL_PIN` (CTL)	Used on any primary input pin or primary output pin that has some unspecified test function which is of no concern in the generation of test data. Encounter Test treats these the same as BDY; that is, it is a pin that will be contacted during RPCT boundary scan internal testing. Although Encounter Test need not be aware of the pin's function, this designation allows manufacturing to distinguish it from uncontacted pins and RPCT boundary data pins (BDY). There is no test function value for CTL.
`NO_INTERCONNECT` (NIC)	Used on a chip pin to indicate that this pin is not to take part in the interconnect test generation process for the higher level structure (MCM/card). This specification is ignored unless the Mode Definition includes `BOUNDARY=EXTERNAL`, MODEL on the SCAN statement (refer to "SCAN" on page 197 for more information), indicating that the only purpose of this test mode is to enable construction of a chip boundary model for interconnect test generation. There is no test function value for NIC. Refer to "Create Interconnect Tests" in *Encounter Test: Reference: Legacy Functions* for additional information.

1149.1 Boundary Controls

These controls are normally specified in the BSDL (Boundary Scan Design Language) but can be specified in the model source or in the test mode definition file.

`TEST_CLOCK` (TCK)	Used on a primary input that is intended to be the clock that controls the IEEE 1149.1 test logic. A TCK value may be specified, though it is automatically assumed to be 0 (the inactive clock state). Any non-zero value, which is specified, will be ignored and 0 will be used.
`TEST_MODE_SELECT` (TMS)	Used on a primary input that synchronizes IEEE 1149.1 test operations. This primary input is used with the TCK clock to change states in the Test Access Port (TAP) Controller. A logic value may be specified for this pin, but it is meaningless and will be ignored, since the term “active state” does not apply to the TMS pin. Depending on the present state of the TAP Controller, a state change may take place with TMS either at 0 or at 1 whenever TCK is pulsed.
`TEST_RESET` (TRST)	Used on a primary input that is used for asynchronous reset of the TAP Controller. To reset the TAP Controller asynchronously this pin is set to 1-0-1. A logic value may optionally be specified for this pin, but if it is anything other than 1, the inactive state, the value will be ignored and 1 assumed.
`TEST_DATA_INPUT` (TDI)	Used on a primary input that is the scan input for IEEE 1149.1 scan. There is no value specified with this test function.
`TEST_DATA_OUTPUT` (TDO)	Used on a primary output that is the scan output for IEEE 1149.1 scan. There is no value specified with this test function.

Pin requirements for an 1149.1 test mode

In order for a test mode to be recognized by Encounter Test as a legitimate 1149.1 test mode, it is necessary that at least one each of the following pins be identified: TCK, TMS, TDI, and TDO. The TRST pin is optional.

Test Function Pins for an 1149.1 Mode

The Boundary Scan Design Language (BSDL) file defines the test function pin attributes for an 1149.1 test mode. The test function pins that support 1149.1 applications are TCK, TMS, TDI, TDO, and TRST. TRST is optional when a TRST pin is not defined for the design.

For 1149.1 components that support multiple test strategies such as LSSD and 1149.1, there may be a “compliance enable” pattern required for the component to put it into the 1149.1 test mode. The compliance enable pattern is a set of test function pins and their values which, when applied to the device, enable and hold the device in the 1149.1 mode. The BSDL statement `COMPLIANCE_PATTERNS` defines the compliance pattern for a design. *Build Test Mode* uses the information defined in the BSDL statements `COMPLIANCE_PATTERNS` and scan port identification (for example, attribute `TAP_SCAN_CLOCK`) to define the test function pins for an 1149.1 mode.

Refer to “Creating an 1149.1 Test Mode for Boundary Scan Verification” in the *Verification and Analysis Reference* for additional information.

Secondary Test Function Attributes for 1149.1 Stored Pattern Test Generation

If the test mode is for the purpose of 1149.1 stored pattern test generation, either TAP scan or LSSD/GSD scan, then Encounter Test automatically creates certain secondary test function assignments for the 1149.1 pins. This is to allow test generation applications to process this test mode normally, without having to be aware of its special 1149.1 nature. For example, if a user assigns the TDI test function attribute to a pin, then while creating a test mode, Encounter Test automatically supplements the pin with an SI test function attribute.

The following table describes the automatic secondary test function assignments made by *Create a Test Mode*.

User-assigned test attribute		Generated secondary test attribute	
		TAP scan	LSSD/GSD scan
(1)	TDI	SI	(none)
(2)	TDO	SO	(none)
(3)	TCK ***	-ES	-TI
(4)	TRST ***	+TI	+TI (inactive)
(5)	TMS ***	(see box)	+TI (to hold TLR state)

```
*** Polarity specification is optional for TCK, TRST and TMS pins.
   If specified it is ignored.
```

Secondary test function for TMS pin:

The secondary test function assignment for a TMS pin depends on the TAP_TG_STATE specified, according to the following table:

TAP_TG_STATE	secondary TMS pin assignment
RUN_TEST_IDLE	-TC
TEST_LOGIC_RESET	+TC
PAUSE_DR	-TC
SHIFT_DR	-TI
CAPTURE_DR	-TC

See "TAP_TG_STATE" on page 199 for additional information.

Power-Up Safe State Controls

These controls enable embedded devices to power-up in a *safe* state to either prevent damage or hold the device configuration.

.

REQUIRED_STATE (RS)

The RS pin provides for the definition of the power-up state. The flag specifies the value (+, -, 1, 0) required on an input of an embedded cell. This flag is usually coded in the TPGTECH cell definitions provided by technology.

A VIM coding example:

```
PPIN TESTISOLATEB I TESTISOLATEB TB_KFLAG=-RS
```

where TB_KFLAG is the test function pin attribute that specifies a 0 (-RS) must be applied to pin TESTISOLATEB upon power-up. This flag can be specified more than once. To apply this flag to a specific test mode, it must be included in the list of attributes specified on the TEST_FUNCTION_PIN_ATTRIBUTES mode definition statement. Refer to Appendix B, "TEST_FUNCTION_PIN_ATTRIBUTES".

INITIAL_STATE (IS)

The IS pin attribute specifies an initial value (+, -, 1, 0) to be applied to the associated primary input pin by the first pattern of the mode initialization sequence. This flag may be placed on one or more chip primary inputs. When coded directly in the model, the IS flag must be associated with a test function pin attribute.

A VIM coding example:

```
PPIN TESTISOLATEB1 I TESTISOLATEB1 TB_KFLAG=-TI
TB_EFPGA=-IS
```

where TB_EFPGA is the test function pin attribute that specifies that a 0 (-IS) is to be placed on primary input pin TESTISOLATEB1. This flag can be specified more than once. To apply this flag to a specific test mode, it must be included in the list of attributes specified on the TEST_FUNCTION_PIN_ATTRIBUTES mode definition statement. Refer to Appendix B, "TEST_FUNCTION_PIN_ATTRIBUTES". An alternate method of specifying the flag is within the ASSIGN statement. Refer to Appendix B, "ASSIGN".

Important

The IS flag differs from the RS flag in that the RS flag defines a required state which must exist on an internal cell input pin after application of the first pattern of the test mode initialization sequence. The IS flag specifies a value to be established on a chip primary input pin in the first pattern of the test mode initialization sequence.

Determining Which Test Functions to Use

The following tables indicate the test functions you might use for different scan design styles/ test methodologies. These tables show the test functions for a "pure" implementation of the specified scan design style. Since Encounter Test supports mixed scan design styles, you should look at the tables for each scan style you have included and ensure that your test function pin specifications are complete. Encounter Test isn't rigid about the test function attributes you use. If the attributes you define allow you to run Test Structure Verification smoothly, your test data will be good.

Test Function	LSSD Scan	GSD MUXed Scan	GSD Clocked Scan	GSD Gated Clock	GSD Gated Data
AC or AS	Yes				
BC or BS	Yes				
PC or PS	Opt				
EC or ES		Yes	Yes	Yes	Yes
SC	Yes	Opt	Yes		
SE	Opt	Yes		Yes (1)	Yes
CI	Opt			Yes (1)	
TI	Opt	Opt	Opt	Opt	Opt
TC	Opt	Opt	Opt	Opt	Opt
FLH	Opt	Opt	Opt	Opt	Opt
SI	Yes	Yes	Yes	Yes	Yes
SO	Yes	Yes	Yes	Yes	Yes
OI	Opt	Opt	Opt	Opt	Opt
LH	Opt	Opt	Opt	Opt	Opt

Test Function	Three-states on Circuit	Built-In Self Test	RPCT Boundary Scan	IEEE 1149.1 Boundary Scan (2)
BI	Yes			
BDY			Yes	
CTL			Opt	
TCK				Yes
TMS				Yes
TDI				Yes
TDO				Yes
TRST				Opt
ME		Opt		
PV		Opt		
PR		Opt		
WS		Opt		

Notes:

1. Either a SE/SG or CI must be specified.
2. The test function definition for 1149.1 Boundary Scan will normally come in through the use of BSDL (Boundary Scan Description Language). The capability to provide it through attributes in the design source or in the test mode definition file is to allow for automatic generation of BSDL from the Encounter Test model without requiring a “skeleton BSDL” file as input.

Valid Combinations of Test Functions

There are restrictions on the combinations of test function attributes that can be specified on a primary input, primary output, pseudo primary input, or a latch for a single test mode. For an obvious example, your `scan_in` pin cannot also be your `system_clock`. And, although you can have one pin that is the MISR_enable in an LBIST test mode and a `scan_enable` in an LSSD test mode, you cannot have a pin that is both the MISR_enable and a `scan_enable` in the same mode. Note that Encounter Test supports a synonym of `scan_gate` for `scan_enable`.

The following is an overview of the test function attributes grouped according to the scope of their application. Note that some attributes exist to identify the pin's function during scan operations; other attributes apply for operations performed between scan operations.

scan-in	Attributes that apply during scan-in operations. With the exception of CI, all pins in this category can, in theory, have some functional purpose that is not related to test. All of them, including CI, are enforced only during scan; at other times they can be manipulated freely according to the needs of a specific test or functional operation.
scan-out	Attributes that apply during scan-out operations.
non-scan	Attributes that apply during non-scan operations. Encounter Test supports two attributes that fall into this category: ■ LH The line hold attribute is unique in being the only one that can be overridden during execution of test generation applications. Whether originating from a test function attribute or a line hold file, these specified values are not enforced during scan. ■ TC The test constraint (TC) attribute applies to a pin that must retain a fixed value for all phases of test generation but is permitted to change for the scanning operation.

global	Attributes that apply all the time (for both scan and non-scan operations). Clock states, even those for shift clocks, are in force at all times unless explicitly stated otherwise by the appearance of a `TC` attribute on the pin. The output inhibit (OI) attribute, by definition, must be enforced during application of any patterns, either scan or non-scan, to minimize switching activity on output drivers. The PRPG save and restore attributes (PV and PR) are applied at the appropriate points during scan operations. See PV for more information on PV and PR attributes. According to the strict definition of this category, one could argue that `NIC` does not belong here, but it is included to make it fit the rules defined below, and the rationale is that a `NIC` pin is not used for interconnect test during both scan and non-scan operations. Refer to "Rules for Determining the Valid Combinations" on page 64.
manufacturing	Attributes whose purpose is related neither to scan nor to non-scan operations. Generally speaking, this category of attributes has relevance only for post-Encounter Test applications, such as the test data converters and test controllers. The bidi inhibit (BI) attribute was defined so that Manufacturing can easily generate their own tests for high-impedance measurements. BDY and CTL identify pins which are to be contacted during internal boundary scan chain testing. The proper use of BDY and CTL is on any internal boundary scan contacted pin that does not have another test function. Encounter Test assumes that all test function pins are contacted for internal boundary scan. Allowed combinations of test function attributes that can be on the same pin in the same test mode are depicted in Figure Figure 2-2 on page 69. The corresponding legend is described in "Rules for Determining the Valid Combinations" on page 64.
latches	Attributes applied to latches/flops. Refer to "Latches" on page 41 and to "Rules for Determining the Valid Combinations" on page 64.
misr observe	Attributes that are applied for an OPMISR test mode and whose values are used in the `MISR_OBSERVE` design state. Refer to "OPMISR Test Modes" on page 111 and "MISR Observe" on page 298.

prpg load	Attributes applied to LBIST test modes. Refer to "Logic Built-In Self Test (LBIST) Controls" on page 42 and "PRPG Load" on page 299.
channel mask	Test function attributes that support OPMISR or OPMISRplus test modes. Refer to "OPMISR Test Modes" on page 111.
OPCG	Test function attributes that support an OPCG test mode. Refer to OPCG Test Mode.

Refer to the following for matrices of the allowed combinations:

- Table 2-2 on page 57 for Scan In pins
- Table 2-3 on page 58 for Scan Out pins
- Table 2-4 on page 59 for Non Scan pins
- Table 2-5 on page 59 for MISR Observe pins
- Table 2-6 on page 60 for Manufacturing pins
- Table 2-7 on page 60 for Latch pins
- Table 2-8 on page 61 for Global pins
- Table 2-9 on page 63 for MISR Reset pins
- Table 2-10 on page 63 for PRPG Load pins
- Table 2-11 on page 63 for Channel Mask pins
- Table 2-12 on page 64 for OPCG pins

Table 2-2 Scan In Test Function Pin Combinations

Pin Category	Pin Function	Allowable Combinations	Rule
Scan In	CI	MRD MO LH TC MRE PGE IS	Rule 3
		BDY BI CTL	Rule 2
	ME	MRD MO LH TC MRE PGE IS	Rule 3
		BDY BI CTL	Rule 2
	SE/SG	MRD MO LH TC MRE PGE SO TDO IS GO OLE	Rule 3
		BDY BI CTL	Rule 2
		SO TDO	Rule 6
	SI	SOG SO TDO SOF MRD MO LH TC MRE PGE CMI IS OLI	Rule 3
		BDY BI CTL	Rule 2
		TDI	Rule 6
	SIG	SOG SO DO SOF MRD MO LH TC MRE PGE IS	Rule 3 Rule 6
		BDY BI CTL	Rule 2
	WS	SOG SO TDO SOF MRD MO LH TC MRE PGE IS	Rule 3 Rule 6
		BDY BI CTL	Rule 2
	TDI	SOG SO TDO SOF MRD MO LH TC MRE PGE IS	Rule 3
		BDY BI CTL	Rule 2
		SI	Rule 6

Table 2-3 Scan Out Test Function Pin Combinations

Pin Category	Pin Function	Allowable Combinations	Rule
Scan Out	CI	MRD MO LH TC MRE PGE IS	Rule 3
		BDY BI CTL	Rule 2
	ME	MRD MO LH TC MRE PGE IS	Rule 3
		BDY BI CTL	Rule 2
	SE/SG	MRD MO LH TC MRE PGE IS CMLE	Rule 3
		BDY BI CTL	Rule 2
		SO TDO	Rule 6
	SOG	SI SIG WS TDI MRD MO LH TC MRE PGE IS	Rule 3
		BDY BI CTL	Rule 2
		SO TDO	Rule 6
	SO	SI SIG WS TDI MRD MO TI LH TC MRE PGE IS	Rule 3 Rule 6
		BDY BI CTL	Rule 2
		SE/SG SOG TDO	Rule 6
	TDO	SI SIG WS TDI MRD MO LH TI TC MRE PGE IS	Rule 3 Rule 6
		BDY BI CTL	Rule 2
		SE/SG SOG SO	Rule 6
	SOF	SI SIG WS TDI MRD MO LH TC MRE PGE IS	Rule 3
		BDY BI CTL	Rule 2

Table 2-4 Non Scan Test Function Pin Combinations

Pin Category	Pin Function	Allowable Combinations	Rule
Non Scan	LH	CI ME SE/SG SI SIG WS TDI SOG SO TDO SOF MRD MO MRE PGE IS	Rule 3
		BDY BI CTL	Rule 2
	TC	CI ME SE/SG SI SIG WS TDI SOG SO TDO SOF MRD MO MRE MRST PGE IS CMLE CME CML_A CML_B OLE OLC OLC_A OLC_B	Rule 3
		AC AS BC BS EC ES PC SC TCK	Rule 5
		BDY BI CTL	Rule 2

Table 2-5 MISR Observe Test Function Pin Combinations

Pin Category	Pin Function	Allowable Combinations	Rule
MISR Observe	MRD	CI ME SE/SG SI SIG WS TDI SOG SO TDO SOF MO LH TC MRE PGE IS	Rule 3
		BDY BI CTL	Rule 2
	MO	CI ME SE/SG SI SIG WS TDI SOG SO TDO SOF LH TC MRD MRE PGE IS CMI CME	Rule 3
		BDY BI CTL	Rule 2

Table 2-6 Manufacturing Test Function Pin Combinations

Pin Category	Pin Function	Allowable Combinations	Rule
Manufacturing	BDY	CI ME SE/SG SI SIG WS TDI SOG SO TDO SOF MRD MO LH TC AC AS BC BS EC ES PC SC TCK PR PS PV NIC OI OSC TI TMS TRST MRST MRE PLD PGE IS CMI CMLE CML CML_A CML_B	Rule 2
	BI	CI ME SE/SG SI SIG WS TDI SOG SO TDO SOF MRD MO LH TC AC AS BC BS EC ES PC SC TCK PR PS PV NIC OI OSC TI TMS TRST MRST MRE PLD PGE IS	Rule 2
	CTL	CI ME SE/SG SI SIG WS TDI SOG SO TDO SOF MRD MO LH TC AC AS BC BS EC ES PC SC TCK PR PS PV NIC OI OSC TI TMS TRST MRST MRE PLD PGE IS	Rule 2

Table 2-7 Latch Test Function Pin Combinations

Pin Category	Pin Function	Allowable Combinations	Rule
Latch	TI	None	None
	FLH	None	None
	FSM	None	None

Table 2-8 Global Test Function Pin Combinations

Pin Category	Pin Function	Allowable Combinations	Rule
Global	AC	BDY BI CTL TC CML IS OLC OLC_A OLC_B	Rule 2
		SC	Rule 5
		MRST PLD	Rule 4
		TC	Rule 7
	AS	BDY BI CTL CML CML_A IS OLC OLC_A OLC_B	Rule 2
		MRST PLD	Rule 4
	BC	BDY BI CTL TC CML CML_A IS OLC OLC_A OLC_B	Rule 2
		SC	Rule 5
		MRST PLD	Rule 4
		TC	Rule 7
	BS	BDY BI CTL CML CML_B IS OLC OLC_A OLC_B	Rule 2
		MRST PLD	Rule 4
	EC	BDY BI CTL CML IS OLC OLC_A OLC_B	Rule 2
		SC	Rule 5
		MRST PLD	Rule 4
		TC	Rule 7
	ES	BDY BI CTL TI CML IS OLC OLC_A OLC_B	Rule 2
		TCK	Rule 6
		MRST PLD	Rule 4
	PC	BDY BI CTL CML IS OLC OLC_A OLC_B	Rule 2
		SC	Rule 5
		TC	Rule 7
		MRST PLD	Rule 4

Pin Category	Pin Function	Allowable Combinations	Rule
Global (continued)	SC	BDY BI CTL CML TI IS OLC OLC_A OLC_B	Rule 2
		AC BC EC PC	Rule 5
		MRST PLD	Rule 4
		TC	Rule 7
	TCK	BDY BI CTL IS TI ES	Rule 2
		PC	Rule 6
		MRST PLD	Rule 4
		TC	Rule 7
	PR	BDY BI CTL IS	Rule 2
	PS	BDY BI CTL IS OLC OLC_A OLC_B	Rule 2
	PV	BDY BI CTL IS	Rule 2
	NIC	BDY BI CTL IS	Rule 2
	OI	BDY BI CTL IS	Rule 2
	OSC	BDY BI CTL IS	Rule 2
	TI	BDY BI CTL TDO SO ES SC	Rule 2
		TMS TRST TCK	Rule 6
	TMS	BDY BI CTL	Rule 2
		TC TI IS	Rule 6
	TRST	BDY BI CTL IS	Rule 2
		TI	Rule 6
	IS	CI ME SE/SG SI SIG WS TDI SOG SO TDO SOF MRD MO BDY BI CTL LH TC AC AS BC BS EC ES PC SC TCK PR PS PV NIC OI OSC TMS TRST MRST MRE PLD PGE CMI CME CMLE CML	Rule 2

Table 2-9 MISR Reset Test Function Pin Combinations

Pin Category	Pin Function	Allowable Combinations	Rule
MISR RESET	MRST	BDY BI CTL	Rule 2
		AC AS BC BS EC ES PC SC TCK PLD CML CML_A CML_B IS	Rule 4
	MRE	CI ME SE/SG SI SIG WS TDI SOG SO TDO SOF MRD MO TC IS	Rule 3
		BDY BI CTL	Rule 2
		TC	Rule 7

Table 2-10 PRPG LOAD Test Function Pin Combinations

Pin Category	Pin Function	Allowable Combinations	Rule
PRPG LOAD	PLD	BDY BI CTL	Rule 2
		AC AS BC BS EC ES PC SC TCK MRST IS	Rule 4
	PGE	CI ME SE/SG SI SIG WS TDI SOG SO TDO SOF MRD MO LH TC IS	Rule 2
		BDY BI CTL	Rule 2

Table 2-11 Channel Mask Test Function Pin Combinations

Pin Category	Pin Function	Allowable Combinations	Rule
Channel Mask	CMI	SI BDY IS MO OLI	Rule 3
	CME	BDY TC IS	Rule 3
	CMLE	SE/SG BDY TC IS OLE	Rule 3
	CML	AC AS BC BS EC ES PC SC MRST IS BDY	Rule 3
	CML_A	AC AS BDY TC MRST IS	Rule 3
	CML_B	BC BS BDY TC MRST IS	Rule 3

Table 2-12 OPCG Test Function Pin Combinations

Pin Category	Pin Function	Allowable Combinations	Rule
OPCG	GO	SG/SE	Rule 3
	OLI	SI CMI	Rule 3
	OLE	SG/SE TC CMLE	Rule 3
	OLC	TC AC AS BC BS EC ES PC SC PS	Rule 3
	OLC_A	TC AC AS BC BS EC ES PC SC PS	Rule 3
	OLC_B	TC AC AS BC BS EC ES PC SC PS	Rule 3

Rules for Determining the Valid Combinations

Note: SG (Scan Gate) is treated as a synonym for SE (Scan Enable).

- *Rule 1*

  At most one attribute from any category. See *Rule 5* for exceptions.

- *Rule 2*

  A test function from the manufacturing category can be specified on the same pins with those in the scan-in, scan-out, non-scan, global, misr observe, misr reset, and prpg load categories.

- *Rule 3*

  Multiple non-manufacturing attributes may be specified on a pin, provided that the attributes are MRE, PGE, or belong to one of the following categories: scan-in, scan-out, non-scan, and misr observe. Note that *Rule 1* still applies. For example, you cannot select an SE from the scan-in category and select a different attribute from the scan-out category.

- *Rule 4*

  MRST may be specified on the same pin with AC, AS, BC, BS, EC, ES, PC, PS, SC, or TCK provided that a corresponding MRE attribute is specified on some other pin and a corresponding PGE attribute is also specified on some other pin. MRST may be specified on the same pin with PLD provided that a corresponding MRE attribute is specified on some other pin and a corresponding PGE attribute is also specified on some other pin. MRST may not be specified on the same pin with any other attribute except AC, AS, BC, BS, EC, ES, PC, PLD, PS, SC, or TCK.

PLD may be specified on the same pin with AC, AS, BC, BS, EC, ES, PC, PS, SC, or TCK provided that a corresponding PGE attribute is specified on some other pin. PLD may not be specified on the same pin with any other attribute except AC, AS, BC, BS, EC, ES, MRST, PC. PS, SC, or TCK.

- *Rule 5*

  Within the global category, SC can be specified on a pin along with AC, BC, EC, or PC provided that the polarities on the two attributes agree. For example, the AS attribute means the clock is both an A Shift Clock (AC) and a system clock (SC). In effect, -AS is a shorthand for -AC,-SC. Thus, the following combinations of attributes within the global category are allowed on a pin, even though in each case the same result can be achieved with a single attribute:

  - -AC and -SC
  - +AC and +SC
  - -BC and -SC
  - +BC and +SC
  - -EC and -SC
  - +EC and +SC
  - -PC and -SC
  - +PC and +SC

- *Rule 6*

  If you have SO or TDO on a bi-directional pin (also known as inout or common I/O), you need to have a SE to inhibit the input while the output is being used for the scan out. The value on this SE is the non-controlling value of the dotted net that is fed by the bi-directional pin. If you don't specify this SE, Encounter Test will automatically insert it for you.

  If you have 1149.1 (TDI, TDO, TCK, TRST, TMS) attributes Encounter Test will automatically generate a secondary test function. Refer to "Secondary Test Function Attributes for 1149.1 Stored Pattern Test Generation" on page 50 for additional information.

- *Rule 7*

  `TC` can be specified on a shift clock. The polarities can be the same or different. When the polarities differ, the shift clock is being held active in the test generation state; in this case, there will normally be logic controlled by other `TC` pins or clocks which prevent the active state of the shift clock from propagating to latch or flip flop clock inputs.

Mode Initialization Sequences (Advanced)

The risks of producing bad test data are minimized when the product operates synchronously, under the control of one or more specific clock signals. Encounter Test exercises proper control of the clock signals to avoid races.

When the product has more than one mode of operation, the static signals that control or select the operational mode have to be controlled carefully to avoid unpredictable behavior. The control signals may be required to be set only during certain times, such as when a scan operation is being applied; but sometimes control signals (TEST_INHIBITs) are used to deactivate a function that is inherently asynchronous or not tractable for automatic test generation. In the latter case, the control signals may have to be held fixed throughout all testing operations in a given mode. Such static control signals may be accessible as primary inputs or they may be implemented as fixed-value latches to avoid allocating precious package pins for this purpose.

Yet another consideration in ensuring valid test patterns is the need to avoid three-state contention--the application of conflicting signals to multi-source nets where the conflict could damage the product. This is commonly the case when multiple CMOS drivers are connected to a bus; damage may occur if two or more drivers are simultaneously given control of the bus.

Requirements of Initialization Sequences

The foregoing considerations lead to the requirement for an initializing sequence (sequence `type=modeinit`) for each test mode, which:

- Initializes any fixed value latches. It may also set an initial state for other latches such as PRPG, MISR, or Floating Latches.
- Leaves all clock signals at their stability values.
- Establishes the necessary state of any primary inputs required to be held at fixed values during test generation. These are the Test Inhibit (TI) and Test Constraint (TC) pins.
- Leaves the design in a quiescent state, the Test Constraint and Clocks Off state, that is devoid of three-state conflicts, and does all the foregoing without causing three-state contention along the way.

Refer to "Sequence Definition Application Objects" in the *Encounter Test: Reference: Test Pattern Data Format* for additional information.

Automatically Generated Initialization Sequence

In the absence of a custom (user specified) mode initialization sequence, Encounter Test will automatically generate one and attach it to each test vector. Before relying on Encounter Test to generate the modeinit sequence, however, you must first realize that there are certain circumstances that require an externally specified modeinit sequence. One example is any test protocol that requires a parent mode for latch initialization. This is generally the case for LBIST. Encounter Test cannot automatically decide when an externally specified mode initialization sequence is required; it is up to you to make that call, based on an understanding of your test requirements and the automatic generation process. Refer to "Coding an Externally Specified Initialization Sequence" on page 68 for additional information.

The automatically generated mode initialization sequence consists of a pattern which does the following:

1. Sets all three-state pins to high-Z unless all internal drivers for that I/O are found to be inhibited by either directly tying the driver's enable to ground or by applying just the TI test function pins (applied below).
2. Sets all Scan Enable (SE, SGI, SGO) pins to their scan state value, unless set to a value as defined below if it is also a TC test function pin. Just to be safe, if the scan enable is 3-state and the strong drivers are not all inhibited, the pin is set to Z as denoted in step 1.
3. Sets all TI and TC test function pins to their stability values.

1149.1 consideration

If this is an 1149.1 test mode then note the following:

- The 1149.1 compliance enable pins, if any, are included among the set of TI pins. This comes about as a natural consequence of their having been assigned the TI test function attribute.
- The TRST pin, if any, has the additional test function attribute of +TI assigned to it and will therefore be set to a logic 1 by this pattern. This is its inactive state.

4. Sets all clock test function pins to their stability (OFF) values.
5. Sets the TMS pin to logic 1 if the test mode is 1149.1.

Following the above pattern, if the test mode is 1149.1, then patterns are added to the modeinit sequence to bring the TAP controller to the Test-Logic-Reset state, either asynchronously, if there is a TRST pin present, or through synchronous pulsing of the TCK pin, if there is no TRST pin.

If this is an 1149.1 test mode with a `TAP_TG_STATE` and instruction(s) specified in the mode definition file, there are additional patterns included which will load the 1149.1 instruction register and then move the TAP finite state machine to the specified `TAP_TG_STATE`. If there are multiple instructions associated with multiple scan chains and scan sections, the modeinit sequence will load only the last instruction.

Coding an Externally Specified Initialization Sequence

There are testing scenarios for which the automatically generated mode initialization sequence is inadequate. One case in particular where this is true is when there are memory elements (RAM or latches) that must be initialized before beginning operations in a test mode, and then Encounter Test requires that the initialization sequence be given as an input file, called the Sequence Definition File, to *Build Test Mode*. Without your specifying the initialization sequence, Encounter Test has no way to know that some memory elements have to be pre-initialized (except for the pattern generator and signature registers used in LBIST).

Also, if there are pseudo primary inputs defined in this test mode with clock, TI, or TC attributes, then it is almost certain that a user-defined mode initialization sequence is required, for Encounter Test would not know how to get the required values on those internal nets that are represented by the pseudo PIs.

The initialization sequence is specified as a *Define_Sequence* block in TBDpatt format. For details on sequence definition, refer to "Sequence Definition Application Objects" in the *Encounter Test: Reference: Test Pattern Data Format*. You have complete flexibility in how the initialization is performed, but for convenience it is recommended you switch to a different test mode in which the latches are scannable whenever it is feasible to do so. If this is the case, you can follow these steps:

1. Create a "parent" test mode. This is the test mode in which you will perform scan operations to initialize the latches for the target test mode.
2. Create the `TBDpatt` file with the initialization sequence for the new test mode, specify the file name in your test mode definition, and run *Build Test Mode*. To create this TBDpatt file, use a text editor of your choice.

Before you create the initialization sequence, you must be familiar enough with the design to know which latches must be initialized, and to what values they must be set. If you did not create the design, you may need to contact the designer or look for some design documentation that will identify the LBIST pattern generator, signature register, fixed-value latches, and any other latches requiring special attention.

When you are ready to create the `TBDpatt` file, it may be helpful to first run the *Create Vector Correspondence* tool in the parent test mode. This produces a file, `TBDvect.`*`parentmodename`*, in the directory that contains the imported design. This file contains a

list of all the scannable latch names. Copy this list into the `TBDpatt` file you are creating, eliminating those latches that you do not need to initialize. You will then need to do further editing to specify the latch values and put it into correct `TBDpatt` syntax. The syntax and structure of an initialization sequence are illustrated in Figure 2-2. The example design has a total of 50 latches, including a four-bit PRPG, a four-bit MISR, and one master-slave latch pair. Only these nine latches needed to be mentioned in the initialization sequence for this case.

Figure 2-2 An Example Test Mode Initialization Sequence in TBDpatt Format

```
# Mode Initialization sequence for an LBIST test mode
 TBDpatt_Format (mode=node,model_entity_form=name);
 [Define_Sequence INIT (modeinit);
  [Pattern 1 (pattern_type = static);
   Event 1 Begin_Test_Mode: lssd;
   ]Pattern 1;
  [Pattern 2 (pattern_type = static);
   Event 1 Scan_Load:
   "ST100DH.0000001.slaveOutput"=1
   "PR100BH.0000001.slaveOutput"=1
   "PR100CL.0000001.slaveOutput"=1
   "PR100DP.0000001.slaveOutput"=1
   "PR100ET.0000001.slaveOutput"=1
   "MI100EH.0000001.slaveOutput"=0
   "MI100EL.0000001.slaveOutput"=0
   "MI100EP.0000001.slaveOutput"=0
   "MI100ET.0000001.slaveOutput"=0;
   ]Pattern 2;
  [Pattern 3 (pattern_type = static);
   Event 1 Stim_PI:
   "T1"=1
   "T2"=1;
   ]Pattern 3;
  ]Define_Sequence;
```

In Figure 2-2 on page 69, pattern 1 identifies the parent test mode (lssd) with an instruction to apply that test mode's initialization sequence. Pattern 2 contains a `Scan_Load` event that specifies the initial values of those latches that we care about. The `Scan_Load` event implicitly invokes the scan sequence of the current test mode, which is in this case the parent mode that was established by the preceding pattern. Following the execution of pattern two, the latches will have been initialized.

For convenience in coding mode initialization sequences, Encounter Test supports some useful attributes on the `Scan_Load` event. `default_value=0|1` causes all unspecified latches to be initialized to the specified state (0 or 1). `default_value=scan_0|scan_1` causes all unspecified latches to be initialized to the state that would result from setting a constant 0 or 1 on the scan data primary input while performing the scan operation.

The third pattern switches to the target test mode by switching those primary inputs whose stability values differ from their stability values in the parent mode.

Note that if there had been three-state primary I/O nets being switched at pattern three, it might have been necessary (and is generally recommended) to expand it into three patterns, corresponding to the first three steps of the automatically generated initialization sequence as described in “Automatically Generated Initialization Sequence” on page 67.

If pseudo primary inputs exist, then in addition to the necessary stimuli for the real primary inputs you must specify the values that result on the pseudo primary inputs. The pseudo primary input values may be thought of as “expects”, but Encounter Test will use the pseudo PI values as stimuli to the downstream logic. Pseudo PI values are specified in Stim_PPI, Stim_PPI_Clock, and Pulse_PPI events.

In some situations, it may be desired to have the tester check certain signal values during the mode initialization sequence. Encounter Test supports this, but only if the signal is captured inside the product (by clocking a latch) and a scan operation is used to unload the value. Use a `Scan_Unload` event to specify a latch value to be observed in this way. Direct measurements on primary outputs (`Measure_PO` events) are not supported inside the mode initialization sequence.

It is possible to define an initialization sequence that doesn't require switching to a parent test mode. When there is no parent mode, the first pattern of the mode initialization sequence should set all three-state primary input pins to high-Z. *Build Test Mode* will help you by automatically setting any pins unspecified on the first pattern to their stability values or high-Z for three-state pins that do not have stability values. Non-three-state pins that don not have stability values are left unspecified.

The initial stim to Z of all three-state PIs is a safety precaution to avoid any potential conflicts with values that may initially be found at the product drivers at power-on.

Custom Scan Sequences (Advanced)

It is likely that the design you are running through Encounter Test has scan capabilities - for example, those associated with Level Sensitive Scan Design (LSSD). If so, then it is necessary that Encounter Test understand how to scan chain test vectors through the design's scan chains. The stimulus that must be applied at the design's primary inputs in order to accomplish this scanning operation is known as a scan protocol, or *scanop*.

In most cases the scanop is automatically generated by Encounter Test from the test function attributes assigned to primary inputs. For an LSSD part, knowing the A_SHIFT_CLOCK, the B_SHIFT_CLOCK, the SCAN_IN and SCAN_OUT pins, and SCAN_ENABLE, CLOCK_ISOLATION, and B_SHIFT_SYSTEM_CLOCK pins is sufficient to derive the scanop. It may be, however, that your design requires a scan protocol sufficiently out of the ordinary that it cannot be automatically derived. A case in point is a design whose scan chains are not all scannable in parallel, or a design in which the scan chain is gated by

non-scannable memory elements which must first achieve a particular state before scanning can proceed. In such situations Encounter Test cannot automatically derive the scan protocol, and instead you must define it to Encounter Test as a “custom scan sequence”.

It is necessary to define a custom scan sequence if any of the following is true:

- There is no single primary input stimulus vector that will enable all scan data paths and all scan clocks simultaneously.
- The clock sequence necessary to shift data one bit forward along the scan chain is something other than pulsing in order all A_SHIFT_CLOCKs, then all E_SHIFT_CLOCKs, then all B_SHIFT_CLOCKs.
- The scan operation requires values on pseudo primary inputs.

Refer to “System and Scan Clocks” on page 34 for details. and “Defining a Custom Scan Sequence” on page 71 for additional information.

Once you have determined that it is necessary to define a custom scan sequence for your design then you must go through the following steps:

1. Break down the scan protocol (scanop) into the set of constituent stimulus sequences recognized by Encounter Test. This is fully explained in the next section.
2. Create a file containing the scanop definition in TBDpatt (TBDseqPatt) format and feed this file into the Build Test Mode process of Encounter Test by specifying the Sequence Definition Path and Sequence Definition File Name.

   For additional information, refer to:

   - “TBDpatt and TBDseqPatt Format” in the *Encounter Test: Reference: Test Pattern Data Format*.
   - “Building a Test Mode” on page 19

From this point on there is no need for further user intervention on behalf of the custom scan sequence.

Defining a Custom Scan Sequence

The fundamental way to develop a `scanop` is by first considering the sequence of primary input events necessary to get to the scan state - i.e., the design state where scanning can take place from scan-in to scan-out - and then the sequence of clock pulses necessary to cause a one bit shift along the scan chain. The first of these sequences is called the Scan Preconditioning Sequence (`scanprecond`) and the second is called the Scan Sequence (`scansequence`). The `scanop` is thus seen to exist in its most elementary form as a

`scanprecond` sequence followed by a `scansequence`. (When the `scanop` is executed the `scansequence` must of course be repeated the number of times necessary to completely scan data through the scan chain.) We must go beyond this simple form, however, in considering parts where not all latches can be scanned in parallel. When this is the case then the `scanop` is broken into a sequence of “scan sections”, each of which scans one or more scan chains that can be scanned in parallel. Each scan section has its own scan preconditioning sequence as well as its own scan sequence. The `scanop` therefore takes the following general form if there are multiple scan sections:

```
scanop
   scansection 1
      scanprecond for scan section 1
      scansequence for scan section 1
   scansection 2
      scanprecond for scan section 2
      scansequence for scan section 2
          .
          .
          .
   scansection n
      scanprecond for scan section n
      scansequence for scan section n
```

Pin identification requirements

Any pin stimulated (other than to Z) or observed by a custom scan sequence must have a test function pin attribute assigned to it. In particular:

- Any pin pulsed in a custom scan sequence must have a clock test function pin attribute of the proper polarity. Refer to “Clock Functions” on page 34.
- Any pin used as a scan-in by the custom scan sequence must have either an SI or TDI (1149.1) test function pin attribute.
- Any pin used as a scan-out by the custom scan sequence must have either an SO or TDO (1149.1) test function pin attribute.
- Any pin used as a clock isolate pin in a custom scan sequence must have the Clock_Isolate (CI) test function pin attribute.
- Any pin that performs a scan gating function must have some test function attribute, at least a BDY.

A custom scan sequence may, in general, manipulate pins quite freely, subject to the pin identification requirements stated above. There are however the following restrictions:

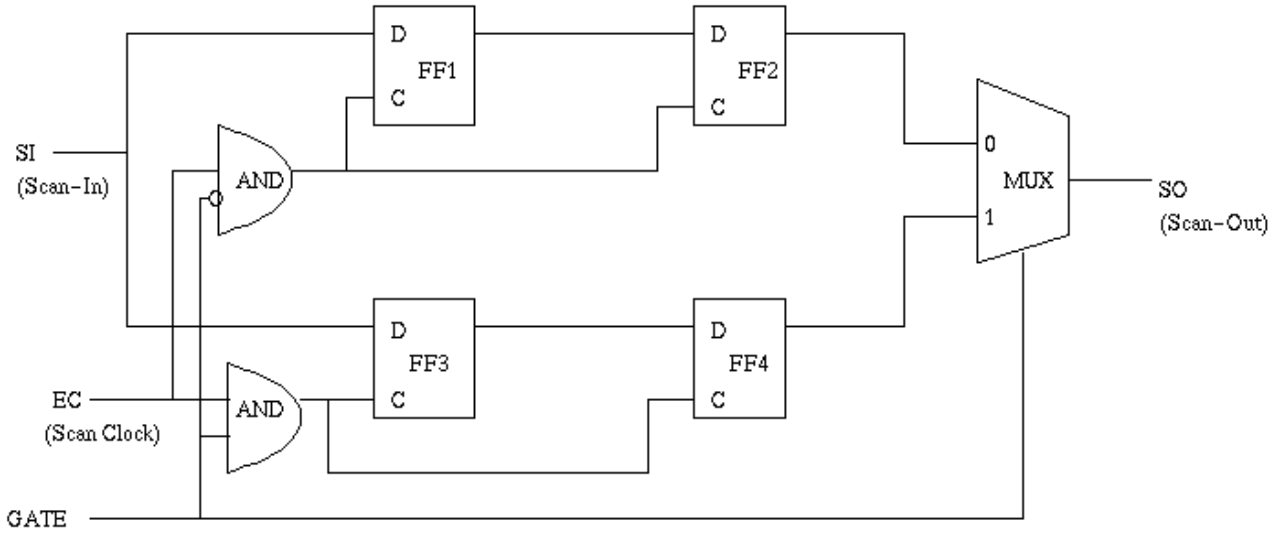
- No TEST_INHIBIT (TI) primary input is allowed to violate its attributed value.

- All OUTPUT_INHIBIT (OI) pins must be at their inhibiting values throughout the scan protocol.
- The final design state resulting from application of the custom scan sequence must not violate the Appendix E, "Test Constraint and Clocks Off" state. This is the state in which stored pattern test generation is performed.
- Any scan data output (SO) pin that is a bi-directional (inout) pin sourced by a three-state device should be stimulated to Z in the scan preconditioning or scanentry sequence.
- Multiple scan sections are not supported for WRPT or LBIST
- Verify Core Isolation and Create Core Tests do not support custom scan sequences
- The *Apply* event is the only allowable event for a `scanop` scansection. Refer to "Apply" in the *Encounter Test: Reference: Test Pattern Data Formats* for related information.

Custom Scan Sequence Example 1 - Multiple Scan Sections

The simple design structure depicted in Figure 2-3 requires that a custom scan sequence be defined because it is not possible to do a parallel scan of all its scannable flip-flops.

Figure 2-3 Design that Requires Multiple Scan Sections



The custom scan sequence (`scanop`), in TBDseqPatt syntax, takes the following form:

```
TBDpatt_Format (mode=node, model_entity_form=name);
[ Define_Sequence Scan_Preconditioning_Sequence_1 1 (scanprecond);
  [ Pattern 1.1 (pattern_type = static);
```

```
     Event 1.1.1   Stim_PI ():
                   "GATE"=0;
  ] Pattern 1.1;
] Define_Sequence Scan_Preconditioning_Sequence_1 1;

[ Define_Sequence Scan_Preconditioning_Sequence_2 2 (scanprecond);
  [ Pattern 2.1 (pattern_type = static);
     Event 2.1.1   Stim_PI ():
                   "GATE"=1;
  ] Pattern 2.1;
] Define_Sequence Scan_Preconditioning_Sequence_2 2;

[ Define_Sequence Scan_Sequence_1 3 (scansequence);
  [ Pattern 3.1 (pattern_type = static);
     Event 3.1.1   Measure_Scan_Data ():
                   "SO" ;
     Event 3.2.1   Set_Scan_Data ():
                   "SI" ;
     Event 3.3.1   Pulse ():
                   "EC"=+ ;
  ] Pattern 3.1;
] Define_Sequence Scan_Sequence_1 3;

[ Define_Sequence Scan_Sequence_2 4 (scansequence);
  [ Pattern 3.1 (pattern_type = static);
     Event 3.1.1   Measure_Scan_Data ():
                   "SO" ;
     Event 3.2.1   Set_Scan_Data ():
                   "SI" ;
     Event 3.3.1   Pulse ():
                   "EC"=+ ;
  ] Pattern 3.1;
] Define_Sequence Scan_Sequence_2 4;

[ Define_Sequence Scan_Section_1 5 (scansection);
  [ Pattern 4.1 (pattern_type = static);
     Event 4.1.1   Apply (): Scan_Preconditioning_Sequence_1;
     Event 4.1.2   Apply (): Scan_Sequence_1;
  ] Pattern 4.1;
] Define_Sequence Scan_Section_1 5;

[ Define_Sequence Scan_Section_2 6 (scansection);
  [ Pattern 4.1 (pattern_type = static);
     Event 4.1.1   Apply (): Scan_Preconditioning_Sequence_2;
     Event 4.1.2   Apply (): Scan_Sequence_2;
  ] Pattern 4.1;
] Define_Sequence Scan_Section_2 6;

[ Define_Sequence Scan_Operation_Sequence 7 (scanop);
  [ Pattern 5.1 (pattern_type = static);
     Event 5.1.1   Apply (): Scan_Section_1;
     Event 5.1.2   Apply (): Scan_Section_2;
  ] Pattern 5.1;
] Define_Sequence Scan_Operation_Sequence 7;
```

Skewed Load and Unload in a Custom Scan Sequence

It may be that the design for which you are writing a custom scan sequence contains shift register latches (SRLs) clocked by A_Shift_Clocks and B_Shift_Clocks. When this is true then it is possible to define a skewed load and skewed unload for any scan chain containing only SRLs. A skewed load (`skewload`) sequence pulses the A_Shift_Clock one final time after the N repetitions of the `scansequence` required to load the scan chain. This makes it possible for the L1 and L2 latches of an SRL to each have different logic values after the scan chain loading operation, and can lead to higher stuck-fault coverage for certain logic configurations. It is also useful for dynamic tests. A skewed unload (`skewunload`) sequence pulses the B_Shift_Clock one time before beginning the N repetitions of the `scansequence` required to unload the scan chain. A skewed unload sequence is used whenever the results of the test for some targeted fault have been captured in the L1 latch of an SRL.

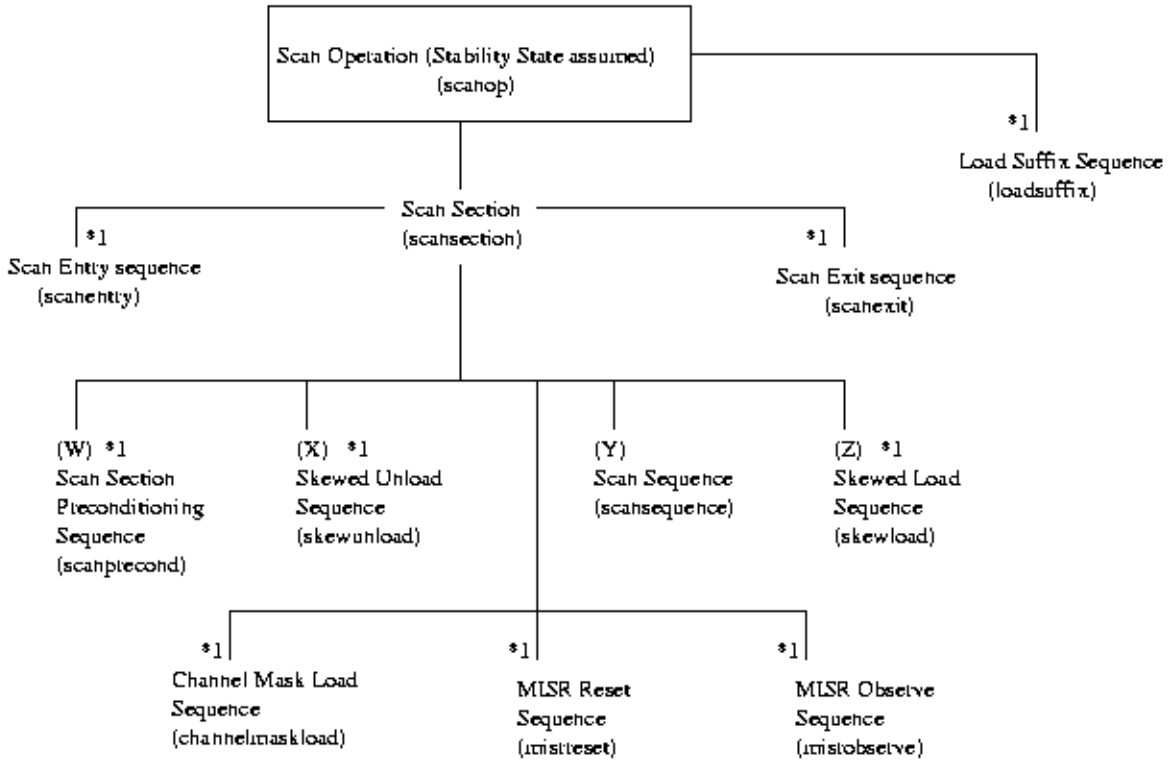
The Load Suffix Sequence

After the `scanop` has been executed to load scannable memory elements it may be desired to apply a load suffix (`loadsuffix`) sequence to load alternate stim latches (ASLs). The simplest example of an alternate stim latch (ASL) is the L3 latch of an LSSD Stable SRL (SSRL). A P_Shift_Clock is pulsed to move data from the L1 latch of the SSRL to the associated L3 latch of the SSRL, after the `scanop` has been executed. This pulsing of the P_Shift_Clock constitutes the `loadsuffix` sequence.

Basic Scanop Structure

Based on the discussion of the previous sections we see that the basic `scanop` with `loadsuffix` has the following structure:

Figure 2-4 Basic Scanop Structure



NOTE: The *1 denotes the sequence is optional.

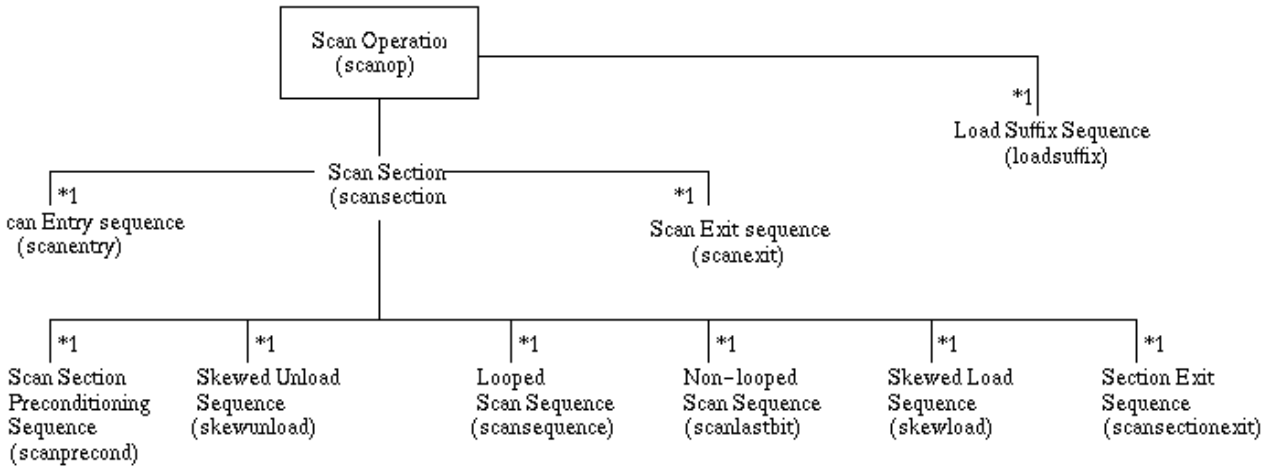
Advanced Scanop Structure

There are complex scan requirements for which the relatively simple `scanop` description of the previous section is inadequate. A good example of this is a design that implements the 1149.1 Boundary standard. To adequately describe the `scanop` for an 1149.1 design it is necessary to introduce the following additional components:

- Scan Entry Sequence (`scanentry`)
- Scan Exit Sequence (`scanexit`)
- Non-Looped Scan Sequence (`scanlastbit`)
- Section Exit Sequence (`scansectionexit`)

We will demonstrate by way of an example how each of these additional sequence types enters into the picture. But first, here is the overall `scanop` structure:

Figure 2-5 Advanced Scanop Structure



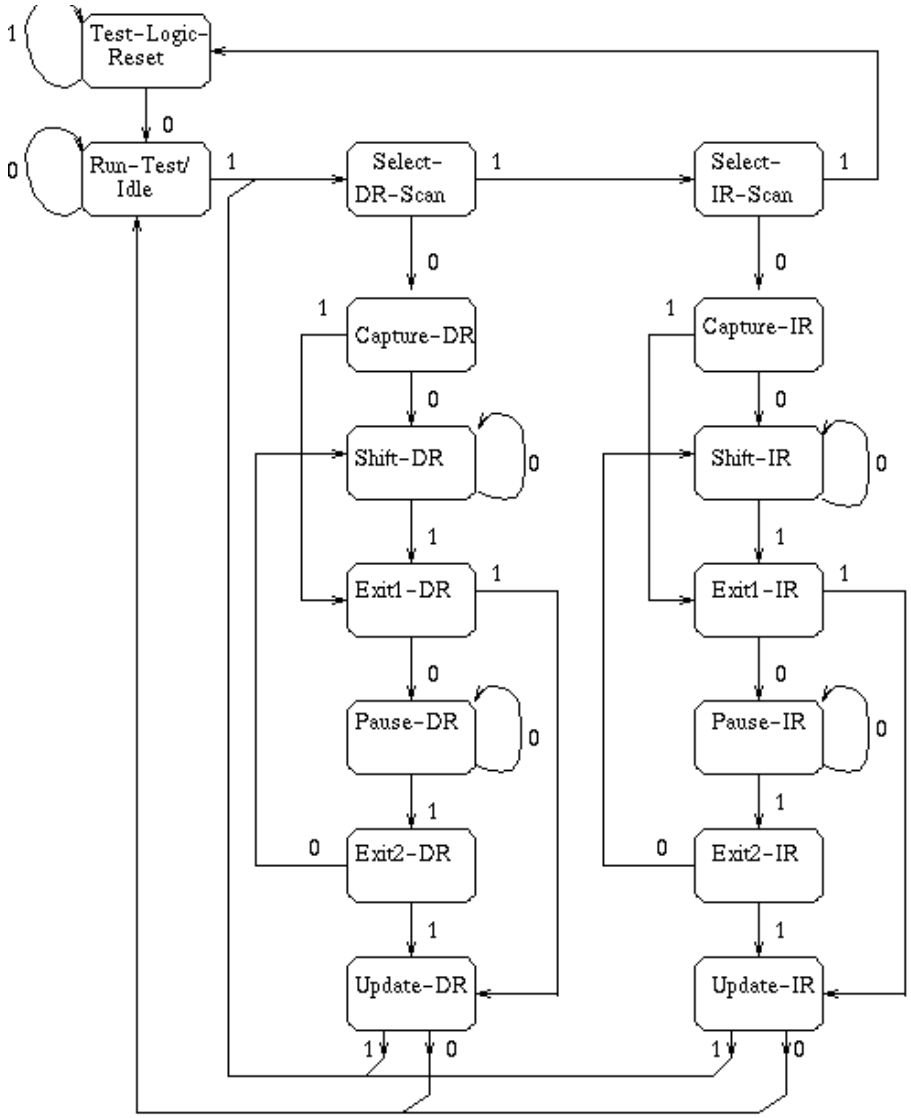
NOTE: The *1 denotes the sequence is optional.

The following section will show by way of an example the need for the additional sequence types of `scanentry`, `scanexit`, `scanlastbit` and `scansectionexit`.

Custom Scan Sequence Example 2 - 1149.1

The example about to be presented requires an understanding of 1149.1 TAP Controller state transitions, shown in Figure Figure 2-6 on page 78.

Figure 2-6 TAP Controller State Diagram



This figure is a representation of the state diagram for the 1149.1 TAP controller contained in IEEE standard 1149.1-1990, IEEE Standard Test Access Port and Boundary Scan Architecture.

The TAP controller is a synchronous finite state machine that responds to changes at the TMS and TCK inputs to the TAP to control the operation of the circuitry defined by the 1149.1 standard.

NOTES:

1. The value shown adjacent to each state transition in this figure represents the signal present on TMS at the time of a rising edge of TCK

2. All state transitions of the TAP controller occur based on the value of TMS at the time of a rising edge of TCK.

Encounter Test supports the generation of stored pattern tests in the Test-Logic-Reset (TLR) state of the TAP Controller, with all scanning operations done exclusively through the TAP port. The TLR state corresponds to the *Test Constraint and Clocks Off* state, which is the generic state in which test generation is always performed. For the purposes of this example it is assumed that the 1149.1 design has all scannable system memory elements linked into one scan chain extending from TDI to TDO (of course some instruction must already have been loaded into the Instruction Register (IR) in order to configure the memory elements into this one long scan chain.) In order to scan data through scannable memory elements via the TAP port it is necessary to traverse the TAP Controller state diagram from the TLR state to the Shift-DR (SDR) state and from there come back to the TLR state in order to resume test generation.

scanentry

For this simple example there is only one scan section, but in the most general case there could be several scan sections, in which case it would be desirable to have all scan sections start from compatible initial states. Run-Test/Idle (RTI) and Update-DR are compatible states, in the sense that they both transition directly to the Select-DR-Scan state when TMS is held at logic 1 and TCK is pulsed. So we therefore define a `scanentry` sequence which pulses TCK once while holding TMS at a logic 1, thus moving from the TLR state to the RTI state.

scanprecond

The `scanprecond` sequence starts from the RTI state and proceeds to the Shift-IR state to load the user-specified instruction. This is necessary because the TLR state, in which test generation is done, forces the IR to either the `BYPASS` or the `ID_CODE` instruction. It is the user-specified instruction that configures scan chains so that the following `scansequence` will operate correctly, so it must be restored at this time. After loading the instruction we then proceed to the Shift-DR state. The TMS pin is left at zero (0). Refer to Figure 2-7 on page 81 for an example.

scansequence

The `scansequence` consists of nothing more than pulsing TCK while TMS is held at logic 0. The `scansequence` is repeated as many times as necessary to scan data through the scan chain except for the last bit. The reason for stopping short of scanning the last bit is that we must eventually get back to the TLR state, and there is no way to do this without traversing the Exit1-DR state. Going from the Shift-DR state to the Exit1-DR state causes one final shift to occur, hence the name `scanlastbit`. Refer to Figure 2-8 on page 82 for an example.

scanlastbit

The *scanlastbit* sequence scans only the last bit of the TDR. It consists of holding TMS at logic 1 while pulsing TCK one time. The TAP is left in the Exit1-DR state upon exit from this sequence. Refer to Figure 2-8 on page 82 for an example.

scansectionexit

Recalling our earlier statement that we want all scan sections to start from compatible states, we define the *scansectionexit* sequence to take us from Exit1-DR to Update-DR, a state which is compatible with Run-Test/Idle. In this way, no matter how many scan sections there might happen to be, as long as they each end at Update-DR then we could conceivably execute them in any order. Refer to Figure 2-9 on page 82 for an example.

Note: The arbitrary ordering of scan sections is not presently supported. However, the `scanop`, including `scansectionexit`, is constructed in such a way as to forestall migration problems if arbitrary scan section ordering is supported in the future.

scanexit

Once all scan sections have been executed it is necessary to return to the state in which test generation is being conducted, the Test-Logic-Reset state. Refer to Figure 2-9 on page 82 for an example. If the latch is clocked in `scanexit`, the message TSV-315 is generated. If you want to manipulate the scan clock in `scanexit`, the clock path to scanable latch should be gated.

misrobserve

This sequence establishes the MISR Observe state by setting the `MISR_READ` pins to their value. A `Measure_MISR_Data` event is then included to observe the MISR contents on the `MISR_OBSERVE` pins. Another Stim_PI is then applied if needed to reset the values on the `MISR_READ` pins back to the scan state. Refer to Figure 2-10 on page 82 for an example.

misrreset

This sequence immediately follows the `MISR_Observe_Sequence`. It establishes the MISR Reset state by setting the `MISR_RESET_ENABLE` pins to their values, then pulses the MISR Reset clock and pulses all the B clocks. Another Stim_PI is then applied if needed to reset the values on the `MISR_RESET_ENABLE` pins back to the scan state. Refer to Figure 2-10 on page 82 for an example.

channelmaskload

This sequence applies the `channelmaskprecon`, `channelmaskcycle` and `channelmaskexit` sequences in order. Refer to Figure 2-10 on page 82 for an example.

Figure 2-7 Example scanprecond sequence for TAP scan SPTG, TAP_TG_STATE=TLR

```
    This example assumes the IR must be loaded with
    instruction=1011.  Note the inclusion of the
    instruction value to keep the sequence name unique.

[ Define_Sequence Scan_Preconditioning_Sequence1011 1 (scanprecond);
   [ Pattern 1.1 (pattern_type = static);
     Event 1.1.1 Stim_PI (): "TMS"=1;
     Event 1.1.2 Pulse (): "TCK"=+; # Enter Select-DR-Scan state
   ] Pattern 1.1;
   [ Pattern 1.2 (pattern_type = static);
     Event 1.2.1 Pulse (): "TCK"=+; # Enter Select-IR state
   ] Pattern 1.2;
   [ Pattern 1.3 (pattern_type = static);
     Event 1.3.1 Stim_PI (): "TMS"=0;
     Event 1.3.2 Pulse (): "TCK"=+; # Enter Capture-IR state
   ] Pattern 1.3;
   [ Pattern 1.4 (pattern_type = static);
     Event 1.4.1 Pulse (): "TCK"=+; # Enter Shift-IR state
   ] Pattern 1.4;
   [ Pattern 1.5 (pattern_type = static);
     Event 1.5.1 Stim_PI (): "TDI"=1;
     Event 1.5.2 Pulse (): "TCK"=+; # scan in right-most IR bit
   ] Pattern 1.5;
   [ Pattern 1.6 (pattern_type = static);
     Event 1.6.1 Stim_PI (): "TDI"=1;
     Event 1.6.2 Pulse (): "TCK"=+; # scan in next IR bit
   ] Pattern 1.6;
   [ Pattern 1.7 (pattern_type = static);
     Event 1.7.1 Stim_PI (): "TDI"=0;
     Event 1.7.2 Pulse (): "TCK"=+; # scan in next IR bit
   ] Pattern 1.7;
   [ Pattern 1.8 (pattern_type = static);
     Event 1.8.1 Stim_PI (): "TDI"=1
                             "TMS"=1;   # move to exit1-IR and
     Event 1.8.2 Pulse (): "TCK"=+; # scan in left-most IR bit
   ] Pattern 1.8;
   [ Pattern 1.9 (pattern_type = static);
     Event 1.9.1 Pulse (): "TCK"=+; # Enter Update-IR state
   ] Pattern 1.9;
   [ Pattern 1.10 (pattern_type = static);
     Event 1.10.1 Pulse (): "TCK"=+; # Enter Select-DR-Scan state
   ] Pattern 1.10;
   [ Pattern 1.11 (pattern_type = static);
     Event 1.11.1 Stim_PI (): "TMS"=0;
     Event 1.11.2 Pulse (): "TCK"=+; # Enter Capture-DR state
   ] Pattern 1.11;
   [ Pattern 1.12 (pattern_type = static);
     Event 1.12.1 Pulse (): "TCK"=+; # Enter Shift-DR state
   ]  Pattern 1.12;
 ] Define_Sequence Scan_Preconditioning_Sequence1011 1;
```

Figure 2-8 Example scansequence and scanlastbit sequences, TAP_TG_STATE=TLR

```
[ Define_Sequence Scan_Sequence 2 (scansequence, repeat=n-1);  # n is TDR length
   [ Pattern 2.1 (pattern_type = static);
     Event 2.1.1 Measure_Scan_Data (): "SO"
     Event 2.2.2 Set_Scan_Data (): "SI"
     Event 2.3.3 Pulse (): "TCK"=+; # shift TDR 1 bit
   ] Pattern 2.1;
] Define_Sequence Scan_Sequence 2;

[ Define_Sequence Scan_Last_Bit 3 (scanlastbit);    # shift nth bit
   [ Pattern 3.1 (pattern_type = static);
     Event 3.1.1 Measure_Scan_Data (): "SO"
     Event 3.1.2 Set_Scan_Data (): "SI"
     Event 3.1.3 Stim_PI (): "TMS"=1; # move to Exit1-DR
     Event 3.1.4 Pulse (): "TCK"=+; # and shift last bit
   ] Pattern 3.1;
] Define_Sequence Scan_Last_Bit 3;
```

Figure 2-9 Example scansectionexit and scanexit sequences, TAP_TG_STATE=TLR

```
[ Define_Sequence Scan_Section_Exit_Sequence 4 (scansectionexit);
   [ Pattern 4.1 (pattern_type = static);
     # TMS=1 already.   Next TCK pulse moves to Update-DR state and updates
    Event 4.1.2 Pulse (): "TCK"=+; # parallel (L3-like) outputs
   ] Pattern 4.1;
] Define_Sequence Scan_Section_Exit_Sequence 4;


[ Define_Sequence Scan_Exit_Sequence 5 (scanexit);
   [ Pattern 5.1 (pattern_type = static);
   Event 5.1.1 Stim_PI (): "TMS"=0;
   Event 5.1.2 Pulse ():  "TCK"=+; # Enter Run-Test/Idle state
   ] Pattern 5.1;
] Define_Sequence Scan_Exit_Sequence 5;
```

Figure 2-10 Example misrobserve, misrreset and channelmaskload Sequence

```
[ Define_Sequence MISR_Observe_Sequence  4 (misrobserve);
 [ Pattern 4.1 (pattern_type = static);
    Event 4.1.1   Measure_MISR_Data ():
"out[0]"
"out[10]"
"out[11]"
"out[12]"
"out[13]"
"out[14]"
"out[15]"
"out[1]"
```

```
"out[2]"
"out[3]"
"out[4]"
"out[5]"
"out[6]"
"out[7]"
"out[8]"
"out[9]" ;
 ] Pattern 4.1;
[ Define_Sequence MISR_Reset_Sequence  5 (misrreset);
 [ Pattern 5.1 (pattern_type = static);
    Event 5.1.1   Stim_PI ():
.....................................1...;
 ] Pattern 5.1;
 [ Pattern 5.2 (pattern_type = static);
    Event 5.2.1   Pulse ():
"mrc"=+ ;
 ] Pattern 5.2;
[ Define_Sequence ChannelMaskLoad_Sequence  7 (channelmaskload);
 [ Pattern 7.1 (pattern_type = static);
    Event 7.1.1   Apply (): ChannelMask_Cycle_Sequence;
 ] Pattern 7.1;
] Define_Sequence ChannelMaskLoad_Sequence 7;
[ Define_Sequence Scan_Section_Sequence  8 (scansection);
 [ Pattern 8.1 (pattern_type = static);
    Event 8.1.1   Apply (): Scan_Preconditioning_Sequence;
    Event 8.1.2   Apply (): ChannelMaskLoad_Sequence;
    Event 8.1.3   Apply (): Scan_Sequence;
    Event 8.1.4   Apply (): MISR_Observe_Sequence;
    Event 8.1.5   Apply (): MISR_Reset_Sequence;
 ] Pattern 8.1;
```

Restrictions for Custom Scan Sequence

- Multiple scan sections are not supported for WRP or LBIST.
- Neither Verify Core Isolation nor Create Core Tests support custom scan sequences.
- There is no support for a scan configuration in which the same latch occurs in more than one scan section.
- The creation of dynamic tests by targeting static faults is not supported.

- Skewed load and unload sequences are not supported for multiple scan sections.

Automatically Generated Initialization Sequence

For non-1149.1 test modes, the automatically generated scan preconditioning (`scanprecond`) sequence consists of a single pattern which does the following:

1. Sets all scan enable (SE, SGI, SGO) test function pins to their scan state value.
2. Sets to Z any 3-state bi-directional pins whose internal drivers are not disabled.

For 1149.1 test modes, a more complicated scan sequence is generated with multiple patterns as needed to bring the TAP state machine from the TAP_TG_STATE to the correct state for scanning (usually the Shift_DR state). If there are multiple 1149.1 instructions and scan chains to be shifted, there is a separate scan section defined for each instruction. When there are multiple scan sections, there is a `scanentry` sequence created to move the TAP finite state machine from the TAP_TG_STATE to a common state for all scan sections to start and end in.

For non-1149.1 test modes, the automatically generated `scansequence` sequence, which is intended to define the complete sequence of clocks needed to shift 1 full bit and which should be able to be completed within one tester cycle, consists of a single pattern which does the following:

1. Indicates that all scan-out pins should be measured.
2. Indicates that all scan-in pins should have their new scan data applied
3. Pulses all defined shift A clocks. If there are multiple shift A clocks and they are numbered, separate Pulse() events are generated to ensure the clocks appear in their numerical order.
4. Pulses all defined shift E clocks. If there are multiple shift E clocks and they are numbered, separate Pulse() events are generated to ensure the clocks appear in their numerical order.
5. Pulses all defined shift B clocks. If there are multiple shift B clocks and they are numbered, separate Pulse() events are generated to ensure the clocks appear in their numerical order.

For 1149.1 test modes, TCK is considered a shift E clock and additional scan clocks may also be defined. Also, when the TAP_TG_STATE is no the Shift_DR state, there is a `scanlastbit` sequence generated which shifts the last bit if the scan chains and moves the TAP finite state machine into the Exit1_DR state.

For non-1149.1 test modes, the automatically generated `scanexit` sequence, which is intended to return the design from the scan state back to the TG state, consists of a single pattern which does the following:

- Sets all Test Constraint (TC) test function pins to their TC value.

Note that scan enable pins are left at their scan state value unless the pin is also specified to be a Test Constraint (TC).

For 1149.1 test modes, the automatically generated `scanexit` sequence moves the TAP finite state machine from the Exit1_DR state (where the `scanlastbit` sequence ended) to the TAP_TG_STATE.

For LSSD test modes in which shift B clocks are defined, an automatically generated `skewedunload` sequence is created which pulses all the shift B clocks (in their proper order if there are multiple numbered shift B clocks).

For LSSD test modes in which shift A clocks are defined, an automatically generated `skewedload` sequence is created which pulses all the shift A clocks (in their proper order if there are multiple numbered shift A clocks).

The automatically generated `scansection` sequence does the following:

1. Applies the `scanprecond` sequence for this scan section.
2. Applies the `skewedunload` sequence (if any) for this scan section.
3. Applies the `scansequence` sequence for this scan section.
4. Applies the `skewedload` sequence (if any) for this scan section.

The automatically generated `scanop` sequence does the following:

1. Applies the `scanentry` sequence (if any) for 1149.1 test modes with multiple scan sections.
2. Applies the `scansection` sequence(s) for all defined scan sections in the order they appear in the mode definition file.

Applies the `scanexit` sequence (if any) for 1149.1 test modes to get to the TAP_TG_STATE or for non-1149.1 test modes to get to the TG state when there are Test Constraint (TC) test function pins defined.

3

Analyze Test Mode Results

Analyzing Build Testmode Log Results

The build_testmode log contains the following messages of varying levels of severity. Refer to the *Message Reference* or use the `msgHelp` command for extended message help.

Mode Init and Scan Sequences

In most cases, mode initialization and scan sequences will be created automatically, as indicated by the following messages:

INFO (TTM-391): A default modeinit sequence will be generated.

INFO (TTM-387): A default scanop sequence will be generated.

Use the following command to review the defined sequences for a specific test mode:

```
report_sequences workdir=<directory> testmode=<test mode name>
```

Active Logic

INFO (THM-814): Testmode contains 99.96% active logic, 0.04% inactive logic and 0.00% constraint logic.

Check the percentage of active logic. Active logic is the logic that can be observed at a primary output or scannable flop/latch. Faults in active logic are targeted for ATPG. In fullscan test modes, the percentage of active logic should be very high. Some reasons for low active logic are:

- Incorrect test function pin assignments such as Test Inhibits (TIs)
- Dangling (untestable) logic included in some simulation libraries for timing

Constraint logic is not actually a part of the design. It is the logic that is added to the model when constraints are specified during build_model for ATPG. The constraints are modeled so that if any constraint is violated by a specific pattern, three-state contention occurs.

The following is an example of test function pin assignments listed in the log file:

```
Test Function Pin Information for Test Mode:   FULLSCAN
-----------------------------------------

3  SC    (System  Clock)             Pins
0  AC    (A Shift Clock)             Pins
0  BC    (B Shift Clock)             Pins
0  PC    (P Shift Clock)             Pins
2  EC    (E Shift Clock)             Pins

0  OSC   (Oscillator)                Pins
1  TI    (Test Inhibit)              Pins
3  SE    (Scan Enable)               Pins
0  CI    (Clock Isolation)           Pins
0  OI    (Output Inhibit)            Pins
14 SI    (Scan Input)                Pins
14 SO    (Scan Output)               Pins

Pin Index    Type Test Function                 Pin Name / Net Name
---------    ---- ----------------------------  -------------------
35           PI   +SC                           I_CORE_RESET1 / I_CORE_RESET1
37           PI   -EC  -SC                      I_CORE_SYS_CLK / I_CORE_SYS_CLK
53           PI   -EC  -SC                      TEST_CLOCK / TEST_CLOCK
54           PI   +TI                           TEST_ENABLE / TEST_ENABLE
0            PI   -SE                           COMPACTOR_SCOMP / COMPACTOR_SCO
1            PI   -SE                           COMPACTOR_SPREAD / COMPACTOR_SP
52           PI   +SE                           SCAN_ENABLE / SCAN_ENABLE
38           PI    SI                           SCANIN[0] / SCANIN[0]
39           PI    SI                           SCANIN[10] / SCANIN[10]
```

The following is an example of fault status information:

```
                        Testmode Statistics: FULLSCAN

                    #Faults  #Tested  #Possibly  #Redund  #Untested  #PTB
Total                151793   0        0          0        151793     21
Total Static         61029    0        0          0        61029      21
Total Dynamic        90764    0        0          0        90764      0
Collapsed            112223   0        0          0        112223     19
Collapsed Static     43675    0        0          0        43675      19
Collapsed Dynamic    68548    0        0          0        68548      0
Pin                  143695   0        0          0        143695     21
Pin Static           53775    0        0          0        53775      21
Pin Dynamic          89920    0        0          0        89920      0
Collapsed Pin        104125   0        0          0        104125     19
Collapsed Pin Static 36421    0        0          0        36421      19
Collapsed Pin Dynamic 67704   0        0          0        67704      0
PI                   217      0        0          0        217        2
PI Static            109      0        0          0        109        2
PI Dynamic           108      0        0          0        108        0
PO                   368      0        0          0        368        0
```

```
PO Static               184      0        0          0        184         0
PO Dynamic              184      0        0          0        184         0
Pattern                 8098     0        0          0        8098        0
Pattern Static          7254     0        0          0        7254        0
Pattern Dynamic         844      0        0          0        844         0
Shorted Net             0
Drvr/Rcvr               61029    0                            61029
Path                    0        0                            0
```

A test mode can be built before or after the fault model. If the fault model is built first, the `build_testmode` log contains a list of the fault status for the test mode. The number of faults in a test mode is determined by the amount of active logic. The important thing here is to note the number of faults in the test mode.

You can generate the same report with current fault status at any time using the command report_fault_statistics. More information about the fault categories may be found in the section Reporit Faults n *Encounter Test: Guide 4: Faults* .

Tip

You can build all the test modes simultaneously to save wall time. Refer to "Building Multiple Test Modes" on page 99 for more information.

Faults Not Active in any Testmode

The `build_testmode` log also lists the faults that are defined in the fault model but are not active in any test mode, and the maximum attainable global test coverage due to the inactive faults. The following is an example of inactive fault information:

```
    Statistics for faults that are not active in any test modes and maximum test
       coverage attainable with the current set of test modes:

Definitions:
#Faults  : Number of Faults defined (independent of test modes).
#Active  : Number of Faults Active in at least one mode.
#Inactive: Number of Faults Inactive in all modes.
%Active  : #Active/#Faults = Maximum Test Coverage attainable.

There are 3 test mode(s) defined:

     FULLSCAN

                          #Faults   #Active    #Inactive %Active
Total                      69162     69117      45        99.93
Total Static               69162     69117      45        99.93
Total Dynamic              0
Collapsed                  51808     51763      45        99.91
Pin                        53804     53775      29        99.95
Collapsed Pin              36450     36421      29        99.92
PI                         110       109        1         99.09
```

```
PO                          184       184       0         100.00
Pattern                     15358     15342     16        99.90
Shorted Net                 0
Drvr/Rcvr                    7168      0         7168       0.00
```

Troubleshooting Build Test Mode Problems

Refer to TTM - Test Mode Messages in the *Message Reference* or execute the following command for extended message help:

```
msgHelp   <message number>
```

For example:

```
msgHelp  TTM-031
```

The following table lists some common problems during build_testmode and how to troubleshoot those problems:

Table 3-1 Syntax or Content Problems

Problem	Possible Cause/Solution
TTM030	Pinname <pinname> not found on design.
	The pin name specified does not exist in the design. View the pin in the schematic viewer. Use a part of the name and the filter option if you cannot determine the correct name.
TTM031	Netname <netname> not found on design.
	The net name specified does not exist in the design. View the net in the schematic viewer. Use a part of the name and the filter option if you cannot determine the correct name.
TTM048	Illegal or missing <statement type> statement terminator.
	All statements must end with a semicolon.
TTM054	Missing <flag type> flag polarity value.
	The specified test function pin/net assignment needs a polarity (+/-).
TTM055	Invalid <flag type> flag polarity value.
	The polarity given for the specified test function flag is incorrect. Generally this means a flag not requiring a polarity has been assigned a polarity.

Problem	Possible Cause/Solution
TTM056	Conflicting <flag type> flag polarity value.
	The specified test function pin/net assignment was entered twice, each with a different polarity.
TTM059	Unable to open MODEDEF assignfile=<file name>.
	Typo in the name of the assignfile or its directory.
TTM451	Pin name <pinid> is a correlated pin and is not allowed to have test function flags. The test function flags will be ignored.
	Correlated pins cannot have test function flags. If you are correlating a pin or set of pins, only the pin to which you are correlating them can have a test function flag.
TTM458	Net name <net name> is not found in the hierModel. This Assign statement will be ignored.
	The net name specified in the assign statement does not exist. View the net in the schematic viewer using a part of the name and the filter option if you cannot determine the correct name.
TTM459	Pin name, <pin name>, is not found in the hierModel. This Assign statement will be ignored.
	The pin name specified in the assign statement does not exist. View the pin in the schematic viewer using a part of the name and the filter option if you cannot determine the correct name.

Table 3-2 Sequence Definition Problems

Problem	Possible Cause/Solution
TTM035	Sequence Definition file does not contain a modeinit sequence.

Problem	Possible Cause/Solution

	If a sequence file is supplied, it must contain a modeinit sequence. If you have supplied special scan and clocking sequences, then you must provide the set of stimulus to put the design in the initial state required to be able to use those sequences. The modeinit sequence is identified in the seqdef file with a statement similar to the following:
	`[Define_Sequence INIT 1 (modeinit);`
	`<set of patterns and events to define the required stimulus>`
	`]Define_Sequence 1;`

Table 3-3 Other Error Indications

Problem	Possible Cause/Solution
TTM347	There is less than 96 percent active logic in this test mode.
	The smaller the percentage of active logic, the more is its impact on global test coverage. This percentage represents the amount of logic of the design that is testable in the testmode. Therefore, if it is 93% and you achieve test coverage of 100% in the testmode then your global coverage would be 93%.
TTM801	Test mode has NOT been defined - see preceding message(s) for details.
	Terminating errors existed that did not allow completion of build testmode.

Unit or Zero Delay Simulations for Test Mode

When using a custom mode initialization sequence, there might be situations where the type of simulator being used affects the final simulation results. The default for building a testmode is to use a unit delay simulation. Some designs might require zero delay simulation because of unbalanced clock lines. The `build_testmode` command has the option `delaymode=zero` to invoke a zero delay simulation.

For unit-delay simulation to work, you have to account for the flip-flops, latches, and RAMs in the model, taking clock distribution unit-delay skews into consideration. If the number of gate delays are not well balanced (a difference of 7 or more gates), you might have to modify the

model to make unit delay simulations work. For example, consider a case in which a single clock feeds two flops A and B, where flop A feeds flop B. If flop B has 10 extra buffers on the clock line, flop A will evaluated and its value will be propagated to flop B. In this case, flop B would capture the input data to flop A. Under normal situation, flop B would have expected to capture the original value in flop A instead of its input value. This is an example of how an unbalanced clock path can cause incorrect simulation results. This situation can be resolved by inserting more gates on the clock line to flop A to make the number of delays more equivalent. In addition to balancing the clock distribution unit delays, you can also insert gates (buffers) into the data inputs or outputs of all latch flip-flops or RAM models. This delays the propagation of new data signals feeding to opposite phase latches until their clocks can turn off.

Zero delay simulation predicts the clock always wins the clock/data race. That is, for clock gating or data hold time races, zero-delay simulation makes calculations assuming that the clock goes off before new data arrives.

When running zero delay simulation in NC-sim, you might need to add the keyword `ncelab -seq_udp_delay 10ps` to add some unit of delay for latch and flip-flop calculations.

Reporting Test Mode Information

The report_test_structures command prints the following information about a test mode:

- The list of test function pins
- Information about scan chains
- Lists of flops/latches with special characteristics (such as fixed value, inactive, and floating)
- Channel masking logic for compression modes
- On Product Clock cutpoints and pseudo-primary inputs
- Embedded pipeline structures for compression modes

Basic Command Syntax for report_test_structures

The basic command syntax for `report_test_structures` is as follows:

```
report_test_structures workdir=<directory> testmode=<modename>
```

where:

- `workdir` is the name of the working directory

- `testmode` is the name for the testmode being created (in some cases, the name of the mode definition file is used, for example, FULLSCAN)

Most of the test mode reports are self-explanatory. Some of the information in the scan chain listing is explained below:

Scan Chain Information Report

```
Controllable/Observable Scan Chain Information for Test Mode:  FULLSCAN
----------------------------------------------------------
    16 Controllable Scan Chains
    16 Observable Scan Chains


  Controllable Scan Chain 1
   Load Pin Index / Pin Name      43 / DLX_CHIPTOP_DATA[0]
   Bit Length                     84
   Scan Section Sequence      Scan_Section_Sequence
Observable Register 1
   Unload Pin Index / Pin Name    50 / DLX_CHIPTOP_DATA[16]
   Bit Length                     84   Unload pin in phase with Load Pin
   Scan Section Sequence      Scan_Section_Sequence
Not proven flushable

Position  LTYPE   Blk Index/IO  ObserveReg  ObserveBit  Block Name  Clock
Affiliation
-------- -------  ------------  ----------- ----------  ------------  -----------
---------
1        rDFF_cS  13334++       1           84          DLX_CORE...  DLX_CHIPTOP_
SYS_CLK:-EC inPhase DLX_CHIPTOP_SYS_CLK: -SC inPhase
2        rDFF_cS  13364++       1           83          DLX_CORE...  DLX_CHIPTOP_
SYS_CLK: -EC inPhase DLX_CHIPTOP_SYS_CLK: -SC inPhase
3        rDFF_cS  13394++       1           82          DLX_CORE...  DLX_CHIPTOP_
SYS_CLK: -EC inPhase DLX_CHIPTOP_SYS_CLK: -SC inPhase
....
```

The preceding report shows that there are 16 controllable and 16 observable scan chains in the design. If a scan chain is both controllable and observable, it will have a unique control and observe register numbers. These numbers may not be the same.

Note the following in the report:

- The `Load Pin` and `Unload Pin` normally indicate scanin and scanout, respectively. In a compression or LBIST test mode, the load or unload pins may be internal to the design.
- `Bit Length` identifies the length of the chain.
- `Scan Section Sequence` applies if a custom scan sequence is defined where all scan chains do not shift in parallel.

- `Not Proven Flushable` - This applies to LSSD designs only. Encounter Test checks if it can hold the scan clocks active and flush the SI data to the SO pin.
- The `Position` column (Control bit positions) is numbered from the scan input to the scan output.
- The `LTYPE` column identifies the flop or latch type.
- All correlated flops/latches (share a common bit position) are listed together.
- The `Blk Index/IO` field contains the block index of the representative block, flop/latch, and its phase relationship with the scanin and scanout.

  Relationship to ScanIn is character under the "I":

  - `+` in phase with scanin and controllable
  - `!` in phase with scanin and NOT controllable (scan chain hole)
  - `-` out of phase with scanin and controllable
  - `~` out of phase with scanin and NOT controllable (scan chain hole)

  Relationship to ScanOut is second character under the "O":

  - `+` in phase with scanout and measurable
  - `!` in phase with scanout and NOT measurable (scan chain hole)
  - `-` out of phase with scanout and measurable
  - `~` out of phase with scanout and NOT measurable (scan chain hole)
  - `#` loaded by scan, NOT measurable, NOT in scanout path
- `Observe reg` and `Observe bit` indicate the observe register and the bit position for this flop/latch. Observe bit positions are numbered from the scan output to the scan input. Observe only registers are listed separately.
- `Block name` contains the block name for the flop/latch.
- `Clock Affiliation` contains the scan state clock affiliation data for the scan port of each reported latch. This information is displayed in the report only when you specify `reportclockaffiliation=scan` for the `report_test_structures` command.

Embedded Pipeline Report

In addition to the above mentioned information, `report_test_structure` also prints the embedded pipeline structures when the keyword `embeddedpipelines` is set to `yes`. An example of the report is shown below:

```
Embedded Pipelines for Test Mode:   FULLSCAN
---------------------------------
Embedded Input Pipeline Latch/Flops
-----------------------------------
  Blk Index   Latch/Flop Block Name
  ---------   ---------------------
         21   AinPipe1.f1.dff_primitive.slave
         51   AinPipe2.f1.dff_primitive.slave
         81   AinPipe3.f1.dff_primitive.slave
        111   AinPipe4.f1.dff_primitive.slave
        351   CinPipe1.f1.dff_primitive.slave
        381   CinPipe2.f1.dff_primitive.slave
        411   CinPipe3.f1.dff_primitive.slave
        441   CinPipe4.f1.dff_primitive.slave
        471   CinPipe5.f1.dff_primitive.slave
        501   DinPipe1.f1.dff_primitive.slave
        171   BinPipe2.f1.dff_primitive.slave
        201   BinPipe3.f1.dff_primitive.slave
        231   BinPipe4.f1.dff_primitive.slave
        141   BinPipe1.f1.dff_primitive.slave
        591   EinPipe2.f1.dff_primitive.slave
        621   EinPipe3.f1.dff_primitive.slave
        531   DinPipe2.f1.dff_primitive.slave
        561   DinPipe3.f1.dff_primitive.slave
        261   CMEpipein1.f1.dff_primitive.slave
        290   CMEpipein2.f1.dff_primitive.master
        291   CMEpipein2.f1.dff_primitive.slave
        320   CMEpipein3.f1.dff_primitive.master
        321   CMEpipein3.f1.dff_primitive.slave
   There are 18 Embedded Input Pipeline Latch/Flops.


Embedded Output Pipeline Latch/Flops
------------------------------------
  Blk Index   Latch/Flop Block Name
  ---------   ---------------------
        711   FoutPipe3.f1.dff_primitive.slave
        681   FoutPipe2.f1.dff_primitive.slave
        651   FoutPipe1.f1.dff_primitive.slave
        831   HoutPipe2.f1.dff_primitive.slave
        891   IoutPipe2.f1.dff_primitive.slave
        801   HoutPipe1.f1.dff_primitive.slave
        861   IoutPipe1.f1.dff_primitive.slave
        771   GoutPipe2.f1.dff_primitive.slave
        741   GoutPipe1.f1.dff_primitive.slave
        921   JoutPipe1.f1.dff_primitive.slave
        951   KoutPipe1.f1.dff_primitive.slave
There are 11 Embedded Output Pipeline Latch/Flops.


Embedded Channel Mask Enable Pipeline Latch/Flops
-------------------------------------------------
```

```
  Blk Index   Latch/Flop Block Name
  ---------   ---------------------
        951   KoutPipe1.f1.dff_primitive.slave
        951   KoutPipe1.f1.dff_primitive.slave
        951   KoutPipe1.f1.dff_primitive.slave
        951   KoutPipe1.f1.dff_primitive.slave
        951   KoutPipe1.f1.dff_primitive.slave
There are 5 Embedded Channel Mask Enable Pipeline Latch/Flops.

Report completed.
```

4

Multiple Test Modes

Building Multiple Test Modes

Encounter Test allows you to define up to 500 different test modes of operation for your design. You can define different test modes for many reasons. The following are examples of test modes that may be defined:

- Fullscan test mode
- XOR compression test mode
- Reduced pin count, boundary scan internal test mode
- Boundary scan external, interconnect test mode
- Different test modes may be needed if test data must be generated for two or more substantially different testers

The following is a sample command syntax to build multiple modes:

```
build_testmode workdir=<directory> testmode=<mode1,mode2,mode3,...> \
       [ assignfile=<assign1,assign2,assign3,...>]  \
        [ modedef=<def1,def2,def3,...> ] \
        [ seqdef=<seq1,seq2,seq3,...> ] ...
```

where:

- `workdir` is the name of the working directory.
- `testmode` is the list of test modes being created, comma separated, no spaces.
- `assignfile` is the list of assign files, comma separated, no spaces.
- `modedef` is the list of mode definition files, comma separated, no spaces.
- `seqdef` is the list of sequence files, comma separated, no spaces.

All other keywords remain the same. Test modes are built sequentially in the specified order.

Tip

You can execute multiple test modes simultaneously to save time but make sure to build the fault model after you have built all the test modes. If you build the fault model first and then run parallel build test mode, the jobs will conflict over the master fault list. This file conflict will significantly impact/reduce the wall time savings or require reprocessing of the jobs to ensure the fault files are correctly created.

Example: Building Fullscan and XOR Compression Modes

```
build_testmode workdir=/local/dlx testmode=FULLSCAN,COMPRESSION \
          assignfile=/local/dlx/FULLSCAN.assign,/local/dlx
          COMPRESSION.assign
```

Cross-Mode Markoff

A major benefit of using multiple test modes is cross-mode markoff. Cross-mode markoff is a technique in which faults tested in one test mode are marked as tested in another test mode so they are not retargeted for ATPG.

Enabling Cross-Mode Markoff

Cross-mode markoff is controlled by the COMET statement in the Mode Definition File. By default, all test modes that use the same TDR will use cross-mode markoff. You can further control the cross-mode markoff behavior of test modes. For more information, refer to “COMETS” on page 233.

Example: Using Cross Mode Markoff

The following example script runs ATPG on a compression test mode (such as XOR compression). If the test coverage is not as high as expected, top-off patterns are generated using the FULLSCAN test mode.

There is cross mode markoff because both the `COMPRESSION` and `FULLSCAN` modedef files reference the same TDR (`Tester_Description_Rule=dummy.tdr`)

```
build_model workdir=<directory> designsource=...

# both test modes must specify same TDR in modedef file
build_testmode workdir=<directory> testmode=COMPRESSION,FULLSCAN...
verify_test_structures workdir=<directory> testmode=FULLSCAN
verify_test_structures workdir=<directory> testmode=COMPRESSION

# make sure both modes can scan
```

```
create_scanchain_tests workdir=<directory> testmode=FULLSCAN
create_scanchain_tests workdir=<directory> testmode=COMPRESSION

# logic tests using compression mode
create_logic_tests workdir=<directory> testmode=COMPRESSION
             experiment=logic
commit_tests workdir=<directory> testmode=COMPRESSION experiment=logic

# top off with fullscan vectors
create_logic_tests workdir=<directory> testmode=FULLSCAN
             experiment=logic
commit_tests workdir=<directory> testmode=FULLSCAN experiment=logic

write_vectors workdir=<directory> testmode=COMPRESSION
write_vectors workdir=<directory> testmode=FULLSCAN
```

Building a Hierarchy of Test Modes (Parent Testmodes)

For the design that contain complex clocking schemes and/or a combination of test structures such as LBIST, MBIST, 1149.1, and FULLSCAN, it may be easier to build a base test mode and use that test mode as a starting point when building other test modes. The base testmode is referred as the parent testmode and each testmode that is created using that base test mode is referred as its child, thus creating a testmode hierarchy.

The creation of a test mode hierarchy requires the introduction of custom test sequences. A child test mode references its parent test mode within the custom test sequence. The child test mode must have the following within the test mode initialization section of its custom sequence:

```
[ Define_Sequence  Mode_Initialization_Sequence (modeinit, user_defined) ;
    [ Pattern  1 (pattern_type = static);
         Event  1.1 Begin_Test_Mode ():  <name of parent test mode>  ;
    ] Pattern 1 ;
......
```

5

Delete Test Mode Information

Deleting a Test Mode

This option removes all files and experiments associated with the test mode, as well as any test data created, including any experiments that have been created but not yet committed. The fault status is reset. Any test modes that used this mode as a parent are also removed.

The basic command syntax to delete a test mode is as follows:

```
delete_testmode workdir=<directory> testmode=<modename>
```

where:

- `workdir` is the name of the working directory
- `testmode` is the name of the testmode being deleted

To remove a test mode using the graphical interface, refer to "Delete Test Mode(s)" in the *Encounter Test: Reference: GUI*.

To remove a test mode using command lines, refer to "delete_testmode" in the *Encounter Test: Reference: Commands*.

Deleting Committed Data from a Test Mode

This option removes all committed test data and resets the fault status information for an existing test mode, without requiring that the test mode be re-created. This function saves time, in the event you wish to decommit experiments that were previously committed, in that the test mode is retained and does not require re-creation. Another time-saving characteristic of deleting committed data is the retention of existing results from Testability Analysis applications, thereby avoiding the need to re-run the application.

The default action of *Delete Committed Tests* is to remove all experiments dependent on the test mode. If you wish to retain existing uncommitted data, you may specify an uncommitted name to rename and re-register the committed vectors file as an uncommitted vectors file.

The basic syntax for removing committed data is as follows:

```
delete_committed_tests workdir=<directory> testmode=<modename>
```

where:

- `workdir` is the name of the working directory
- `testmode` is the name of the testmode being deleted

To perform *Delete Committed Tests* using the graphical interface, refer to "Delete Committed Tests" in the *Encounter Test: Reference: GUI*.

To perform *Delete Committed Tests* using command lines, refer to "delete_committed_tests" in the *Encounter Test: Reference: Commands*. Mode-dependent test pattern information and fault status information are removed.

Example: Deleting a single testmode

```
delete_testmode workdir=/local/dlx testmode=FULLSCAN
```

Example: Deleting multiple testmodes

```
delete_testmode workdir=/local/dlx testmode=FULLSCAN,COMPRESSION
```

A

Design Structures and Testmode Details

Delay Test Modes

Several mode definition files are provided with Encounter Test (in the *$Install_Dir*/defaults/rules/modedef directory) that enable either timed or untimed delay testing. You can use these test modes as is, unless you have specific tester information, or you need to customize the mode definition file for some other reason. Following are the installed mode definition files:

- FULLSCAN
- COMPRESSION
- OPMISR+

Note: Prior to Encounter Test 6.2, the mode definition file names were called *<testmode>*_DELAY and *<testmode>*_TIMED. After 6.2, the main testmode (Ex FULLSCAN) contained the information found in the FULLSCAN_TIMED mode definition.

Note: When running delay test, the model should be built using build_model truetime=yes. This keyword causes some additional checks to be performed specifically for delay testing. It also inserts buffers into the design as needed to facilitate the placement of delays. These buffers have no impact on the results of test generation.

Example A-1 Delay Test Mode using Standard Mode Definition

```
build_testmode workdir=<directory> testmode=FULLSCAN assignfile=..
```

Customizing a Mode Definition for Delay Test

The following are required for a delay test mode:

1. Mode Definition File:
   a. The test_types statement should include dynamic tests as follows:

```
test_types dynamic timed early (0,1,0) late (0,0,1) logic dynamic scan
shift_register
```

b. The faults statement should include dynamic faults as follows

```
faults static, dynamic;
```

c. The TDR statement must reference a TDR that supports timed tests:

```
Tester_Description_Rule = timed.tdr
```

2. Tester Description Rule

a. A `PIN_TIMING` statement must be specified that describes the capabilities of the target tester. The TDRs provided with Encounter Test will support many testers, but may not be optimal as in the following example.

```
PIN_TIMING
 TIMING_RESOURCE = SHARED_RESOURCE
  CLOCKS=5 NON_CLOCKS=13 PO_STROBES=7
MAX_PULSES = 1
MAX_STIMS = 1
MAX_MEASURES = 1
MAX_CYCLE_TIME = 1 MS
MIN_CYCLE_TIME = 2500 PS
...
```

Example A-2 Customize a Mode Definition File and TDR for Delay Test

3. Copy the standard mode definition file from $Install_Dir/rules/modedef and modify it.

4. Copy the standard TDR from $Install_Dir/rules/tdr and modify it (make sure to use the same name as specified in the mode definition file).

5. Run `build_testmode`:

```
build_testmode workdir=<directory> testmode=<modename>  \
modedef=<modified_modedef> modedefpath=<directory> \
tdrpath=<directory>
```

Assumed Scan Test Modes

Assumed scan chain test modes allow for the creation of test modes prior to the insertion of scan structures. This facilitates early assessment of design testability and fault coverage with reasonable accuracy. Encounter Test identifies existing functional latches/flip-flops and act as if a single scan chain exists comprised of all the latches and flip-flops.

Assumed scan chain test modes are identified by use of the `assumed` attribute on the SCAN statement in the mode definition file. See “SCAN” on page 197 for details.

Example A-3 Sample assumed_mode Definition File

```
Tester_Description_Rule = base_tester;
scan          type = assumed gsd
              boundary=no
```

```
              in = pi
              out = po;
test_types    static logic signatures no
              shift_register
              static macro;
faults        static;
test_function_pin_attributes K_FLAG , O_FLAG;
```

Example A-4 Example Command Invocation for an Assumed Scan Test Mode

```
build_testmode workdir=<directory> testmode=assumed \
modedef=assumed_mode modedefpath=<directory>
```

Note: When creating or modifying your own mode definition file, specify the containing directory using the `modedefpath` keyword.

Excluding Specific Flops from a Scan Chain

To remove specific flops/latches from consideration during creation of a scan chain for Assumed Scan test modes, create a file listing these latches and specify the file using the keyword `excludelatchfile` for `build_testmode`.

Example A-5 Exclude File Example

```
// Sample exclude file for Assumed Scan hierarchicalDelimiter=/
// . (dot) is always recognized as hierarchical delimiter
// so for this file . and / will be used as hierarchical delimiters
my2Alu_0.ADD1.co_reg my2Alu_0.ADD1.q_reg[0]
// my2Alu_0.ADD1.q_reg[1]/*
my2Alu_0.ADD1.q_reg[2] */
my2Alu_0/ADD1/q_reg[3]
my2Alu_0.ADD1.q_reg[4]
my2Alu_0.ADD1.q_reg[5]
my2Alu_0.ADD1.q_reg[6]
my2Alu_0.ADD1.q_reg[7]
// existing scan chain flop
my2Alu_0.MUL1.\q_reg[11]
// the hierarchical delimiter @ is not interpreted as such since / was given above
// therefore the block below will be read as the name of a block on the top
// level of the hierarchy my2Alu_0@MUL1@\q_reg[12]
```

Example A-6 Example Command Invocation using the excludelatchfile keyword

```
build_testmode workdir=<directory> testmode=assumed \
               modedef=assumed_mode modedefpath=<directory> \
               excludelatchfile=<directory>/exclude_flops
```

Exclude File Syntax

The following are the elements of an *exclude* file:

The `excludelatchfile` option is only recognized for Assumed Scan test modes and is ignored for all other test modes.

- Line comments

  Comments are denoted by lines starting with `//` or `/*`. Lines starting with either of these comment delimiters are ignored during processing. Block comments (comments spanning multiple lines) are not supported.

- Specification of the Hierarchical Delimiter

  Use the following format:

  `hierarchicalDelimiter=`*`hierarchical delimiter`*

  The default hierarchical delimiter used by Encounter Test is `.` (a dot). The following are accepted hierarchical delimiters::

  - `.` (dot)
  - `/` (forward slash)
  - `^` (caret)
  - `|` (vertical bar)
  - `@` (at-sign)

  Therefore if the specified hierarchical delimiter is `/` (forward slash), the statement would look like the following:

  ```
  hierarchicalDelimiter=/
  ```

  If no hierarchical delimiter is specified in the file or if an unsupported delimiter is specified, a `.` (dot) will be interpreted as the hierarchical delimiter.

- Specification of the flop/latch to be excluded in the form of a hierarchical flop block or latch primitive.

Assumed Scan Mode Limitations

These limitations apply to the use of assumed scan:

- Use of assumed scan is not supported for signature-based ATPG, including On-Product MISR support.

- Assumed scan requires the use of `BOUNDARY=NO` and excludes all signature-based testing parameters and statements in the mode definition file.
- Assumed scan chain test modes cannot be mixed in the same MARKOFF comet as non-assumed scan chain test modes.
- TSV checks associated with signature-based testing as well as checks, which verify scan operation, should not be used.
- LSSD Flush and Scan Chain tests cannot be generated.
- ATPG results for an assumed scan chain test mode cannot be included on a TMD.
- 1149.1 scan types are not supported as assumed.
- At least one primary input and output pin must exist in the design.

Pipelined Scan Test Modes

Encounter Test supports the creation and use of test modes that support two types of pipeline logic:

A Control Pipeline design (refer to Control Pipelines in the *Encounter Test: Reference: Legacy Functions Guide*) is one where a control signal feeds into a set of latches/flip-flops that are logically configured into a pipeline. Pipelined control signals are primary input signals which perform a specific function, such as a scan enable (`SE`) or test constraint (`TC`). The reason for pipelining certain control signals is to allow them to switch quickly in spite of the fact that they may fan out to a large number of nodes within the design and therefore may have trouble meeting tight timing specifications without the use of pipelining.

As an example, consider a muxed-scan design (defined as a general scan design - GSD) for which an at-speed test is desired and for which the tests will launch transitions via a final scan-shift cycle and capture the responses with a system cycle. In such cases, where the release and capture clocks are the same, and only the scan enable signal must switch between the two pulses, it is necessary to switch the scan enable signal as quickly as possible between the launch and capture events. Using normal design techniques in which the scan enable signal is fanned out to all muxed-scan flip-flops in the design, unless the scan enable is distributed much like a clock tree, it is very unlikely to be able to switch from the scan state to the non-scan state at anything close to the normal system cycle time. By inserting a single stage pipeline close to the ends of the fan-out tree for the scan enable signal, it will become possible to pulse the launch (scan) clock cycle and follow it with a capture (system) clock cycle with minimal delay between them. This can happen because the scan enable primary input pin can be switched out of scan state and the change can propagate down to the pipeline stage at a relatively slow pace. When the launch clock occurs, the scan chains shift

one bit and simultaneously, the scan enable pipeline stage is updated so that the next clock cycle will not be a scan cycle.

The second type of pipeline logic design is the Data Flow Pipeline (refer to Data Flow Pipelines in the *Encounter Test: Reference: Legacy Functions*). This design is one where non-scan latches/flip-flops can be initialized to a known value from a latch/flip-flop that is on a scan chain. This design approach assists ATPG because these non-scan latches/Flip-Flops are more easily controlled and reduces area and routing costs associated with placing latches on a scan chain.

Note: Data Flow Pipelines are also referred as Non-Scan Flush Pipelines because the user needs to submit a non-scan flush sequence as input into Encounter Test. We are aware that there are multiple types of Pipeline logic and that there is no consistent terminology used within the industry.

Control Signal and Data Flow Pipelines in the *Encounter Test: Reference: Legacy Functions* provides a more comprehensive explanation of each type of pipeline logic, examples, associated commands and the required input data.

OPMISR Test Modes

Encounter Test Synthesis supports Test Compression by means of an On-Product Multiple-Input Signature Register (OPMISR). A more advanced option of OPMISR technology called OPMISR+ is supported if the Encounter Test installation has an OPMISR+ option.

Introduction to OPMISR and OPMISR+ Compression

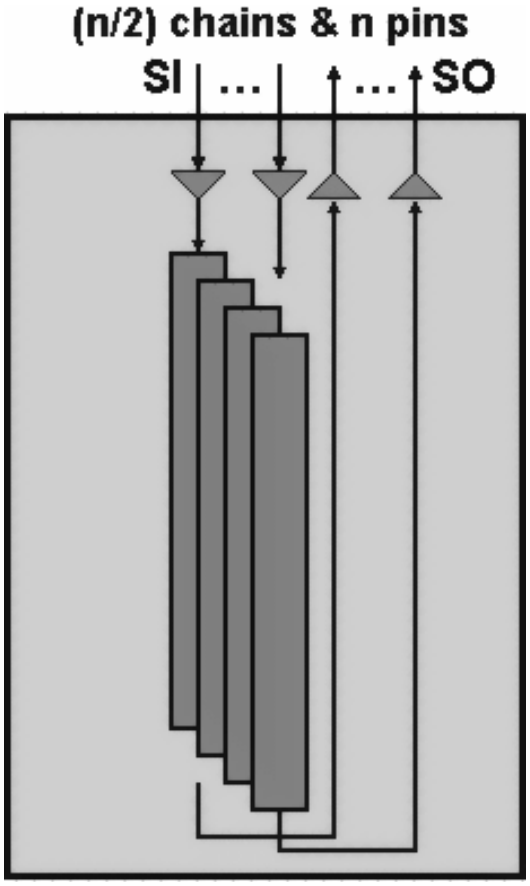
Using an On-Product MISR (scan-to-MISR) test mode can result in reduced test data volume. See "Reducing the Cost of Chip Test in Manufacturing" in the *Automatic Test Pattern Test Generation User Guide* for a detailed explanation.

It is recommended to first create a chip fullscan chain test mode for diagnostics, as specified in the DIAGNOSTIC_MODE statement, and then create an On-Product MISR test mode.

The MISR test structure modifies the typical fullscan scan chain such that each Scan Data Input internally drives many (~10-50) shorter chains. These shorter chains feed the inserted MISR structure. The chain's values are captured into the MISR during shift, generating a resulting signature that can be shifted out. Also, Scan Data ports, both Outputs and Inputs, can be configured to behave in unison to provide an additional 2X more compression.

Figure A-1 on page 112 shows a traditional scan configuration for Automatic Test Pattern Generation (ATPG) with N scan pins (N/2 scan-ins and N/2 scan-outs) and N/2 scan chains.

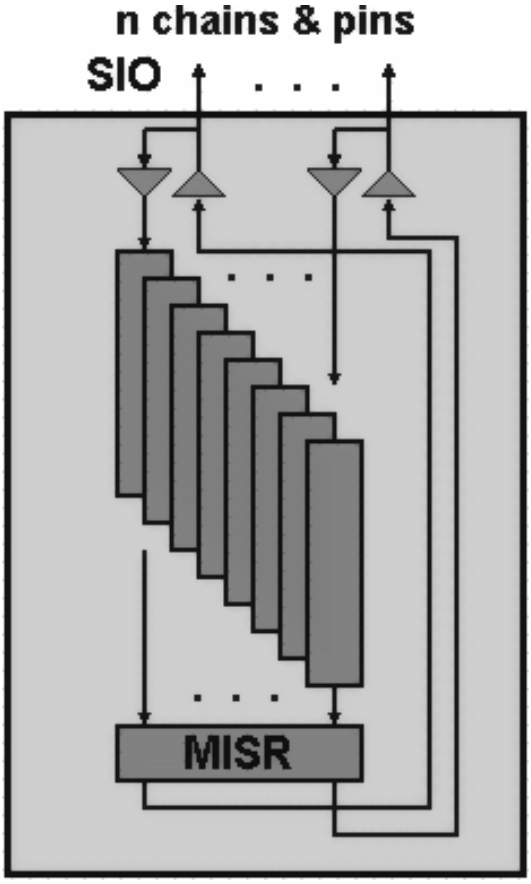
Figure A-1 Normal FullScan ATPG Configuration



During the scan operation phase of ATPG test vectors, the Automatic Test Equipment (ATE) will load test stimuli for the next test through the scan-in pins while concurrently scanning out the test results from the current test via the scan-out pins. The number of scan pins is very small (typically 16 to 128) relative to the number of scan elements (typically 100K in 0.25 micron technologies to 10M for 0.065 nanometermicron technologies). Since the number of scan chains is equal to the number of scan-in pins, the test time, which is dominated by the time it takes to scan into and out of the longest scan chain, will naturally increase along with the increase in the number of scan elements.

Figure A-2 on page 113 shows a typical OPMISR configuration that includes an embedded MISR to compress the test vector responses. This is achieved by using N scan chains and N bi-directional scan pins.

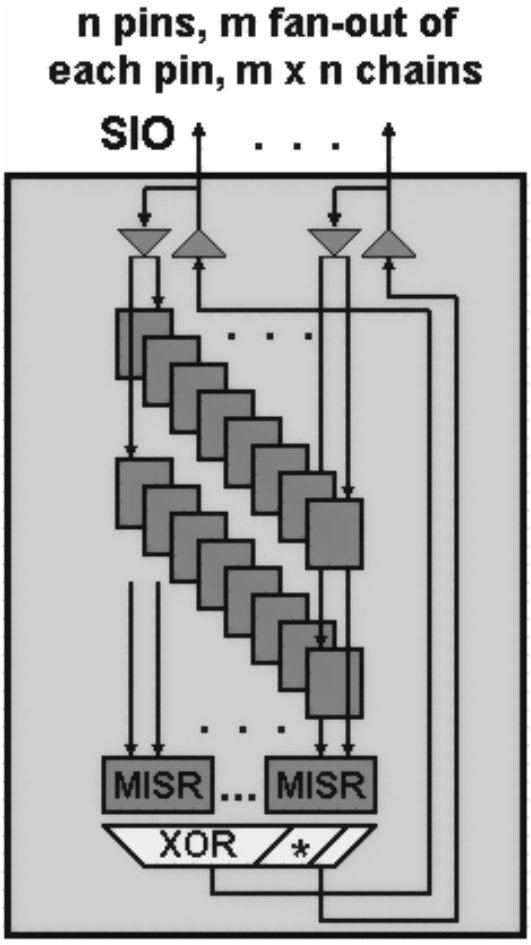
Figure A-2 OPMISR Configuration



The *N* scan bi-directional pins are used for scan-in during the scan operation and as MISR observe pins after the scan-out collects the response data into a MISR signature. By cutting the chain lengths in half, the test time is cut by half and tester storage by 2 to 3 times. The savings in test time are primarily due to the use of shorter scan chains while the savings in storage are primarily due to the fact that only the MISR response values have to be stored on the ATE instead of the values of all the scan elements. Although the MISR observe pins do not have to be shared with the scan-in pins, this sharing of test functions allows the use of reduced-pin count testers.

Further improvements can be achieved by using an enhancement called OPMISR+. Figure A-3 on page 114 shows an OPMISR+ implementation.

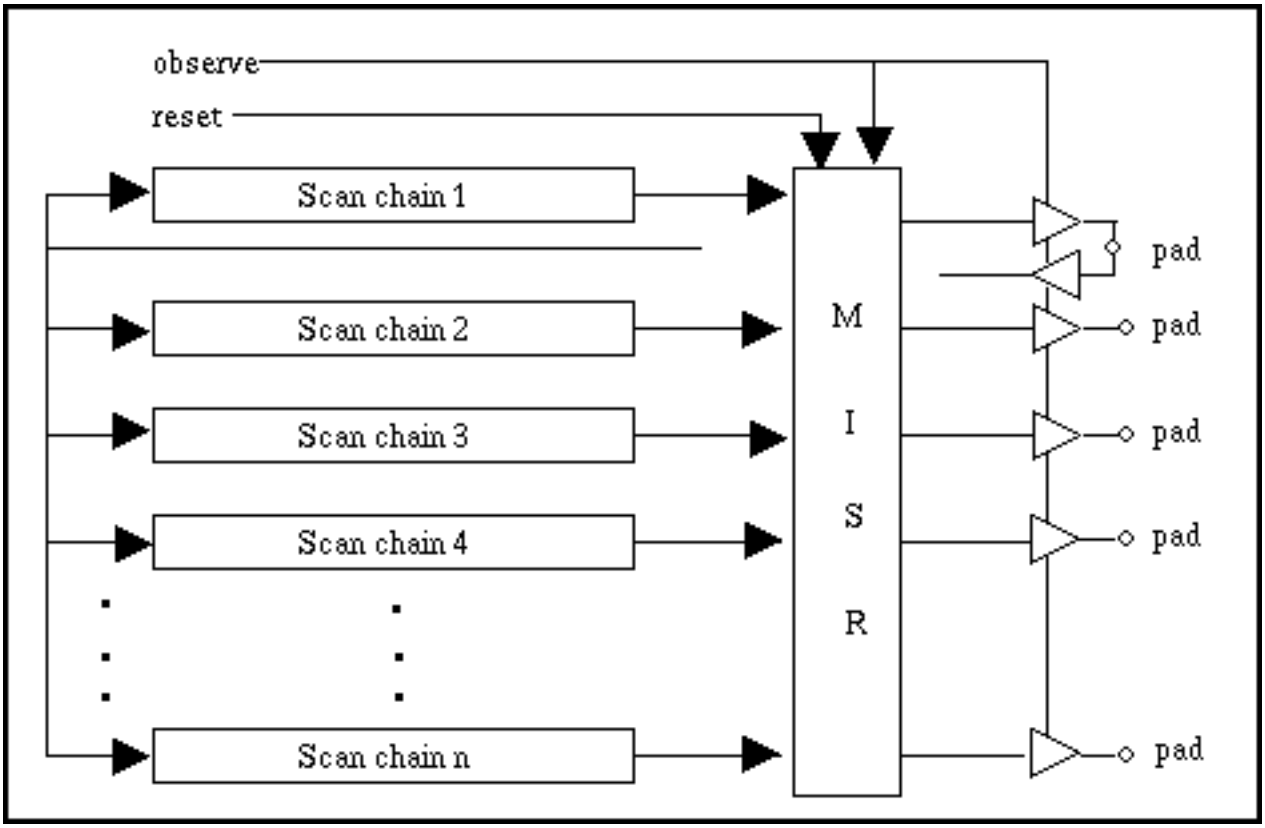
Figure A-3 OPMISR+ Configuration



The *N* bi-directional pins are fanned out to *NxM* scan chains by broadcasting each scan pin to *M* different scan chain inputs. The *NxM* wide MISR is composed of *M* MISRs, each of size *N*, and is observed at the scan pins via an XOR space compactor. Since this cuts the scan chain length by a factor of *2xM* (if the internal scan chains are reasonably well balanced), it reduces the test time by a factor of *2xM* and storage by *2xM* or *3xM*. Because the scan chains are channeled into a signature register it is customary to refer to them as channels instead of scan chains.

OPMISR+ allows scan-in pins to fan out to multiple scan chains. The combined use of OPMISR and fan out techniques produces shortened scan chain lengths, and thus shortened scan time, that can result in significant test time reductions. An example of OPMISR with fan out is depicted in Figure A-4 on page 115.

Figure A-4 OPMISR using Scan Fan Out



One limitation of embedded MISR based techniques is due to the presence of unknown states (also known as X-states) in the design. During the test generation process, it is possible that these X-states get captured in a scan flop and get scanned into the MISR. Once an unknown (X) value is captured in a MISR, that MISR's signature is corrupted and becomes unpredictable. Encounter Test has implemented a technique called Channel Masking which prevents this corruption of signatures. This technique involves the use of masking logic at the end of the channels that allows the ATPG to block the unknown (X) values from entering the MISR. The OPMISR methodology supports both a full-scan ATPG mode as well as an OPMISR mode. The full-scan mode is used to clean up faults that were not detected during the OPMISR test mode such as faults in the test logic and faults whose tests were invalidated by the X-states or masking. The full-scan mode may also be used for diagnostics.

The OPMISR+ methodology supports three test modes: OPMISR+, OPMISR, and full-scan ATPG. The OPMISR mode can be used to detect any faults that were not detected in the OPMISR+ mode due to correlations caused by the fan-out of the scan in pins to the channels, which adds additional constraints to the test generator. The full scan mode is primarily intended to be used for diagnostics and can also be used for applying any other kinds of tests

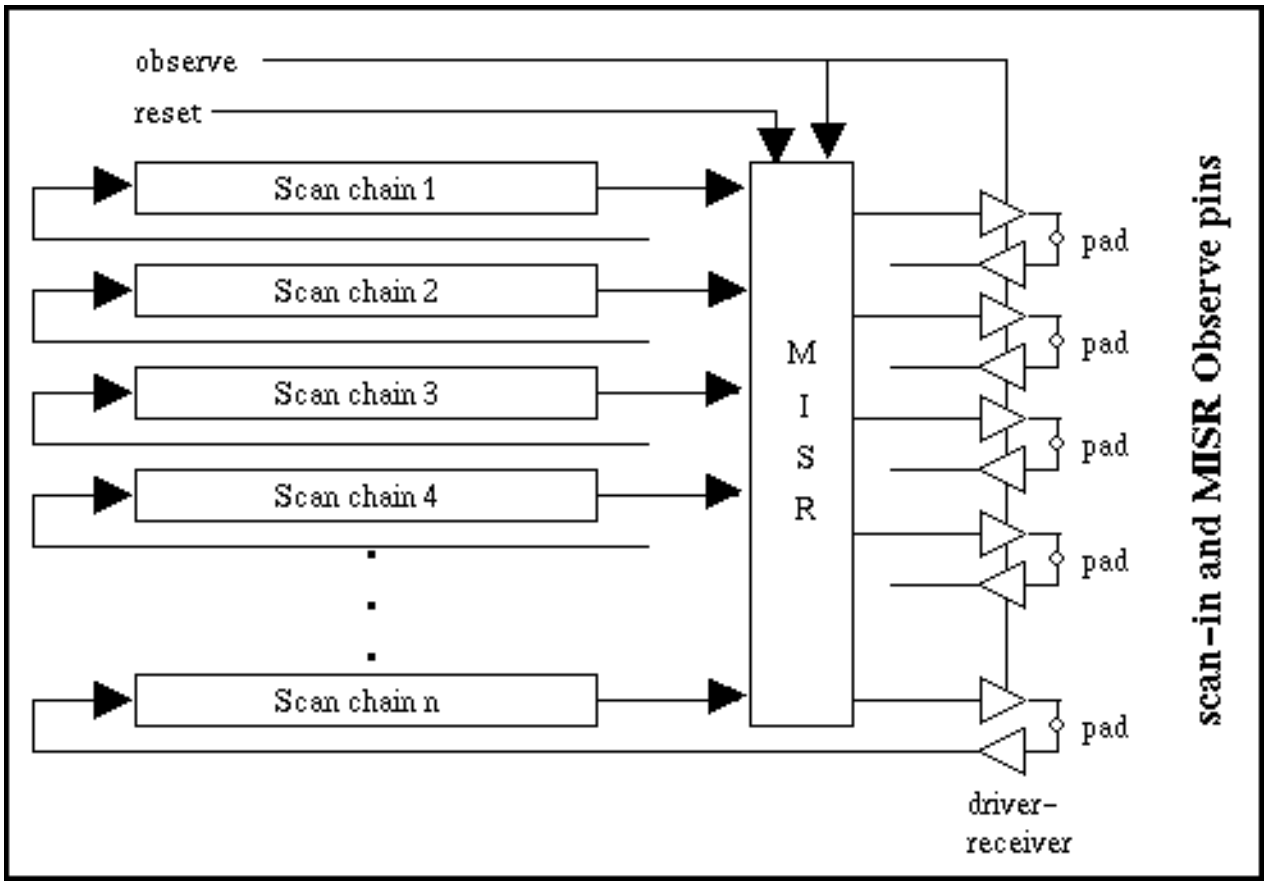
that may not be applicable in the OPMISR or OPMISR+ test modes, such as migrated test vectors for internal Cores.

Generating OPMISR TestMode

To generate an On-Product MISR test mode:

- Specify that scan-out data goes to `TO_MISR` in the SCAN mode definition statement (`OUT=TO_MISR`).

Figure A-5 On-Product MISR Configuration Example



- Specify as appropriate, these test function pins:
  - `MISR_Observe`
  - `MISR_Read`

- `MISR_Reset`
- `MISR_Reset_Enable` (optional)
- `Scan_In_Gate` (optional)
- `Scan_Out_Gate` (optional)

See "Valid Combinations of Test Functions" on page 54 for details.

- Specifying the test function pins allows automatic generation of corresponding sequences of:
  - `MISR_Observe`
  - `MISR_Reset`
  - `Diagnostics_Observe_Sequence`
  - `Diagnostics_Return_Sequence`

  Refer to the descriptions of "Define_Sequence" types in the *Test Pattern Data Reference* for additional information.

Creating a chip full scan chain test mode for diagnostics, in addition to creating an On-Product MISR test mode, ensures the completion of any processing that cannot be done in the On-Product MISR mode.

Further reduction in test data volume can be achieved by the use of test structure independent techniques such as Run-Length Encoding and more efficient algorithms for test vector compaction.

Note: OPMISR is often referred to by its shorthand `S2M` which stands for Scan2MISR. OPMISR+ is often referred to by its shorthand `OPPLUS`. These shorthand names are primarily used for pins in the logic designs that implement these test structures.

OPMISR Building Blocks

There are two building blocks that are available to support the OPMISR methodology:

1. OPMISR_<*N*> where *N* is the size of the MISR and is equal to twice the number of scan inputs (or scan outputs). The OPMISR block includes the embedded MISR and all the control logic and scan path switching logic to support the OPMISR methodology for a design with *N*/2 scan in and *N*/2 scan out pins. Table A-1 describes the pins on the OPMISR_<*N*> building block.

Table A-1 OPMISR Description

Pin Name	Direction	Bit Width	Purpose
S2M	Input	1	Enables OPMISR mode
SE	Input	1	Scan Enable
MRD	Input	1	MISR Read Enable
MRC	Input	1	MISR Reset Control/Enable
CK	Input	1	Clock that operates MISR
TEST_CLOCK_IN	Input	1	Dedicated Test Clock for scan Elements in user logic
RSI_SI	Input	*N*/2	Signals from primary scan inputs (bi-dir pin) used to scan in data from ATE
RSI_SO	Input	*N*/2	Signals from primary scan outputs (bi-dir pin) used to scan in data from ATE
SWBOX_SO	Input	*N*	Signals from channel tails or SwitchBox scan out pins used to feed test responses to MISR
SWBOX_SI	Output	*N*	Signals feeding channel heads from the RSI_SI and RSI_SO signals
DSO_SI	Output	*N*/2	OPMISR output feeding the primary scan in (bi-dir pin) during MISR Observation

Table A-1 OPMISR Description

Pin Name	Direction	Bit Width	Purpose
DSO_SO	Output	*N*/2	OPMISR output feeding the primary scan output (bi-dir pin) during MISR Observation
ESO_SI	Output	*N*/2	scan pin Bi-dir enable control for the Primary Scan Inputs
ESO_SO	Output	*N*/2	scan pin Bi-dir enable control for the primary scan outputs
TEST_CLOCK_OUT	Output	1	TEST_CLOCK_IN gated off during MISR Reset or Channel Mask scan. Serves as test clock input to user's logic

2. SIO_SYS_S2M: This building block is used to interface the OPMISR block to the bi-directional scan in and scan out pins. The following table describes the pins on the SIO_SYS_S2M building block.

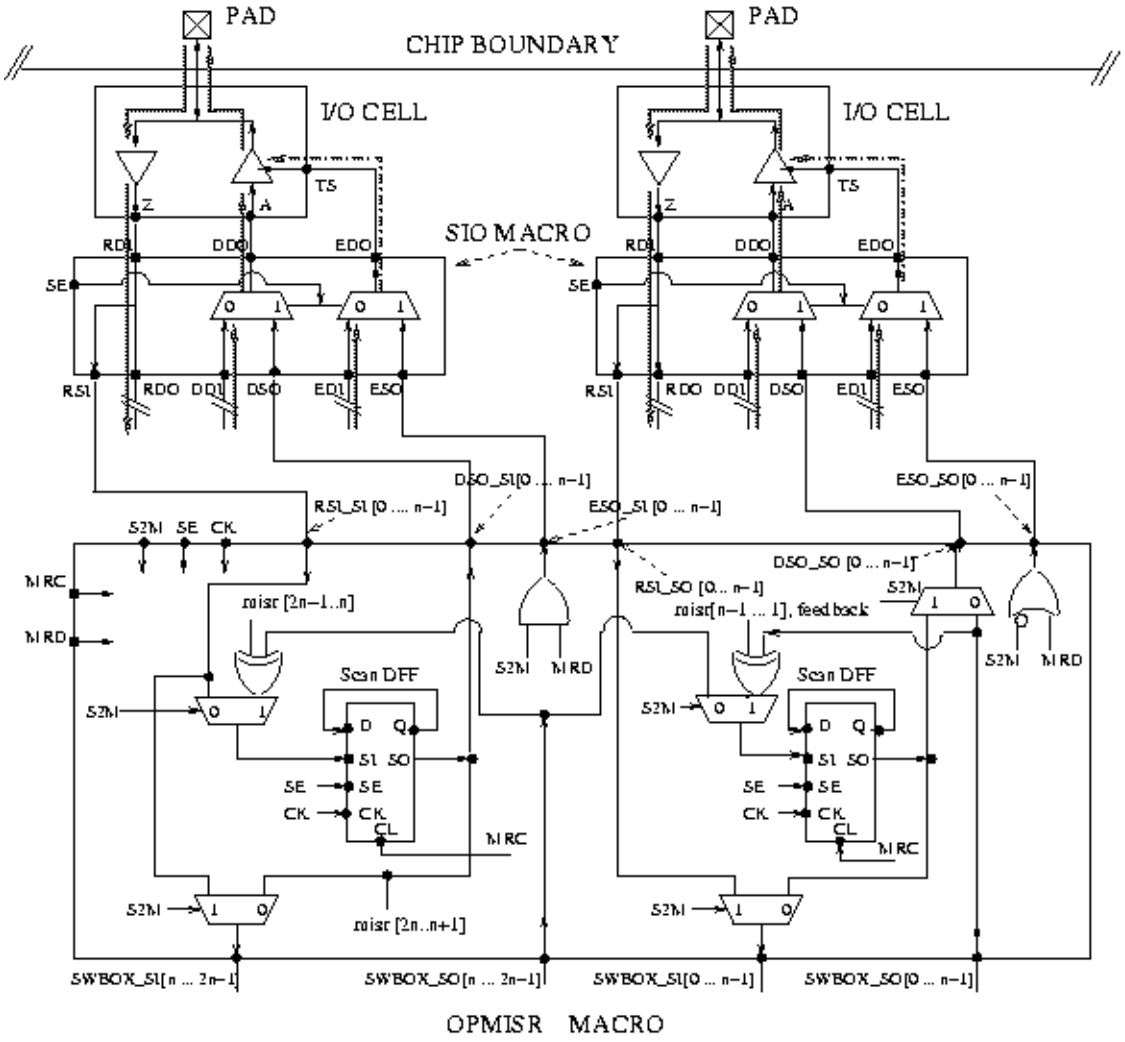
Table A-2 SIO_SYS_S2M Pin Description

Pin Name	Direction	Purpose
RDI	Input	Receiver Data Input from I/O cell
DDI	Input	Driver Data Input from system logic
DSO	Input	Driver Scan Output from last scan cell
EDI	Input	Enable Data Input from system logic
ESO	Input	Enable Scan Output from test logic
SE	Input	Scan Enable
RDO	Output	Receiver Data Output to system logic
RSI	Output	Receiver Scan Input to first scan cell
DDO	Output	Driver Data Output to system logic
EDO	Output	Enable Data Output to system logic

Modes of Operation

Figure A-6 on page 121 shows the interconnection between the OPMISR block, `SIO_SYS_S2M` blocks, the I/O cells, the internal scan chains/channels, and the flow of data during the functional mode. Essentially the `RDI` pin of the I/O cell feeds the system input logic, the `EDI` pin of the I/O cell is driven by the internal driver enable logic, and the `DDO` pin of the I/O cell is fed by the system output logic. The MISR data is not observable and no scan data is being scanned into the MISR.

Figure A-6 OPMISR in Functional Mode

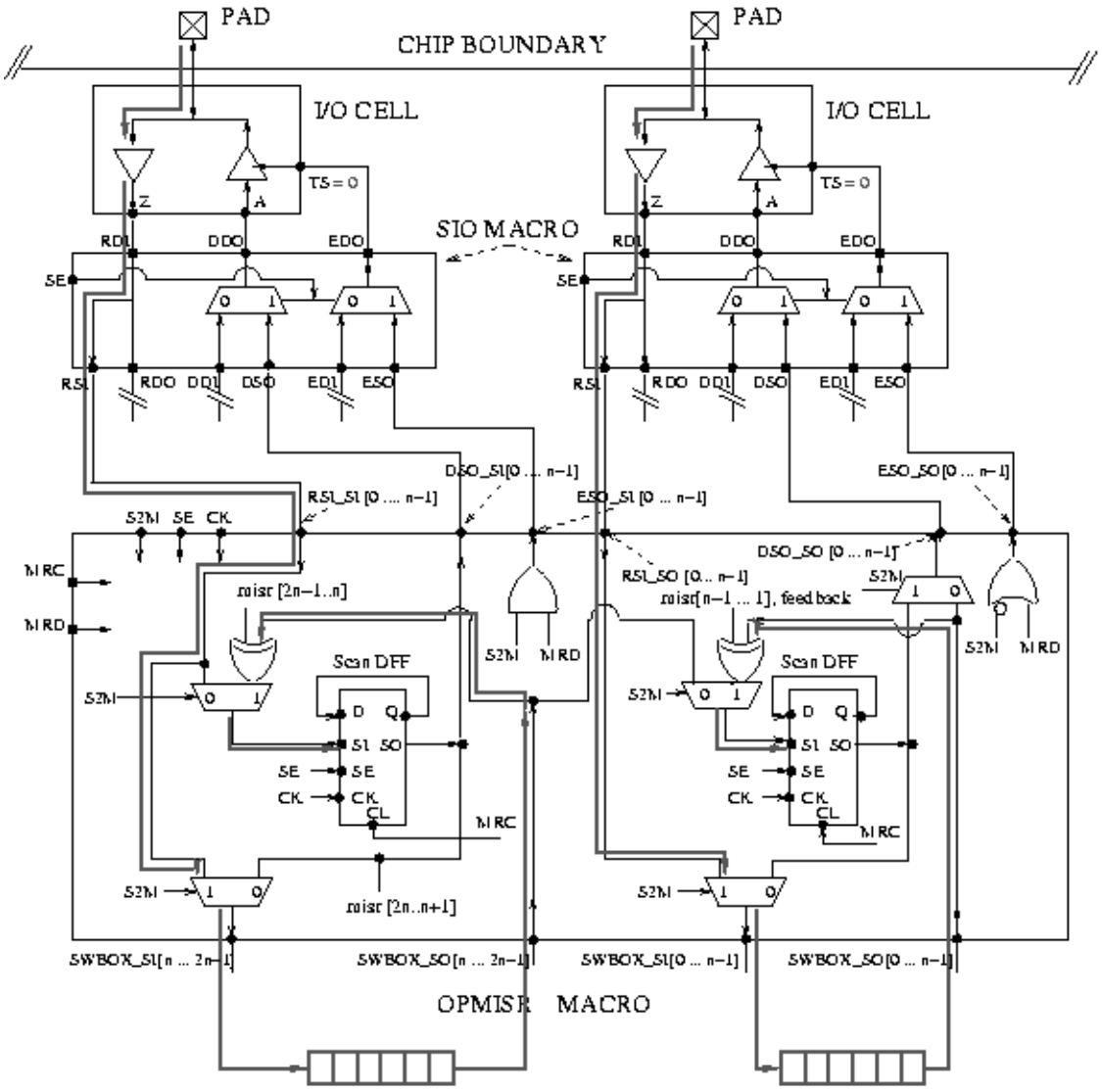


Functional Mode
SE: Scan Enable = 0
S2M: OPMISR Mode = 0
MRC: MISR Reset Clear = 0
MRD: MISR Read = 0

Figure A-7 on page 122 shows the flow of data during the OPMISR scan mode. The scan data from the scan in and scan out pins feeds the `RSI` input pins of the OPMISR block from which it is directly passed on to the scan channel heads via the `SWBOX_SI` pins. After the test is completed the test results are scanned out through the scan channel tails to the `SWBOX_SO` pins of the OPMISR block from where it is captured by the MISR. Every bit shift of the scan channel is accompanied by a bit shift of the MISR and the new test data is added into the accumulated signature. At the end of the test the scan phase is exited.

The MISR value can be read when the MRD pin is asserted as shown in Figure A-8 on page 123. The SIO_SYS_S2M cell forces the I/O cell to accept the MISR output values and transport them to the I/O pads.

Figure A-7 OPMISR Scan Mode



OPMISR Mode (During Scan shift operation)

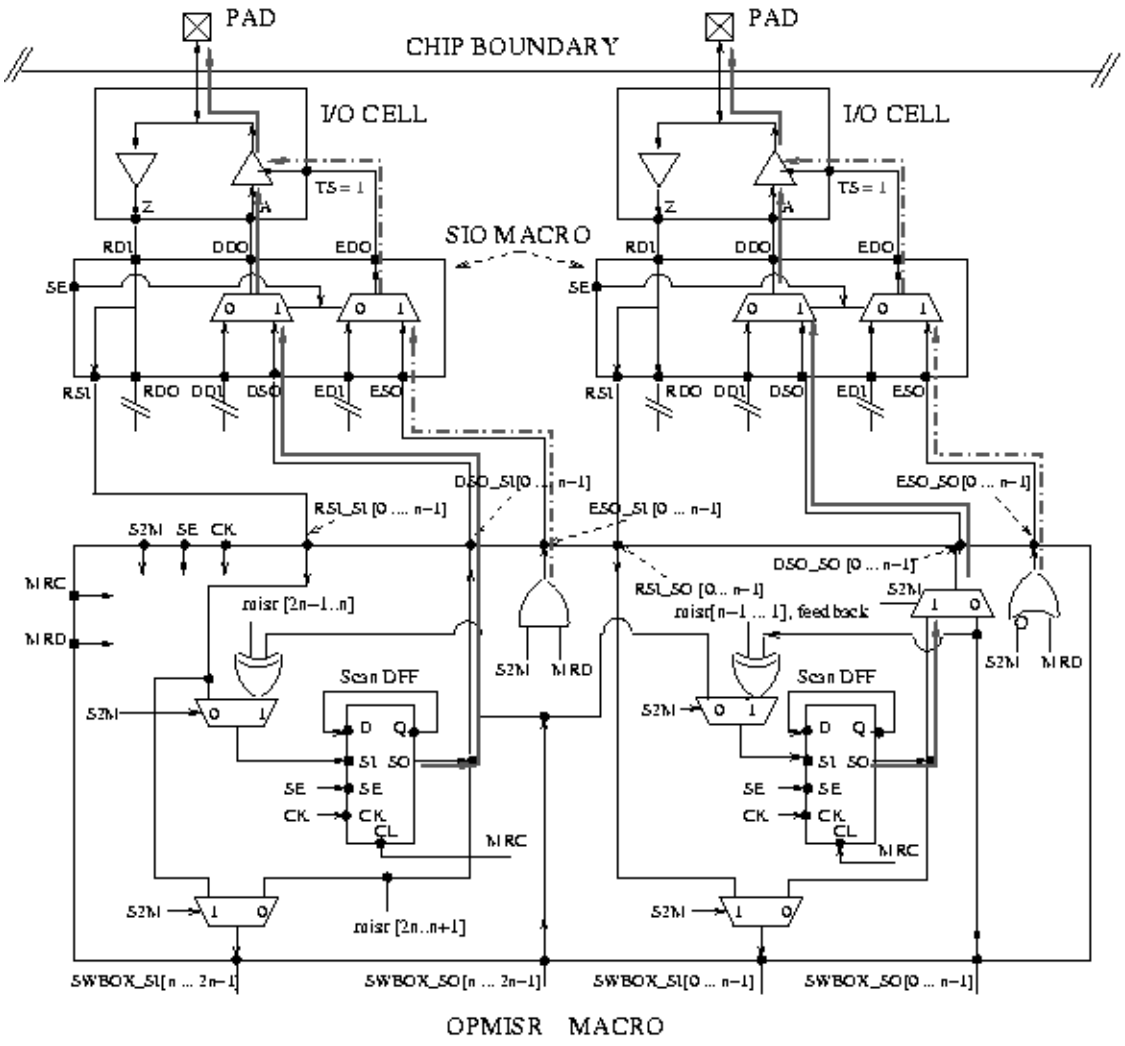
SE: Scan Enable = 1 (During scan phase)

S2M: OPMISR = 1

MRC: MISR Reset Clear = 0

MRD: MISR Read = 0

Figure A-8 OPMISR Read Operation

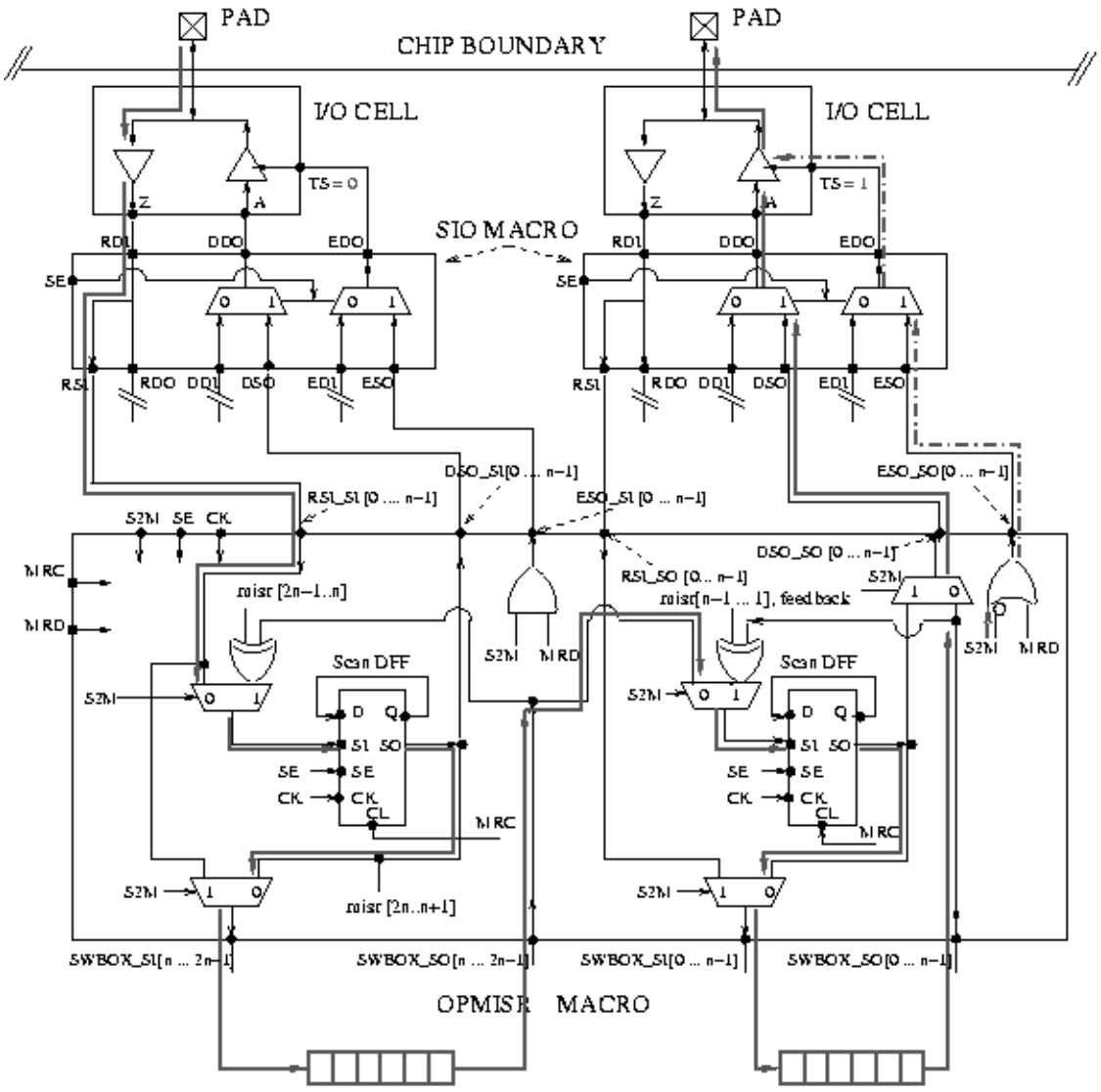


OPMISR Mode (During MISR Read Operation)

SE: Scan Enable = 1 (During scan phase)
S2M: OPMISR Mode = 1
MRC: MISR Reset Clear = 0
MRD: MISR Read = 1

Another mode that has to be considered is the full scan ATPG mode, shown in Figure A-9 on page 124. In this mode the multiplexing logic inside the OPMISR block sets up the channels to form $N/2$ scan chains that are fed scan data from the $N/2$ scan in pins and the test responses are collected from the $N/2$ scan out pins.

Figure A-9 Normal ATPG Scan Test using OPMISR



Normal Full Scan Test during Scan Operation

SE: Scan Enable = 1 (During scan phase)

S2M: OPMISR Mode = 0

MRC: MISR Reset Clear = 0

MRD: MISR Read = 0

Channel Masking Logic

Channel masking is a method that provides a way to ignore unknown (X) values that may appear in measure register bits. The MISR signature is unpredictable without the ability to mask these unknown values.

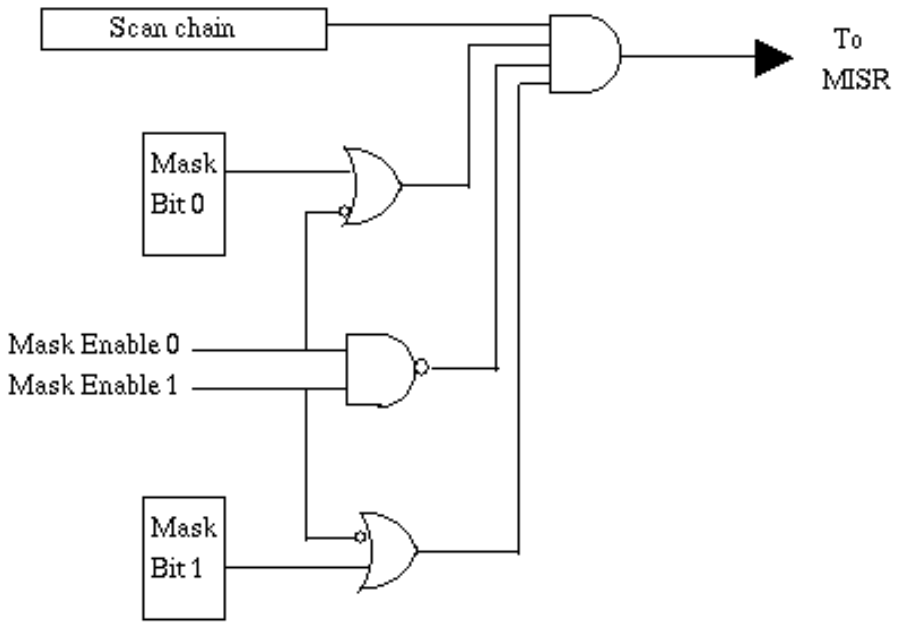
Typical scenarios where channel masking may be used are:

- In an ASIC environment where automatic insertion of channel masking logic is desired.
- In a custom environment where user-defined channel masking logic is desired.

Refer to "Implementing Channel Masking Logic in a Design" on page 127 for additional information.

Channel mask registers are loaded from scan-in pins and allow some channels to be masked out while others can be allowed into the MISR. A single channel mask bit may mask out one or more channels. The Channel_Mask_Enable (`CME`) test function pin indicates whether the channel mask bits should apply to the current scan cycle. When all CME pins are at their specified (stability) value, no masking is performed. Two `CME` pins support a total of three sets of channel mask bits plus one non-masking state. This implementation is called a `WIDE2` channel mask and is illustrated in Figure A-10.

Figure A-10 WIDE2 Channel Masking Logic for One Channel

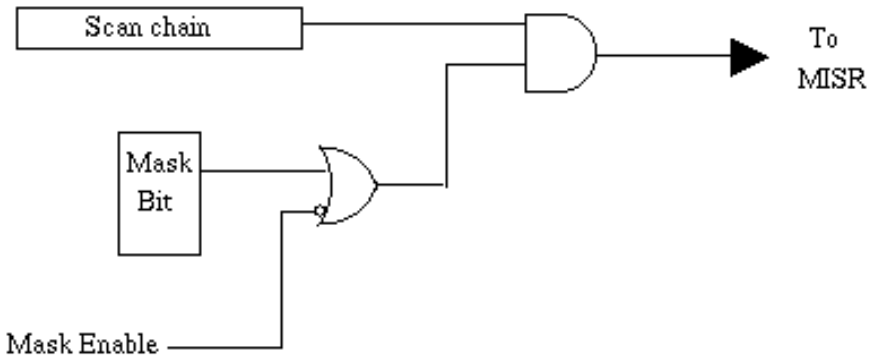


The preceding figure shows that one of the four `CME` encoding states (`CME0, CME1 = 11`) is used to mask out all channels without the need to load any mask bits. Note that `CME` pins are specified with a polarity. In Figure A-10, both signals would have been specified as `-CME` to denote that when they are both at `0`, the channels should pass through unimpeded to the MISR. This is also the state that is applied when first tracing backward from the MISR to look for the measure registers (channels). The masking logic example shown in Figure A-10 could

be shared with several channels so that all like channels would be masked in or masked out simultaneously. For channels that can be independently masked, this logic is replicated for each independent set of channels that can be masked, including a replication of the mask bit memory elements.

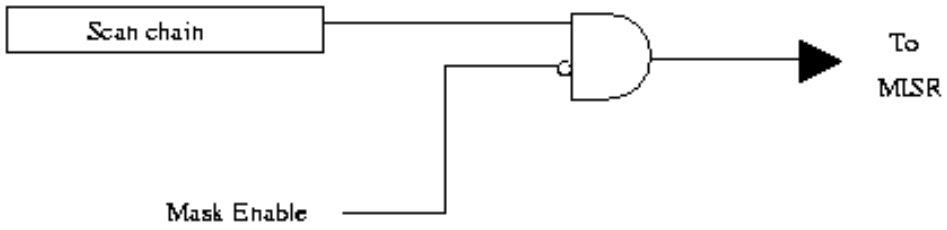
The examples in Figure A-11 and Figure A-12 denote how a single CME pin may be used. In Figure A-11 , a single mask bit is applied when the CME pin is asserted. This implementation is called a WIDE1 channel mask.

Figure A-11 WIDE1 Channel Masking with a Single CME Signal



In Figure A-12, there is no such mask bit; instead, when the CME is asserted, all channels are masked out. This implementation is called a WIDE0 channel mask.

Figure A-12 WIDE0 Channel Masking without Mask Bits

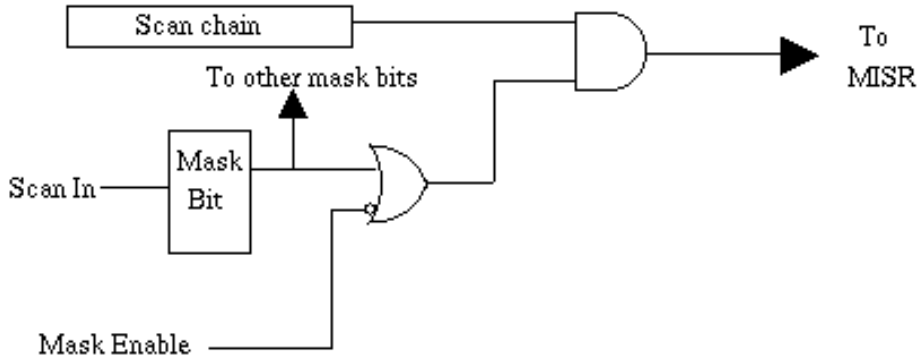


Encounter Test supports any combination of the three channel masking implementations (WIDE0, WIDE1, WIDE2) in the same test mode except that no CME pin state can unconditionally block some channels while conditionally (under control of mask bits) blocking other channels.

Figure A-13 shows channel mask bits loaded from the scan-in pins using the Channel_Mask_Load (CML) clock. The Channel_Mask_Load_Enable (CLME) test function pin defines how to enable the CML clocks to load values from the scan-in pins into the channel mask register bits. This allows the CML clock function to be shared with an existing clock function, but this is not required if a separate CML clock is defined that is used only for loading the channel mask bits.

Use `Channel Mask Load` design state to analyze problems related to the loading of channel masks. Channel_Mask_Input (CMI) pins indicate where mask bits are loaded from, however if unspecified, Encounter Test assumes that Scan_In (SI) pins serve this function. Refer to "Design States" on page 295 for related information.

Figure A-13 Mask Bits Loaded from the Scan-in Pin



Note: Note that each `mask_enable` signal will operate as if it was a scan input, as far as the tester interface is concerned. This means that `mask_enable` signals will consume ATE scan pin resources and thus will reduce the number of scan pins available for loading test data into the design i.e., the masking-per-cycle information is considered to be additional test data.

Implementing Channel Masking Logic in a Design

To automatically insert channel masking logic, use Test Synthesis to select whether the masking logic should be included. Test Synthesis inserts two CME pins and two sets of mask bits with logic equivalent to that shown in Figure A-10 on page 125.

After the masking logic is inserted, perform the following tasks:

1. Define the test mode, specifying at least one CME pin and one CML pin. Optionally, Channel_Mask_Load_Enable (CMLE) pins may be defined that provide the state in which the CML clock will load the mask bit data into the mask bit registers. CMI pins may also

be defined, however Encounter Test assumes that all `SI` pins are also `CMI` pins when channel masking is defined.

Refer to "ASSIGN" on page 210 for the syntax required to define the test mode using these test function pins.

Note: If both `CME` and `CML` test functions are defined, the resulting sequence definition file will contain these channel masking logic related sequence types:

- `channelmaskprecond`
- `channelmaskload`
- `channelmaskcycle`
- `channelmaskexit`

Refer to "Define_Sequence" in the *Test Pattern Data Reference* for descriptions of these sequence types.

2. Analyze Test Structures to perform checking functions that validate masking logic.
3. Perform ATPG to determine fault coverage. If fault coverage is unacceptable due to channel masking, submit another ATPG run and specify either:

   - `Xprevent=cycle` to indicate that ATPG is to block all sources of `X` that can be loaded onto any measure latch that scans out on the same scan cycle as the measure latch used to observe the target fault.

     or

   - `Xprevent=channel` to indicate that ATPG is to block all `X`s that can be loaded into the same measure register (channel) as the measure latch used for the target fault, and any other measure registers that are masked with the same mask register bit.

   Note: The fault coverage should improve using either method, however pattern counts will likely increase.

   If fault coverage is still unacceptable, specify `Xmask=yes` to allow all faults to be detected, however this method produces higher data volumes.

Channel Masking on Pad Cycles

Channel masking signatures for OPMISR designs with unbalanced scan chains are automatically calculated. Signature calculations for OPMISR+ designs with channel masking differ in the following ways:

- Short chains receive pre-padding zeros unless their scan-in also feeds longer chains.

- OPMISR+ produces `Fix_MISR` events to account for non-zero padding and scan overlap. This allows for tests to be reordered. Refer to "Fix_MISR" in the *Test Pattern Data Reference* for details.

The preceding allows the possibility to mask out a scan cycle containing non-zero padding and rendering the `Fix_MISR` event ineffective. The solution is to prevent masking on scan cycles that may have a non-zero pad.

Figure A-14 illustrates a channel masking scenario with variable length overlapped scans.

Figure A-14 Variable Length Overlapped Scans

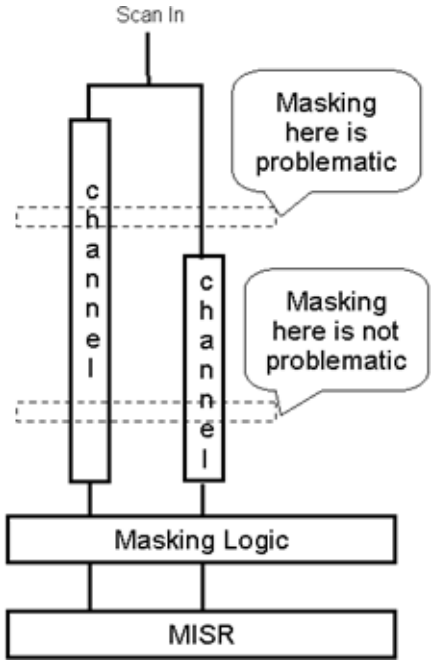


Figure A-15 depicts a scenario of exposing the effectiveness of the `Fix_MISR` event due to overlapped scans.

Figure A-15 Exposure from Overlapped Scans without Masking

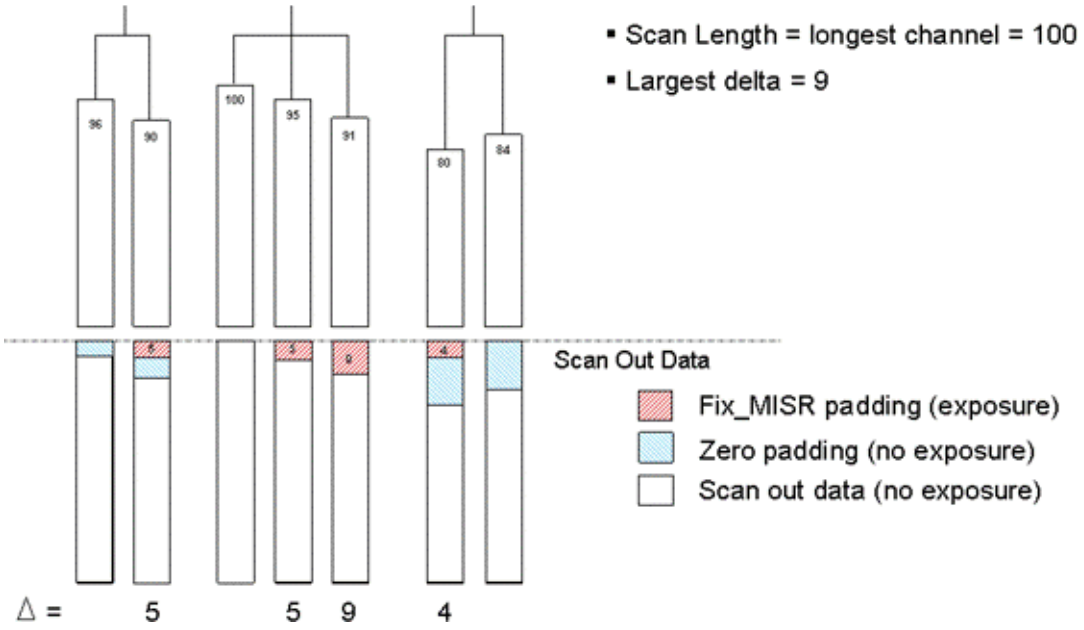
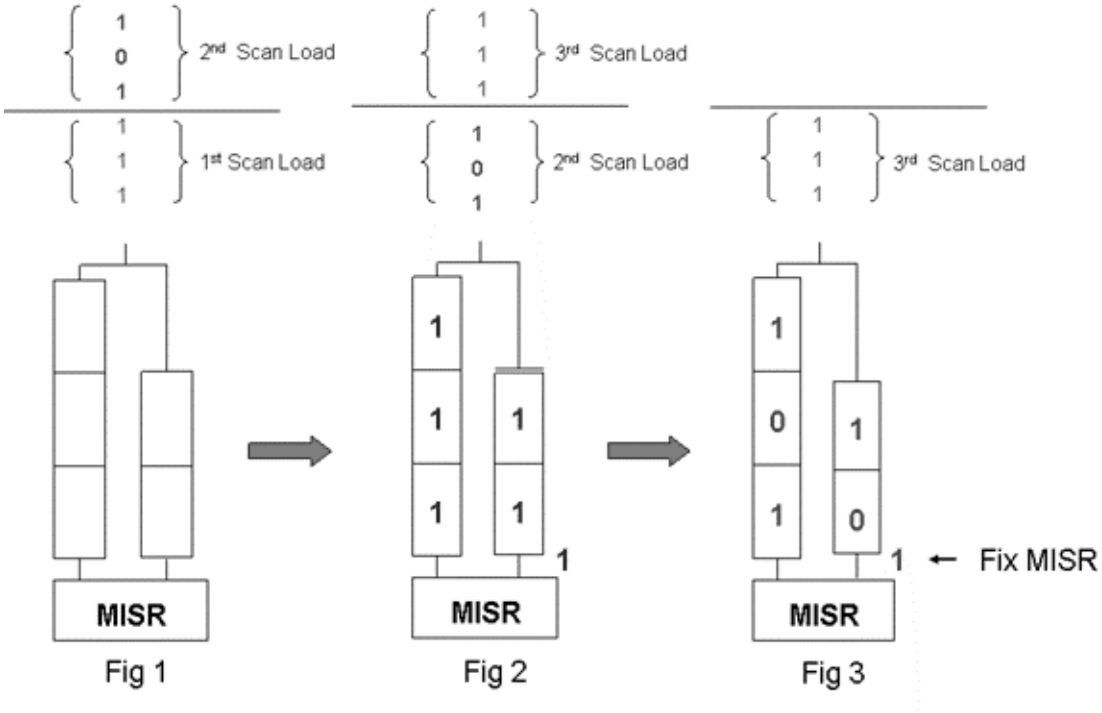


Figure A-16 shows that the first shift in `1` of the second set of vectors (`1 0 1`) overlaps the MISR for the shorter scan chain, while the MISR bit from the longer scan chain is still unloading the previous scan unloads into the MISR.

Figure A-16 Scan Overlap



For the shorter scan chain, the MISR for pattern `1` is affected by the second scan load in the preceding figure.

The MISR signature is processed to properly calculate the extra value from the second cycle of the MISR. In other words, the MISR Observe expected values are calculated considering an extra bit from the next cycle.

In terms of tester hardware, this extra `1` from the next scan load would go inside the MISR hardware and when misr observe attributes are applied it will affect the signature. In order to account for this, the `Fix_MISR` event ensures that this extra bit has been considered when determining the final MISR signature. The "`Fix_MISR`" signature is affiliated with the second test pattern to allow for pattern reordering and ensuring correct MISR signatures.

Tip

The `Fix_MISR` event is not required if there is no scan overlapping; there are constant 0's instead of changing values in the second scan.

If an X value is propagated into a MISR, the signature is rendered invalid. The term *X-mask* padding is the use of the channel mask unit to mask X values to avoid masking a value from the next cycle in the shorter scan chain.

Figure A-17 illustrates an X-mask padding problem caused by a two bit shift.

Figure A-17 X-Mask Padding Problem

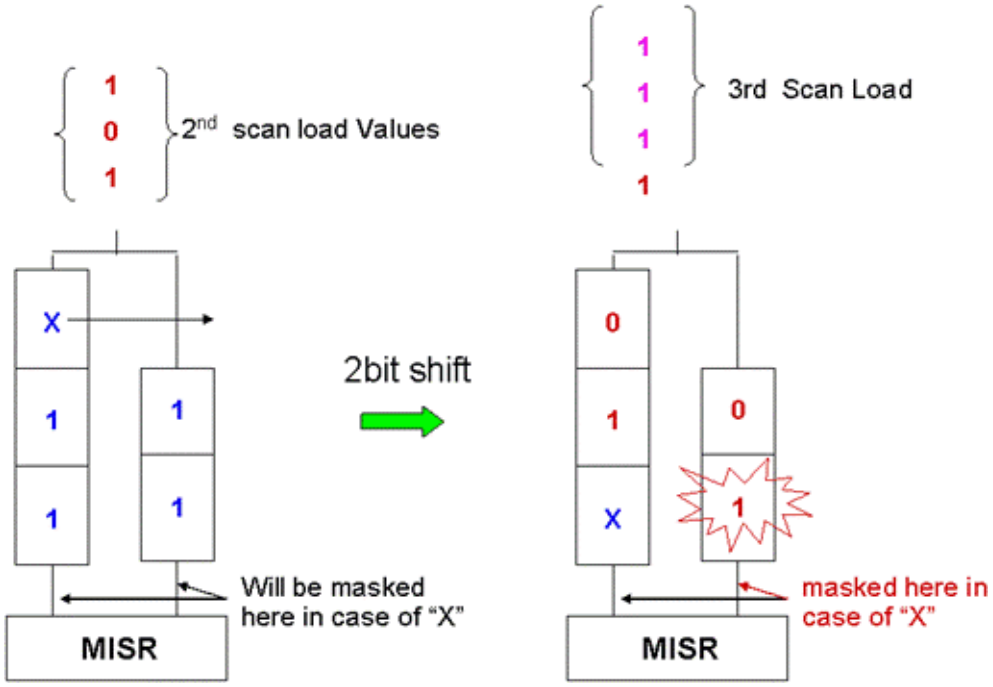


Figure A-18 illustrates a solution to the X-mask problem in the preceding figure by using X-mask 0 padding.

Figure A-18 X-Mask 0 Padding Solution

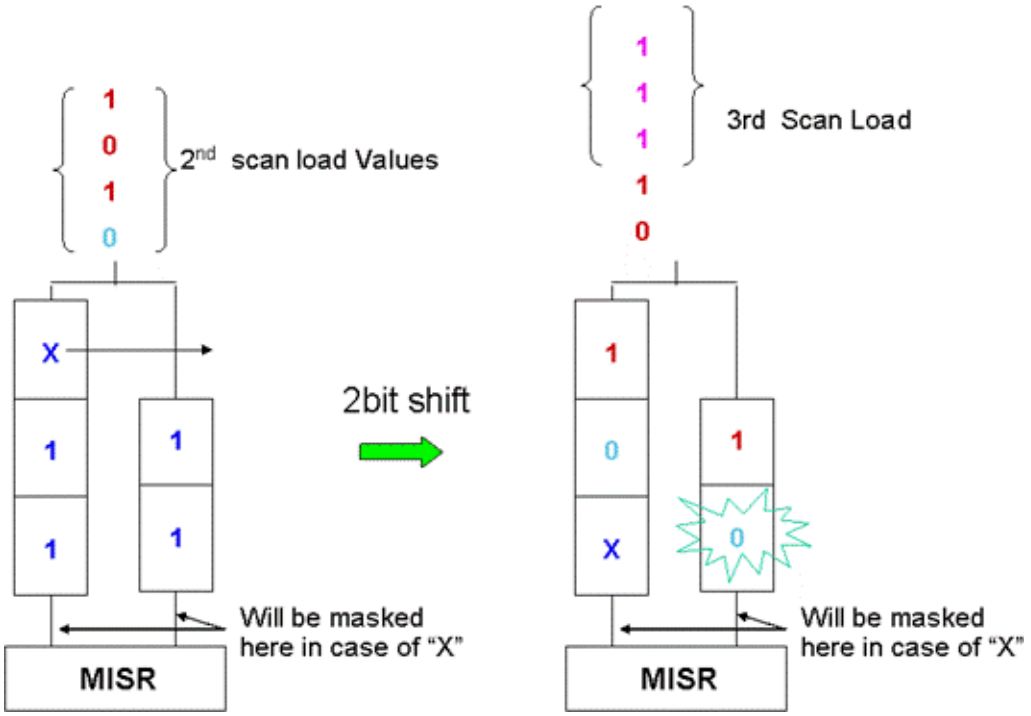
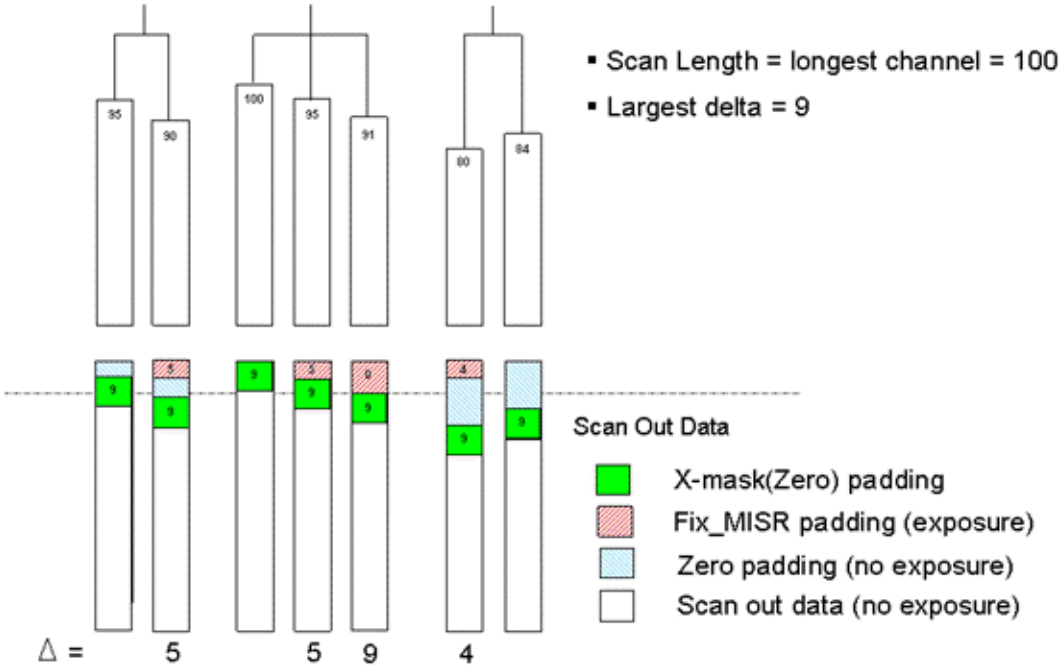


Figure A-18 illustrates why "X's" must be masked by the channel mask unit before they can propagate inside the MISR. The term X-mask 0 padding is the avoidance of masking a value from the next cycle in the shorter scan chain by adding padding cycle "0" values. X-mask 0 padding is required to reorder patterns.

The amount of 0 padding is based upon where the X's are found in the channels and the presence of channel length differences feed by the same scan inputs as illustrated in Figure A-19 .

Figure A-19 X-Mask Channel Masking on Pad Cycles



Tip

X-mask 0 padding is not required without *scan overlapping* because there will be no second cycle following the first cycle.

An optimum OPMISR+ design goal is to ensure all scan channels fed by the same scan-In are the same approximate length. Stated differently, for each scan-in pin where the pin fans out, it is recommended that each fan-out have the same scan chain length

The following is a design example where 0 padding is required:

```
Controllable Scan Chain 5
 Load Pin Index / Pin Name     211 / PI_Pin
 Bit Length    493
 Scan Section Sequence    Scan_Section_Sequence
Observable Register 5
 Unload Pin Index / Pin Name   2583136 / macro_pin1
 Bit Length                        493   Unload pin inverted from Load Pin
 Scan Section Sequence      Scan_Section_Sequence
 Feeds MISR Register 1  Bit 5  Inverted
Observable Register 53
 Unload Pin Index / Pin Name   2583412 / macro_pin2
```

```
Bit Length                          43   Unload pin inverted from Load Pin
Scan Section Sequence       Scan_Section_Sequence
Feeds MISR Register 1  Bit 53  Inverted
```

The preceding example is a 43 bit scan chain and a 493 bit scan chain within the same fan-out group. Because the 43 bit scan chain will finish scanning before the 493 bit scan chain, the scan in data must be padded with 0's to ensure that values are going into the MISR for the full length of the scan. This means the scan-in becomes approximately 900 bits as opposed to approximately 500 bits. This is because for the 450 bits (493-43 bits) pad 0's must be padded to ensure the short 43 bit scan chain has consistent values into the MISR. The scan data can be scanned in after the padding.

Building a Test Mode for OPMISR, OPMISR+

The Encounter Test test modes will automatically be generated from RTL Compiler allowing users to process a design containing an OPMISR or OPMISR+ module through Test Structure Verification and ATPG. The assign file identifies the scan in and scan out data pins and the various test control signals. Figures Figure A-20 on page 137, Figure A-21 on page 138, and Figure A-22 on page 139 show the ATPG, OPMISR and OPMISR+ test mode assign files for the DLX tutorial design. Refer to "Assign File" on page 22 for additional information on creating a test mode assign file. This design uses an OPMISR+ macro with a misr size of 16, a fanout of 5, and included `WIDE2` channel masking.

Configuring Pin Assignments for Scan

Scan is entered by the `scanentry` sequence, and exited by the `scanexit` sequence. In between, in order, are the optional `mask_load`, scan load/unload, `MISR_read`, and `MISR_reset` states. The `+/-SE` flag sets the value of a pin for all four of those states unless overridden by one or more of the `MRD`, `MRE`, or `CMLE` flags for a specific state. Thus, the `MRD`, `MRE`, and `CMLE` pins require an `SE` flag to set the default scan value and any necessary state override. When not in scan (during test launch/capture), these can take any value unless there is also a `TC` (Test Constraint) flag.

`MRC` is a port on the OPPLUS macro; `MRE` is the test flag associated with that port, MISR Reset Enable is the function (it is not a clock, but a clock gate).

`CK`, the `MISR_Clock`, is assumed to be a test clock dedicated to the MISR/Mask logic and not used elsewhere. It can be used elsewhere, but must be degated for at least the `Mask_Load` and `MISR_Reset` states. This must be manually configured.

When using OPMISR+, the scan tests for `FULLSCAN`, `OPMISR`, and `OPPLUS` modes should all be generated and committed first before starting ATPG on the rest of the logic. Verify that `FULLSCAN` and `OPMISR/OPPLUS` are getting cross-mode markoff.

The following tables depict pin usage combinations for the FULLSCAN, OPMISR/OPPLUS, and Channel Masking.

Table A-3 FULLSCAN Mode Pin Matrix

Pin Type	Direction	Shareable	Test Function
Test Clock	Input	Yes	-ES (can be functional clock)
Scan Enable	Input	Yes	+SE, -MRE (MRE for compression only)
Test Enable	Input	No	+TI
Scan In (N)	Input	Yes	SI
Scan Out (M)	Output	Yes	SO

Table A-4 OPMISR/OPMISR+ Mode Pin Matrix

Pin Type(s)	Direction	Shareable	Test Function
S2M	Input	Yes	+TI, -SE (optional if using OPMISR+ only)
OPPLUS	Input	Yes	-TI, -SE
MISR Reset MISR Clock	Input	No	MRST, -ES, -CML (CML for masking only)
MISR Reset Enable	Input	Yes	+MRE, -SE
MISR Read Enable	Input	Yes	+MRD (optional if using unidirectional)

Table A-5 Channel Masking Pin Matrix

Pin Type(s)	Direction	Shareable	Test Function	WIDE0	WIDE1	WIDE2
CME0	Input	Yes	-CME	Yes	Yes	Yes
CME1	Input	Yes	-CME	No		Yes
CMLE	Input	Yes	-CMLE	No	Yes	Yes

Refer to "Identifying Test Function Pins (Advanced)" on page 28 for related information.

Assign File Examples

The following are examples of ASSIGN files that configure scan-based test modes. Refer to "ASSIGN" for related information.

Figure A-20 Test Mode Assign File for Full-Scan ATPG

```
assign pin=DLX_CHIPTOP_TEST_ENABLE test_function= +TI;
assign pin=DLX_CHIPTOP_TEST_CLOCK test_function= -ES;
assign pin=DLX_S2M test_function= -TI;
assign pin=DLX_MRD test_function= -SE;
assign pin=DLX_MRE test_function= -SE;
assign pin=DLX_MRST test_function= -ES;
assign pin=DLX_OPPLUS test_function= -TI;
assign pin=DLX_CME0 test_function= -SE;
assign pin=DLX_CME1 test_function= -SE;
assign pin=DLX_CMLE test_function= +SE;
assign pin=DLX_CHIPTOP_TDI test_function= -TI;
assign pin=DLX_CHIPTOP_TCK test_function= +TI;
assign pin=DLX_CHIPTOP_TMS test_function= -TI;
assign pin=DLX_CHIPTOP_TRST test_function= -TI;
assign pin=DLX_CHIPTOP_SE test_function= +SE;
assign pin=DLX_CHIPTOP_RESET test_function= +SC;
assign pin=DLX_CHIPTOP_SYS_CLK test_function= -ES;
assign pin=SI0 test_function= BDY, SI;
assign pin=SI1 test_function= BDY, SI;
assign pin=SI2 test_function= BDY, SI;
assign pin=SI3 test_function= BDY, SI;
assign pin=SI4 test_function= BDY, SI;
assign pin=SI5 test_function= BDY, SI;
assign pin=SI6 test_function= BDY, SI;
assign pin=SI7 test_function= BDY, SI;
assign pin=SO0 test_function= BDY, SO;
assign pin=SO1 test_function= BDY, SO;
assign pin=SO2 test_function= BDY, SO;
assign pin=SO3 test_function= BDY, SO;
assign pin=SO4 test_function= BDY, SO;
assign pin=SO5 test_function= BDY, SO;
assign pin=SO6 test_function= BDY, SO;
assign pin=SO7 test_function= BDY, SO;
```

Figure A-21 Test Mode Assign File for OPMISR Mode

```
============================================================
/* Identify the feedback net of the OPMISR */
misr OPMISR.OPMISR_N.MISR_OUT(15) BUF_IN=(0,2,3,5,16)     ;
assign pin=DLX_CHIPTOP_TEST_ENABLE test_function= +TI;
assign pin=DLX_CHIPTOP_TEST_CLOCK test_function= -ES;
assign pin=DLX_S2M test_function= +TI;
assign pin=DLX_MRD test_function= +MRD, -SE;
assign pin=DLX_MRE test_function= +MRE, -SE;
assign pin=DLX_MRST test_function= -ES, -MRST, -CML;
assign pin=DLX_OPPLUS test_function= -TI;
assign pin=DLX_CME0 test_function= -CME;
assign pin=DLX_CME1 test_function= -CME;
assign pin=DLX_CMLE test_function= +CMLE;
assign pin=DLX_CHIPTOP_TDI test_function= -TI;
assign pin=DLX_CHIPTOP_TCK test_function= +TI;
assign pin=DLX_CHIPTOP_TMS test_function= -TI;
assign pin=DLX_CHIPTOP_TRST test_function= -TI;
assign pin=DLX_CHIPTOP_SE test_function=+SE, -MRE;
assign pin=DLX_CHIPTOP_RESET test_function= +SC;
assign pin=DLX_CHIPTOP_SYS_CLK test_function= -ES;
assign pin=SI0 test_function= SI, MO;
assign pin=SI1 test_function= SI, MO;
assign pin=SI2 test_function= SI, MO;
assign pin=SI3 test_function= SI, MO;
assign pin=SI4 test_function= SI, MO;
assign pin=SI5 test_function= SI, MO;
assign pin=SI6 test_function= SI, MO;
assign pin=SI7 test_function= SI, MO;
assign pin=SO0 test_function= SI, MO;
assign pin=SO1 test_function= SI, MO;
assign pin=SO2 test_function= SI, MO;
assign pin=SO3 test_function= SI, MO;
assign pin=SO4 test_function= SI, MO;
assign pin=SO5 test_function= SI, MO;
assign pin=SO6 test_function= SI, MO;
assign pin=SO7 test_function= SI, MO;
```

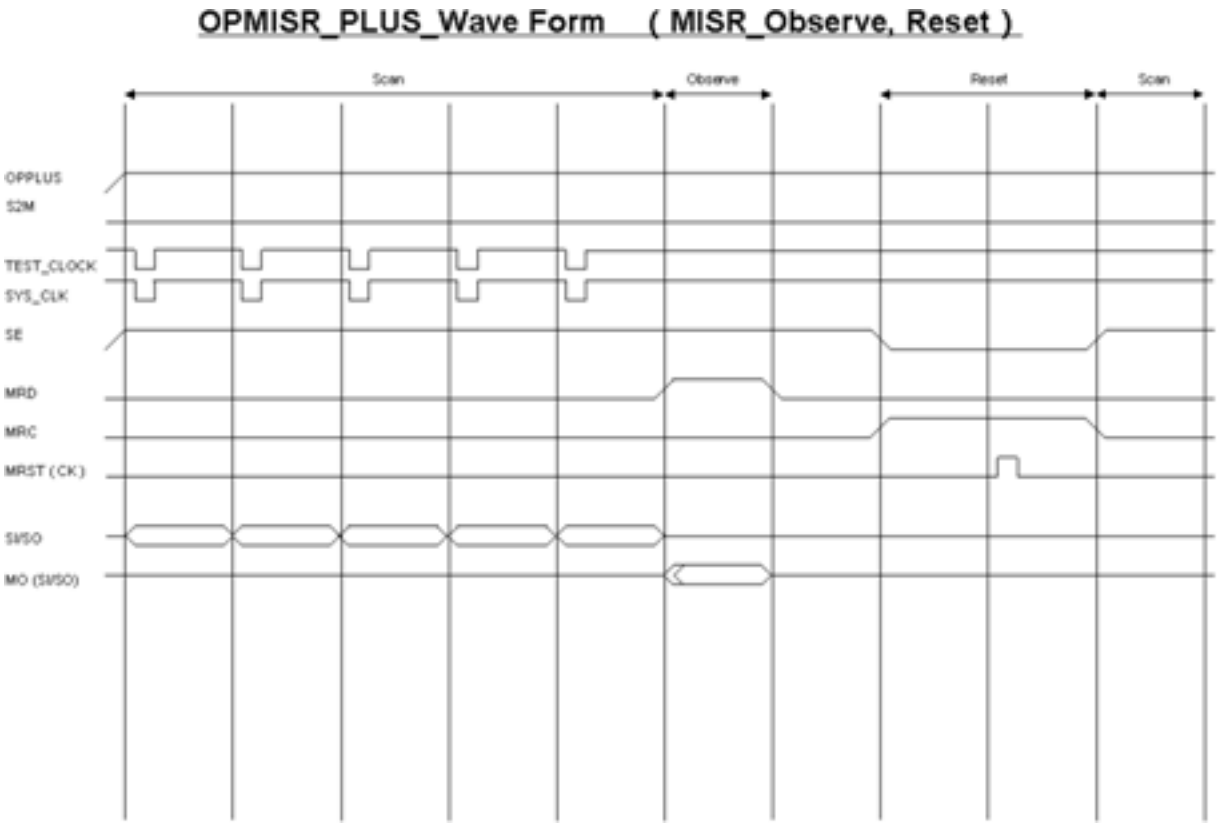
Figure A-22 Test Mode Assign File for OPMISR+ Mode

```
misr OPMISR.OPMISR_0.MISR_OUT(15)_BUF_IN=(0,2,3,5,16)     ;
misr OPMISR.OPMISR_1.MISR_OUT(15)_BUF_IN=(0,2,3,5,16)     ;
misr OPMISR.OPMISR_2.MISR_OUT(15)_BUF_IN=(0,2,3,5,16)     ;
misr OPMISR.OPMISR_3.MISR_OUT(15)_BUF_IN=(0,2,3,5,16)     ;
misr OPMISR.OPMISR_N.MISR_OUT(15)_BUF_IN=(0,2,3,5,16)     ;
assign pin=DLX_CHIPTOP_TEST_ENABLE test_function= +TI;
assign pin=DLX_CHIPTOP_TEST_CLOCK test_function= -ES;
assign pin=DLX_S2M test_function= -TI;
assign pin=DLX_MRD test_function= +MRD, -SE;
assign pin=DLX_MRE test_function= +MRE, -SE;
assign pin=DLX_MRST test_function= -ES, -MRST, -CML;
assign pin=DLX_OPPLUS test_function= +TI;
assign pin=DLX_CME0 test_function= -CME;
assign pin=DLX_CME1 test_function= -CME;
assign pin=DLX_CMLE test_function= +CMLE;
assign pin=DLX_CHIPTOP_TDI test_function= -TI;
assign pin=DLX_CHIPTOP_TCK test_function= +TI;
assign pin=DLX_CHIPTOP_TMS test_function= -TI;
assign pin=DLX_CHIPTOP_TRST test_function= -TI;
assign pin=DLX_CHIPTOP_SE test_function=+SE, -MRE;
assign pin=DLX_CHIPTOP_RESET test_function= +SC;
assign pin=DLX_CHIPTOP_SYS_CLK test_function= -ES;
assign pin=SI0 test_function= SI, MO;
assign pin=SI1 test_function= SI, MO;
assign pin=SI2 test_function= SI, MO;
assign pin=SI3 test_function= SI, MO;
assign pin=SI4 test_function= SI, MO;
assign pin=SI5 test_function= SI, MO;
assign pin=SI6 test_function= SI, MO;
assign pin=SI7 test_function= SI, MO;
assign pin=SO0 test_function= SI, MO;
assign pin=SO1 test_function= SI, MO;
assign pin=SO2 test_function= SI, MO;
assign pin=SO3 test_function= SI, MO;
assign pin=SO4 test_function= SI, MO;
assign pin=SO5 test_function= SI, MO;
assign pin=SO6 test_function= SI, MO;
assign pin=SO7 test_function= SI, MO;
```

Waveform Example

Figure A-23 is an example waveform based on an OPMISR+ test mode with defined `MISR_Observe` test function.

Figure A-23 OPMISR Waveform Example



Specifying Attributes for a MISR Test Mode

MISR attributes may be specified in a netlist to identify properties to be considered as MISR properties for a test mode. Build Test Mode accepts a specification of property names to be used as MISR properties for the test mode either via the `MISR_PROPERTIES` mode definition statement or the ASSIGN file. Matching the properties in the design to the allowed properties list enables differing sets of MISRs to be selected in multiple test modes.

Use the MISR properties in the netlist to identify nets corresponding to the last latch in each MISR LSFR with any meaningful property name. The syntax of the property must follow the same syntax as the `TB_MISR` property. Refer to "PRPG and MISR Properties" in the *Encounter Test: Reference: Legacy Functions*.

Important

It is strongly recommended to begin the name of specified MISR properties with the prefix `TB_MISR` to avoid conflict with other property names.

The following illustrates a source example:

Example A-7 MISR Attributes using TB_MISR Properties

```
TB_MISR_aaa Net.f.l.SSC1.nl.opmisr.M_OUT[31]=(0,23,29,30,32);
TB_MISR_bbb Net.f.l.SSC1.nl.opmisr2.M_OUT[31]=(0,23,29,30,32);
TB_MISR_ccc Net.f.l.SSC1.nl.opmisr3.M_OUT[31]=(0,23,29,30,32);
TB_MISR_aaa Net.f.l.SSC1.nl.opmisr4.M_OUT[31]=(0,23,29,30,32);
TB_MISR_ccc Net.f.l.SSC1.nl.opmisr4.M_OUT[31]=(0,23,29,30,32);
```

Specifying the following mode definition statement:

```
MISR_PROPERTIES=TB_MISR_aaa, TB_MISR_bbb;
```

produces the following nets that identify MISRs in the test mode:

```
Net.f.l.SSC1.nl.opmisr.M_OUT[31]=(0,23,29,30,32);
Net.f.l.SSC1.nl.opmisr2.M_OUT[31]=(0,23,29,30,32);
Net.f.l.SSC1.nl.opmisr4.M_OUT[31]=(0,23,29,30,32);
```

If the `MISR_PROPERTIES` statement is not used, then blocks with the `TB_MISR` property are used to determine the MISRs identified for the test mode.

If the `MISR_PROPERTIES` statement is used, then only the blocks with matching MISR properties will be selected for MISR identification. All blocks with non-matching MISR properties, including `TB_MISR`, will be ignored during MISR identification,

Refer to "MISR_PROPERTIES" on page 242 for details on the mode definition syntax.

Inserting Block-Level Test Compression

The preceding sections describe the OPMISR+ methodology that has been automated for insertion at the top-level of the design. With the standardization of core-based testing methods by the IEEE 1500 standard (see IEEE 1500 Core Wrapping Logic), it may be desirable to integrate cores that already have built-in test compression hardware. In a SoC methodology the following are potential advantages of supporting OPMISR+ insertion at the block or core-level:

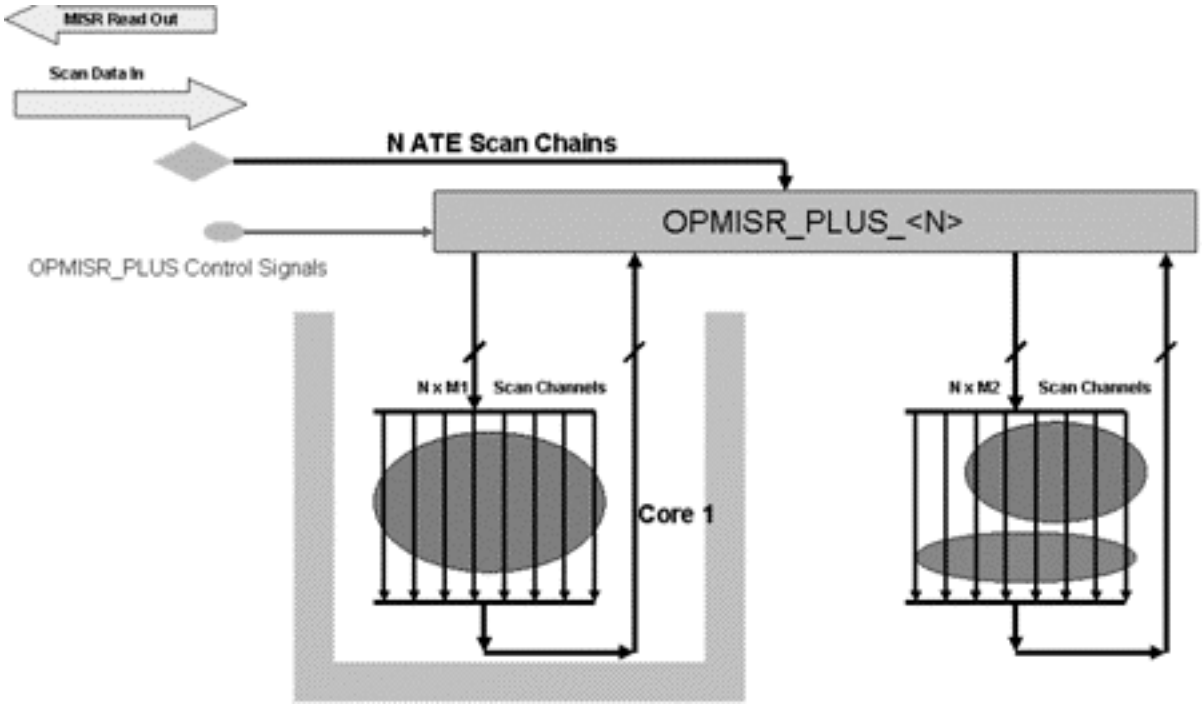
- An efficient bottom-up design flow is supported with the following features:

  - ❑ Each core can be separately verified using Verify Test Structures. Refer to "Verify Test Structures" in the *Encounter Test: Guide 3: Test Structures* for additional information.
  - ❑ Core fault coverage can be analyzed before SoC is integrated.
  - ❑ Independently verifying each core simplifies SoC-level DFT verification.
  - ❑ Impact of late design changes and ECs on DFT is minimized.
- ■ Multiple embedded cores with OPMISRs can be concurrently tested.
- ■ Mutually exclusive Scan I/Os
- ■ The same test control inputs are shared
- ■ Tests are generated at the SoC-level (no test re-use / migration).
- ■ Cores with embedded OPMISRs can be concurrently tested with a global OPMISR for glue logic and other user-designed logic (UDL).
- ■ Sequential testing of embedded cores can be easily supported by scheduling the test session of each (group of) core(s) serially.

Note: Insertion of OPMISR+ compression logic is only supported on systems that have RTL Compiler installed (when running RTL Compiler within Encounter Test a separate RTL Compiler license is not required).

Figure A-24 on page 143 shows the top-level OPMISR+ used in a SoC design. One embedded core is shown on the left and the UDL is shown on the right. An OPMISR block of size *N* is embedded at the top-level. The core has *NxM1* scan chains and the UDL has *NxM2* scan chains. The total number of internal scan chains is *Nx(M1+M2)*. At the top-level of the chip these may be connected to *N*/2 bi-directional scan in ports and *N*/2 scan out ports. Alternately, these may be connected to *N* uni-directional scan input ports and *N* uni-directional scan out ports.

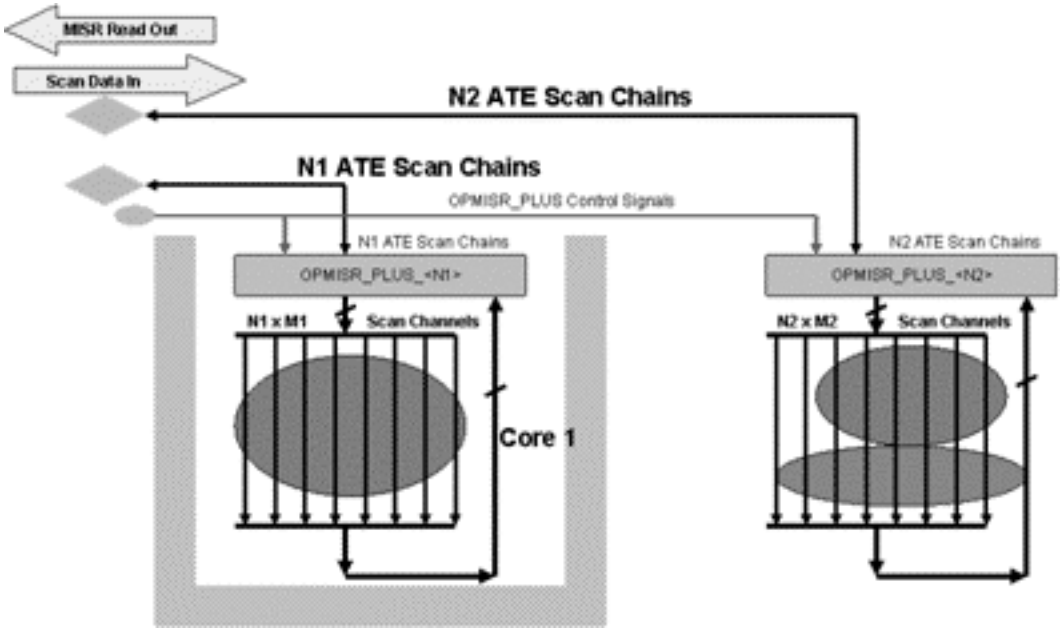
Figure A-24 Top-Level OPMISR+



An alternate approach that may be used in a core-based flow is to use an *N1*-sized OPMISR+ block within the core and an *N2*-sized OPMISR+ block within the UDL with fanouts of *M1* and *M2* respectively. This results in *N1xM1* scan chains within the core and *N2xM2* scan chains within the UDL. These can be connected to *N1+N2* scan I/O ports at the top-level of the design. If the scan I/Os are bi-directional then there can be *(N1+N2)/2* scan in and scan out ports each. If there are only uni-directional scan I/Os, then there will be *(N1 + N2)* scan in and scan out ports respectively.

Currently there is no support for automatically connecting the top-level test access mechanism (TAM) to the embedded OPMISR+ blocks

Figure A-25 One OPMISR+ Embedded Within a core and Another Within the UDL



Commands for Block Level Test Compression

The RTL Compiler command `compress_scan_chains` creates scan chains beginning and ending at the module boundary of the selected block. The command also supports embedding the OPMISR+ logic within a block or core. Additionally, the command supports the configuration, insertion, and connection of an embedded OPMISR+ block and creates scan channels starting from the `SWBOX_SI` pins of the OPMISR+ module and terminating at the `SWBOX_SO` pins of the OPMISR+ module. Refer to the following in *Design For Test in Encounter RTL Compiler* for details.

- “Inserting Scan Compression Logic”
- “Reducing Pin Count for Compression”
- “Manually Inserting a Scan Compression Macro

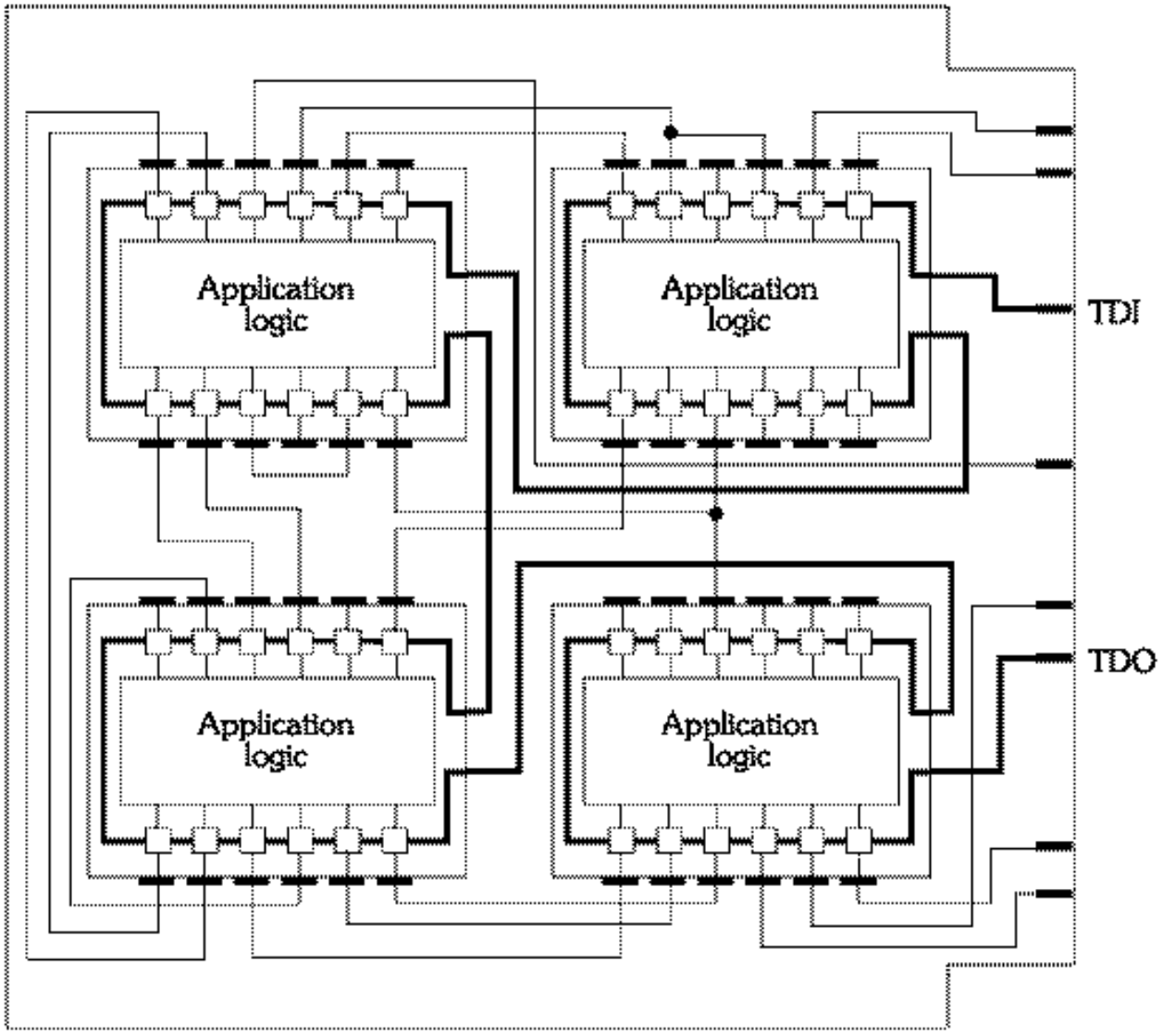
1149.1 Test Mode

IEEE 1149.1 is the standard for boundary scan implementations. You design the core logic, and the boundary scan is inserted along the periphery. Additional reference information is contained in *Supplement (B) to Standard Test Access Port and Boundary Scan Architecture, IEEE Std. 1149.1-2001*.

The JTAG macro inserted should conform to the IEEE 1149.1 1994 Standard. To observe and control the functional I/O's of a chip, independent of arbitrary system logic, Test Synthesis creates a Boundary Register. The Boundary Register is comprised of boundary scan cells inserted between each chip I/O and the system logic. By shifting values in and out of the boundary register, via a standard 5 pin Test Access Port (TAP), you can observe the value of each chip input and control the value of each chip output. This facilitates board level interconnect testing independent of on-chip system logic.

If multiple boundary scan devices exist on a card, you can connect all Boundary Registers into one large scan chain as shown in Figure A-26 on page 146. By scanning appropriate test data into, and capturing results out of this scan chain, the interconnections between the devices can be tested. For additional information, see the IEEE Standard 1149.1, "IEEE Standard Test Access Port and Boundary-scan Architecture".

Figure A-26 Scan Chain with Multiple Boundary Scan Devices



IEEE 1149.1 Boundary Scan Features

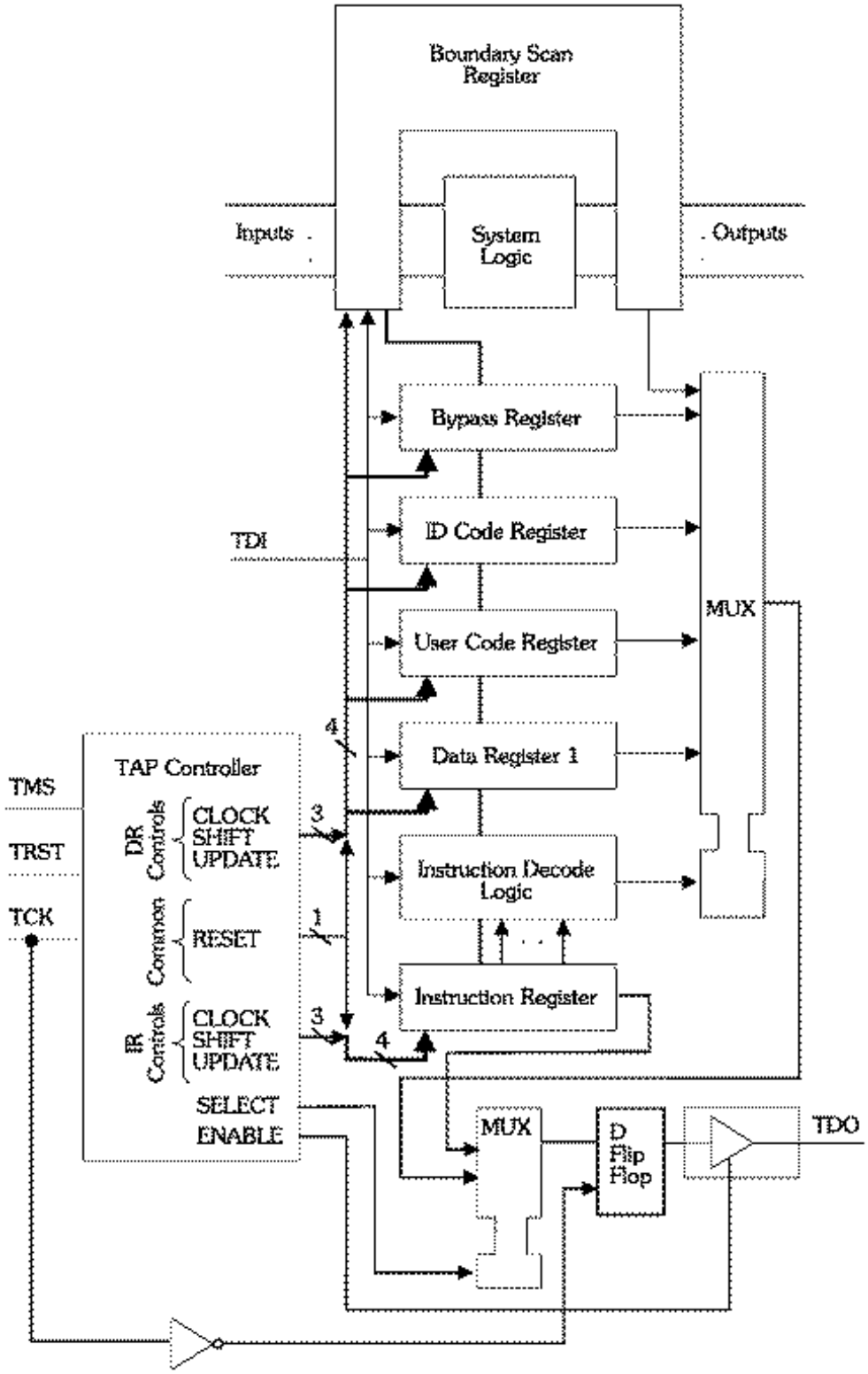
In compliance with the IEEE 1149.1 standard, Encounter Test supports the following mandatory features:

- Five pin TAP interface
- TAP Controller (which controls the boundary scan architecture)
- Instruction Register

- Instruction Decode Logic
- Bypass Register
- Boundary Register
- Test Data Output (TDO) logic

Encounter Test also supports any number of optional data registers for control of other special test structures. An example boundary scan architecture showing the mandatory design units, both IDCODE and USERCODE registers, as well as a single optional test data register called “Data Register 1” can be seen in Figure A-27 on page 148.

Figure A-27 IEEE 1149.1 Boundary Scan Architecture



In Figure A-27 on page 148, the ENABLE line from the TAP Controller enables the output driver of the TDO line. The SELECT line switches a multiplexor to select between the Instruction Register and the bank of data registers that includes the Bypass Register and Boundary Register. Each data register is selected through the Instruction Decode Logic and the data register multiplexor. All registers use some subset of the DR, Common, and IR Control lines from the TAP Controller.

The inputs to the TAP Controller are the Test Data Input (TDI), Test Clock (TCK), Test Mode Select (TMS), and an optional Test Reset (TRST). The output is the Test Data Output (TDO). All test data is shifted from the TDI input to the TDO output through any one of the device's registers, as selected by the current instruction.

When testing is performed, an instruction is serially loaded into each boundary scan device on a card. Depending on the instruction, the Instruction Decode Logic selects which register to connect between each device's TDI and TDO pins. Only one data register can be selected at a time, however different instructions may select the same register. For example, the EXTEST, SAMPLE, and INTEST instructions all select the Boundary Register. See the IEEE 1149.1 specification for further details on these instructions.

After the instruction is loaded, the test data is shifted in via TDI, and the test instruction is executed. Upon completion the results can be obtained by shifting out the test data through the TDO output. Once a test instruction is loaded, you can perform multiple tests by repeatedly shifting test data through the TDI-TDO path.

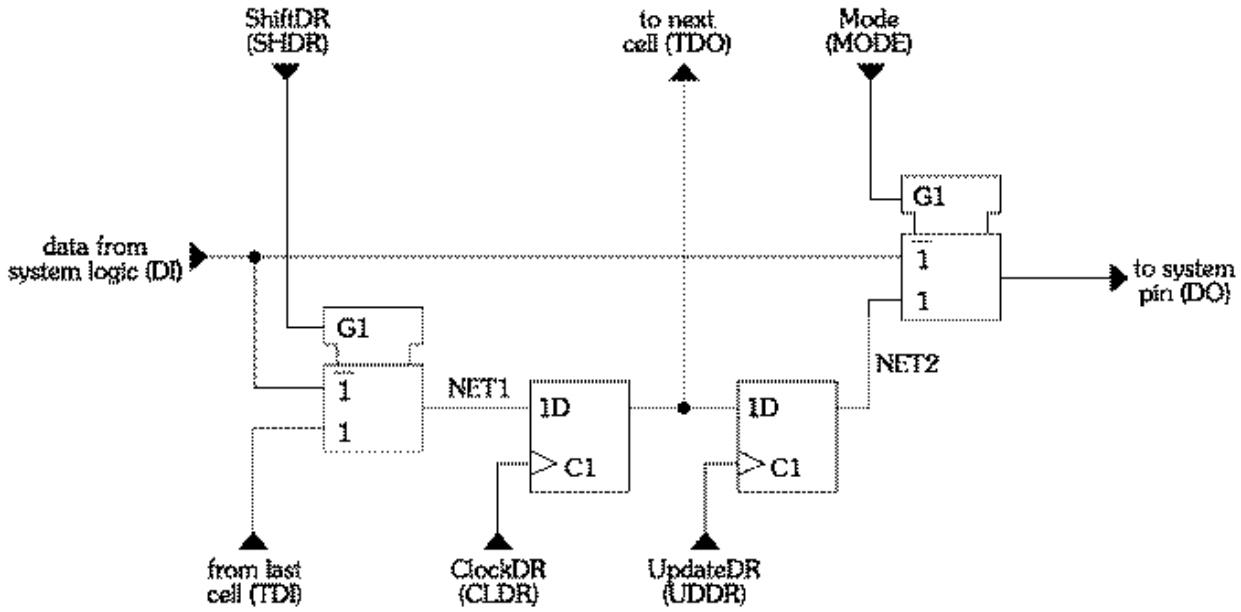
Depending on the value of the TMS input, the state of the TAP Controller is changed on each rising edge of the TCK. (The states and transitions of the TAP Controller are defined by the IEEE 1149.1 Standard.) If the TAP Controller inputs include the optional TRST input, it can be asynchronously reset to the Test-Logic-Reset state; otherwise, it can be synchronously reset to the Test-Logic-Reset state by holding the TMS input high and clocking through at least five cycles of TCK.

Boundary Register

A Boundary Register is typically made up of boundary scan flops and multiplexers which are configured differently for different 1149.1 instructions. Figure A-28 on page 150 shows a typical boundary cell .This boundary cell contains 2 storage elements known as the capture and update registers. The Capture element participates in shift operations for load/unload of the boundary register and is connected between TDI and TDO (through other boundary registers). The Update element is used to freeze a boundary cell output value such that

downstream logic can be isolated from the effects of switching capture data during load/ unload operations. The two multiplexers in this figure are used to select between shift data (from/to other boundary cells) and functional data (from/to the users functional design).

Figure A-28 Typical Boundary Scan Structure



Note: Test Synthesis requires appropriate synthesis rules for the boundary scan structures.

Boundary Scan Design Requirements

The most commonly recognized requirements for boundary scan design are described in the IEEE 1149.1 Standard. However, two additional common test methodologies place additional requirements on the boundary scan: Reduced Pin Count Testing, and Logic BIST.

Reduced Pin Count Testing (RPCT)

This is a test methodology being commonly adopted for chip manufacturing scan chain test, particularly for larger chips. The I/O signal pins are divided into two groups: test pins, which are contacted by full-function tester channels; and non-test pins, which are not contacted at all during wafer test and may be contacted by inexpensive parametric test channels during module test after packaging is complete.

Clearly, an I/O pin that is not contacted by a full function tester channel cannot be tested in the normal way with digital test patterns. This is commonly resolved by what is called I/O

Wrap (IOW). All such non-test pins are provisioned with bidirectional driver/receiver designs, even if the I/O is functionally input or output only. Boundary scan cells, accessible by the tester during normal scan chain test, are used to drive patterns out the driver and then to receive the signals back through the receiver without having to use tester facilities.

Logic BIST

Logic BIST is designed to take advantage of the boundary scan cells. The boundary scan chain will be clocked with the rest of the STUMPS channels, and will be used to drive all inputs to, and observe all outputs from, the functional logic. Boundary cells on inputs should not capture data, and data scanned into boundary cells on outputs should not be visible at the pads. This requires different designs for input and output boundary cells. An input to logic that does not have the boundary scan cell controlling it can be tolerated only if the input is held to a fixed, known, value at all times during logic BIST.

In the IEEE 1149.1 standard, some of the example boundary cells, for example BC-1, are described as supporting the commands INTEST and RUNBIST, and other boundary cells, for example BC-2, are described as not supporting INTEST and RUNBIST.

Refer to "Logic Built-In Self Test (LBIST) Generation" in the *Encounter Test: Guide 5: ATPG* for additional information.

1149.1 Test Mode Input Files

- BSDL file (optional)

  BSDL (Boundary Scan Design Language) is recommended by the IEEE 1149.1 standard to specify boundary scan. Refer to "Boundary Scan Design Language" on page 301 for details.

- Package File(s)

  A package file is an ASCII file which describes boundary scan cell definitions. The package file is in the form of a standard VHDL package file definition.

  One package file must be provided for each package reference in the BSDL file "USE" statement. This normally includes an IEEE 1149.1 Standard package file (for example, STD_1149_1_1994) and zero or more user defined package files (for example, IBMDFT_1149_1_1998_V5).

  The Standard package files are shipped with Encounter Test (`$Install_dir/defaults/bsdl`) while the user-defined package file(s) are provided by the technology provider or design team, in the case of custom boundary scan cell designs.

For an example, refer to the package file that was shipped with Encounter Test in directory `$Install_dir/defaults/bsdl`.

1149.6 Test Mode

IEEE 1149.6 is the standard that addresses interconnect testing between chips. Two areas are focused upon: differential circuits and AC coupled circuits. Additional reference information is contained in *"IEEE Standard for Boundary-Scan Testing of Advanced Digital Networks", IEEE Std. 1149.6 -2003*.

The IEEE 1149.1 boundary scan standard addressed the problem of static testing of interconnects between chips on a board. Two topics that were not completely addressed by the 1149.1 standard are the following:

1. Differential Circuits

   The 1149.1 standard models differential circuits as single ended circuits ignoring a large percentage of the potential defects that may occur on such a circuit.

2. AC Coupled Circuits

   These are capacitatively coupled interconnects that do not permit static signals. This is due to the difficulty in switching an output at much greater than 100 KHz using the capabilities provided by the 1149.1 standard, which is virtually a DC level with respect to a typical AC Coupled circuit.

The 1149.6 standard addresses both limitations. The 1149.6 standard requires that both legs of a differential input are capable of being independently monitored. It also addresses passing signals through capacitors by providing the following new capabilities.

1. Generation of Pulses of a period equal to one period of the Test Clock (`TCK`) input to the TAP controller (the new `EXTEST_PULSE` instruction).
2. Generation of Trains of Pulses whose frequency is one half of the Test Clock (`TCK`) input to the TAP controller (the new `EXTEST_TRAIN` instruction).
3. Definition of a test receiver that is able to detect the signals received by an AC coupled receiver in the presence of a Pulse or Pulse Train and reconstruct the original signal.

The 1149.6 Test Receiver

The key to measuring AC signals in the 1149.6 standard is the Test Receiver. The truth table for the Test Receiver is shown in Table A-6 on page 153. This cell is a storage element that stores a one whenever an input signal (`A`) is above a certain limit (`VHIGH`) and stores a zero

whenever the input signal (A) is below another limit (VLOW). When the input signal (A) is between VHIGH and VLOW, the state can be set using a data signal (PD) and a clock (PC). To detect for the input signal (A) exceeding the upper threshold, VHIGH, the state of the Test Receiver is first set to zero using the PD and PC signals. A pulse is transmitted to the input (A) and the output of the Test Receiver is examined. If the state is one, the pulse passed the threshold. If it remains a zero, the pulse never passed the threshold. A similar test can be used to detect for a pulse which pulls the input signal below VLOW. In this case, PD and PC are used to first set the state of the Test Receiver to a one. During AC testing in 1149.6, failure to detect such pulses is typically considered a fault in the circuit.

Table A-6 Test Receiver Truth Table

Input Signal (A)	PC	PD	OUTPUT (Z)
$A > V_{HIGH}$	X	X	1
$A < V_{LOW}$	X	X	0
$VLOW < A < V_{HIGH}$	0	X	Previous state
$VLOW < A < V_{HIGH}$	1 (or rising)	0	0
$VLOW < A < V_{HIGH}$	1 (or rising)	1	1

Test Receivers are typically employed on the inputs of circuits to measure AC signals arriving at the chip inputs. By contrast, the capability to transmit AC signals on the chip outputs is achieved by a slight addition to the boundary cell logic. Figure A-29 on page 154 shows the configuration of the test receiver and how it can be observed by a boundary cell.

Figure A-29 Test Receiver Configured with Input Boundary Cells for Observability and Controllability

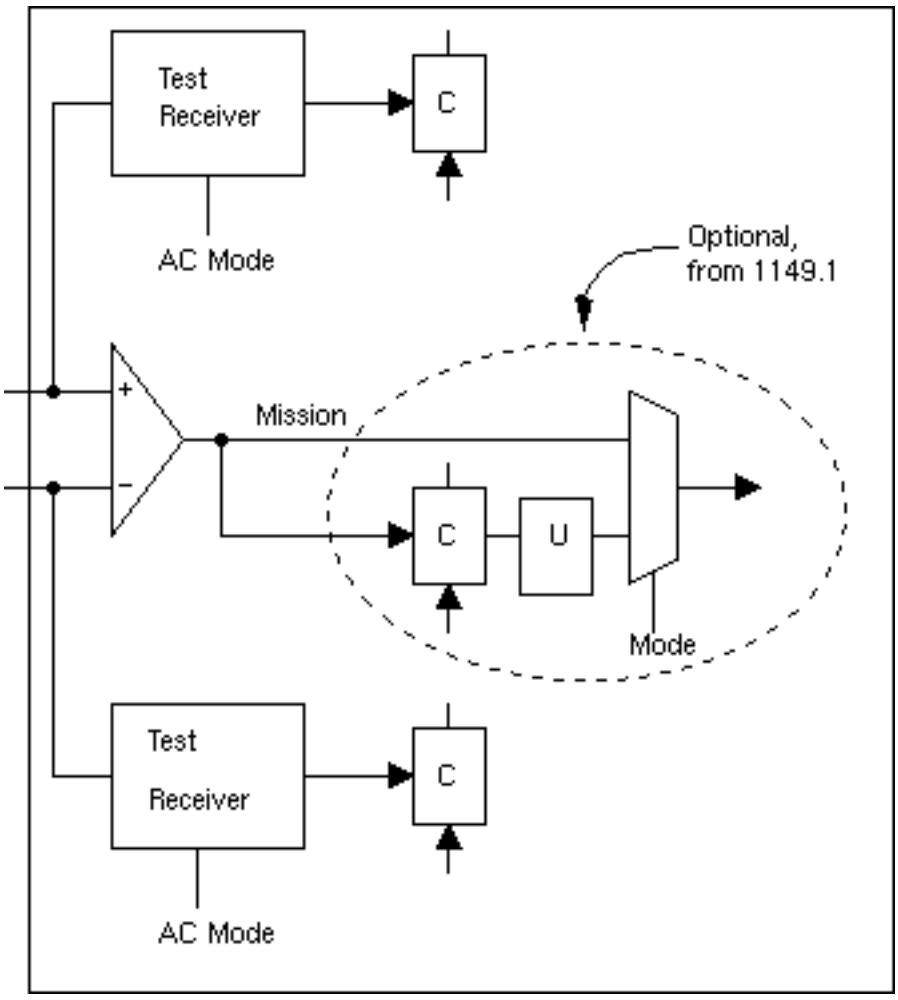
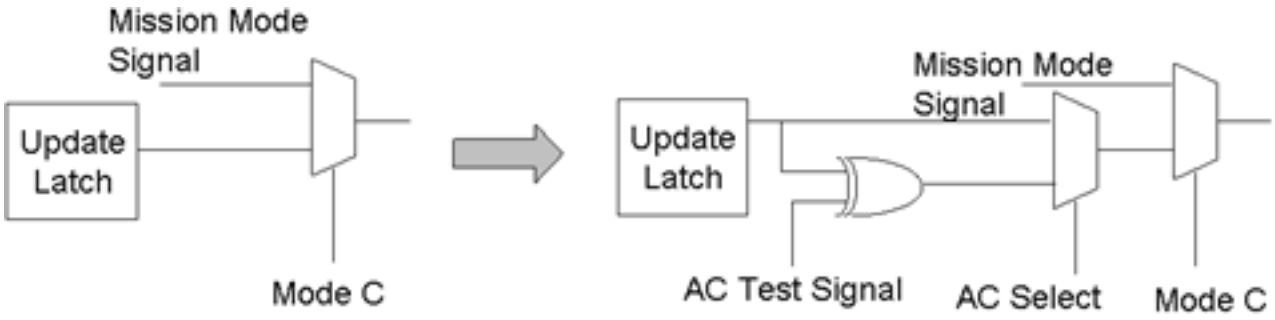


Figure Figure A-30 on page 155 shows the modification to the boundary cell's output logic. As in the 1149.1 standard, a Mode signal is used to switch the boundary cell output between mission mode and the data in the update latch. Two additional signals have been added to control behavior for AC signals. First, the AC mode signal selects the DC behavior and the new 1149.6 AC behavior. When the AC mode signal is selected, the second new signal, the AC Test Signal, is used to toggle the output of the boundary cell.

Figure A-30 Changes to Output Boundary Scan Cells



Support for the Level Sensitive Test Receiver

The preceding description is for the edge-sensitive test receiver. IEEE Standard 1149.6-2003 was written assuming that the initialization of 1149.6 (ac) test receivers would be accomplished at a particular clock edge, and the examples in the standard are based on an assumption that test receivers are initialized in response to a rising transition of a preload clock. However, in practice, many 1149.6 (ac) test receivers are initialized by a level-sensitive enable signal; as long as the enable signal is active, the test receiver is held in its initialized state.

The 1149.6 `JTAG_MACRO` supports both test receivers with edge-sensitive initialization and those with level-sensitive initialization. Specifically, the `JTAG_MACRO` output `JTAG_ACPSCLK` is created to supply a positive-active edge-sensitive clock signal to test receivers that have edge-sensitive initialization. The `JTAG_MACRO` output `JTAG_ACPSEN` is created to supply a positive-active level-sensitive enable signal to test receivers that have level-sensitive initialization. Furthermore, the `dft_pin_function` attribute `TR_PEN` is used to identify a pin on a test receiver cell that is initialized by a high level. The `dft_pin_function` attribute `TR_PC` is used to identify a pin on a test receiver that is initialized by a rising transition.

Changes to the 1149.6 TAP Controller

Support of the 1149.6 standard requires very few changes to the TAP controller described in the 1149.1 standard, These changes are intended to be upward compatible to the 1149.1 specification in order to permit 1149.1 to perform DC testing on 1149.6 compliant chips. The following are the changes:

- Introduction of the `EXTEST_TRAIN` and `EXTEST_PULSE` instructions. These instructions must have unique opcodes but must cause the Boundary Scan Register to operate in a manner that is identical to the `EXTEST` instruction described in the 1149.1 standard. This implies that the existing boundary register mode control outputs of the 1149.1 controller

must behave in the same manner for these new instructions as they do for the 1149.1 EXTEST.

- Addition of an output (referred to in the standard as "AC Test Signal") which is used by the boundary scan cells of all AC Coupled output pins.
- Addition of a clock output used to preset the storage circuitry that is part of the hysteretic comparator in the Test Receiver cell. Addition of an output which is the logical OR of the decode of the EXTEST_TRAIN and EXTEST_PULSE instructions. The output may either directly drive or be ANDed with the output of an AC/DC selection cell to drive the AC Mode input of a boundary scan cell which supports AC Test.

Figure A-31 Generation of an AC Test Signal

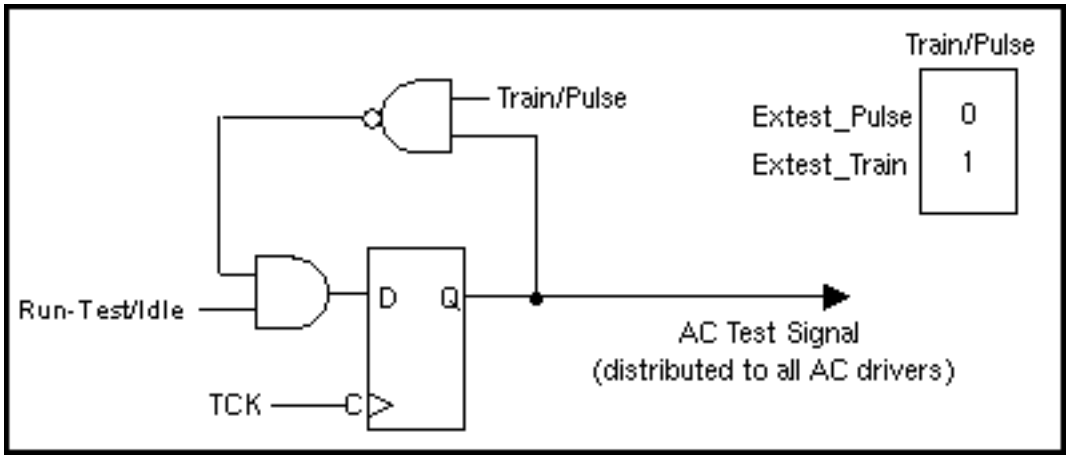


Figure A-32 Generation of an Initialization Clock

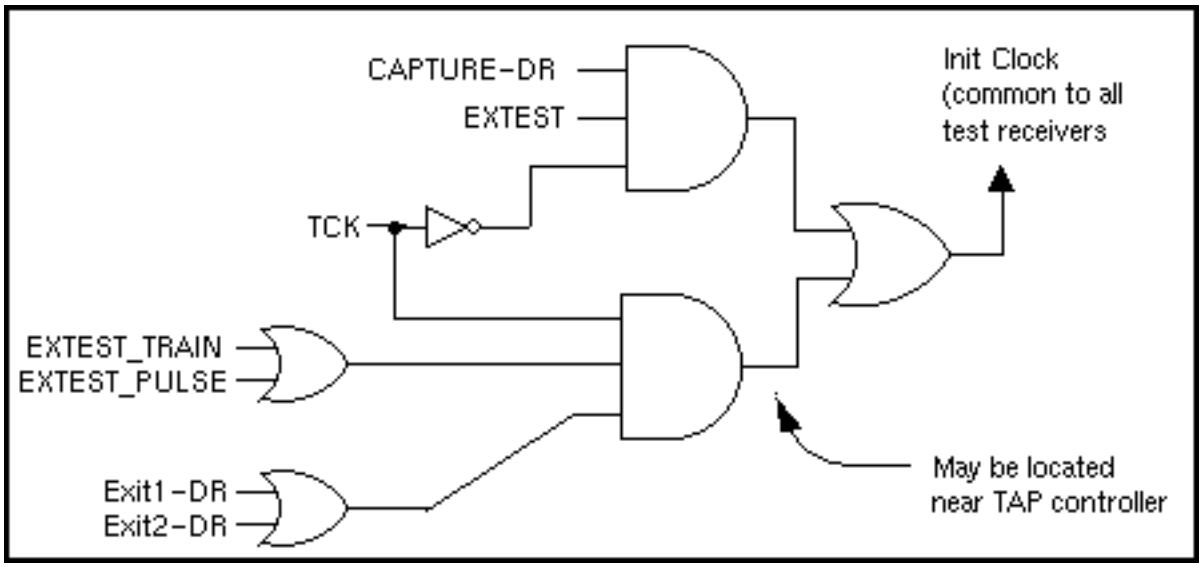
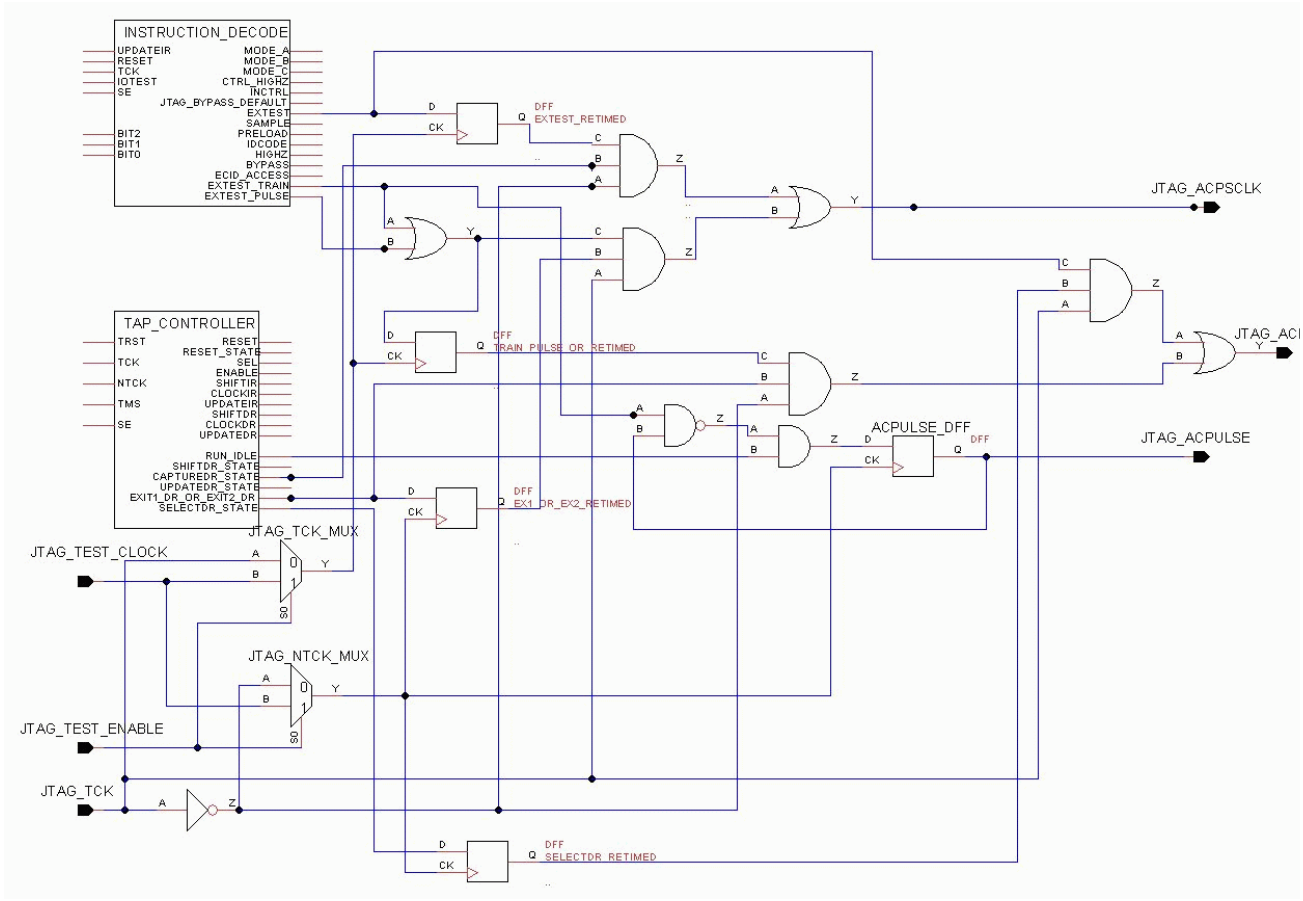


Figure A-33 JTAG_ACPSCLK, JTAG_ACPSEN, and JTAG_ACPULSE Implementation Details



Given the preceding requirements, modify the 1149.1 controller logic (called the JTAG_MACRO if inserted by Test Synthesis) by adding the following four output ports:

1. JTAG_ACDCSEL

   The logical OR of the EXTEST_PULSE and EXTEST_TRAIN being decoded. This should be implicitly connected to the AC pin on all Test Receivers. This signal should also control the multiplexer which is added to the output Boundary Scan cells.

2. JTAG_ACPSCLK

   The Init clock signal shown in Figure A-33 is an example implementation. This signal should be implicitly connected to the PC pin of all test receivers.

3. JTAG_ACPSEN

   The JTAG_ACPSEN signal shown in Figure A-33 on page 158 is an example implementation. This signal should be implicitly connected to the PEN pin of all level sensitive test receivers.

4. JTAG_ACPULSE

   The AC Test Signal output shown in Figure A-31 on page 156. This signal should be implicitly connected to one input of the XOR gate which is added to the Boundary Scan Cell.

5. JTAG_ACTRENBL

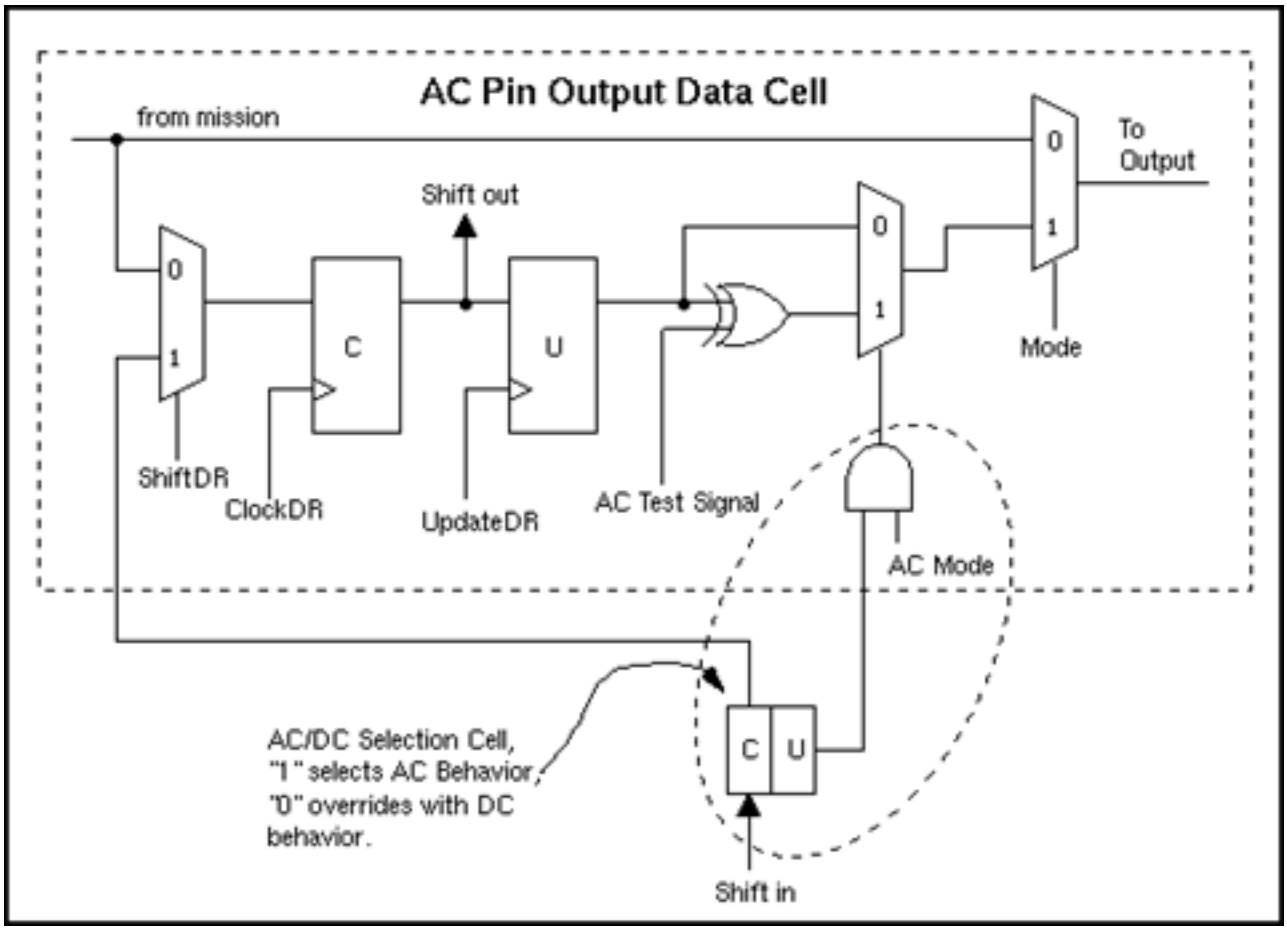
   Drives the RD pin of the Test Receiver if present. This signal is the logical OR of EXTEST, EXTEST_PULSE and EXTEST_TRAIN and any other instruction which expects the Test Receiver to be enabled.

Changes to the Output Boundary Scan Cell

Boundary Scan Cells which drive system outputs in support of the 1149.1 specification normally contain logic similar to the logic shown on the left of Figure A-30 on page 155. To support the requirements of the IEEE 1149.6 standard, the Boundary Scan Cell must be augmented as shown in Figure A-30 on page 155 on the right for all output cells which are to support AC Testing. The AC Mode signal is driven by the TAP controller or by the logic AND of the TAP controller and a boundary scan cell. The AC Test Signal is generated by the logic shown in Figure A-31 on page 156. The 1149.6 standard requires that the coding of the BSDL (Boundary Scan Description Language) specification of the AC Mode signal be done in a manner to cause an 1149.1 compliant tool to set it to a zero value, leaving the Boundary Scan logic effectively unchanged.

An example implementation of an augmented output boundary cell is shown in Figure A-34 on page 160.

Figure A-34 Augmented Output Boundary Cell with 1149.6 Support



IEEE 11496 Boundary Cells

Table A-7 lists the names of the `IEEE_11496` boundary cells and describes their usage.

Table A-7 IEEE_11496 Boundary Cells and Usage

Cell Name	System Function	Test Function	Usage	Required Pins	Reference
`BC_11496_ACTR`	Input Clock Bidirectional	Any with appropriate sharing logic	Test I/O	`ADI`, `TDI`, `CLOCKDR`, `SHIFTDR`, `ADO`, `TDO`	Figure A-35 on page 163

Cell Name	System Function	Test Function	Usage	Required Pins	Reference
BC_11496_BIDIR	Bidirectional	Any with appropriate sharing logic	Test I/O or Non-Test I/O	OE, DDI, RDI, TDI, CLOCKDR, SHIFTDR, UPDATEDR, MODE_A, MODE_C, ACPULSE, ACDCSEL, DDO, RDO, TDO	Figure A-36 on page 164
BC_11496_BIDIR_TI	Bidirectional	Test input	Test I/O	OE, TE, DDI, RDI, TDI,, CLOCKDR, SHIFTDR, UPDATEDR, MODE_A, MODE_C, ACPULSE, ACDCSEL, DDO, RDO, TDO	Figure A-37 on page 165
BC_11496_BIDIR_TO	Bidirectional	Test Output	Test I/O	SE, OE, DSO, DDI, RDI, TDI, CLOCKDR, SHIFTDR, UPDATEDR, MODE_A, MODE_C, ACPULSE, ACDCSEL, DDO, RDO, TDO	Figure A-38 on page 166

Cell Name	System Function	Test Function	Usage	Required Pins	Reference
BC_11496_BIDIR_TO_OO	Bidirectional	Test output; also used for macro test observation	Test I/O	OE, DSO, OO, SE, MTOC, DDI, RDI, TDI, CLOCKDR, SHIFTDR, UPDATEDR, MODE_A, MODE_C, ACPULSE, ACDCSEL, DDO, RDO, TDO	Figure A-39 on page 167
BC_11496_OUT	Output2 Output3	Any with appropriate sharing logic	Test I/O	DDI, TDI, CLOCKDR, SHIFTDR, UPDATEDR, MODE_C, ACPULSE, ACDCSEL, DDO, TDO	Figure A-40 on page 168
BC_11496_OUT_NT	Output2 Output3	None	Non-Test I/O	RDI, DDI, TDI, CLOCKDR, SHIFTDR, UPDATEDR, MODE_C, ACPULSE, ACDCSEL, DDO, TDO	Figure A-41 on page 169
BC_11496_OUT_TI	Output2 Output	Test input	Test I/O	DDI, TDI, CLOCKDR, SHIFTDR, UPDATEDR, MODE_C, ACPULSE, ACDCSEL, DDO, TDO	Figure A-42 on page 170

Cell Name	System Function	Test Function	Usage	Required Pins	Reference
BC_11496_OUT_TO	Output2 Output3	Test output	Test I/O	SE, DSO, DDI, TDI, CLOCKDR, SHIFTDR, UPDATEDR, MODE_C, ACPULSE, ACDCSEL, DDO, TDO	Figure A-43 on page 171
BC_11496_OUT_TO_OO	Output2 Output3	Test output; also used for macro test observation	Test I/O	DSO, OO, SE, MTOC, DDI, TDI, CLOCKDR, SHIFTDR, UPDATEDR, MODE_C, ACPULSE, ACDCSEL, DDO, TDO	Figure A-44 on page 172

Figure A-35 BC_11496_ACTR

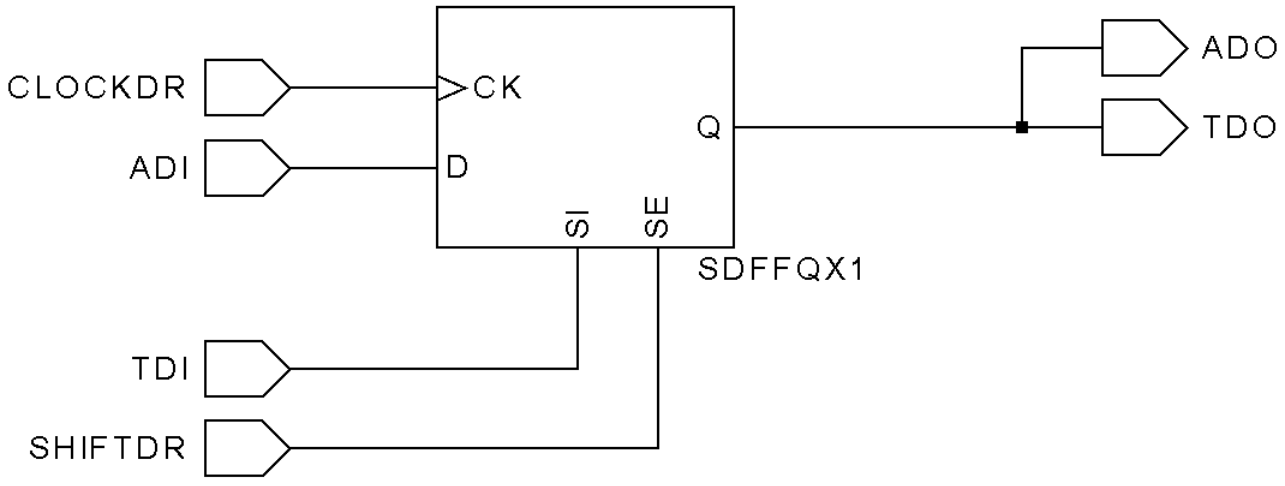


Figure A-36 BC_11496_BIDIR

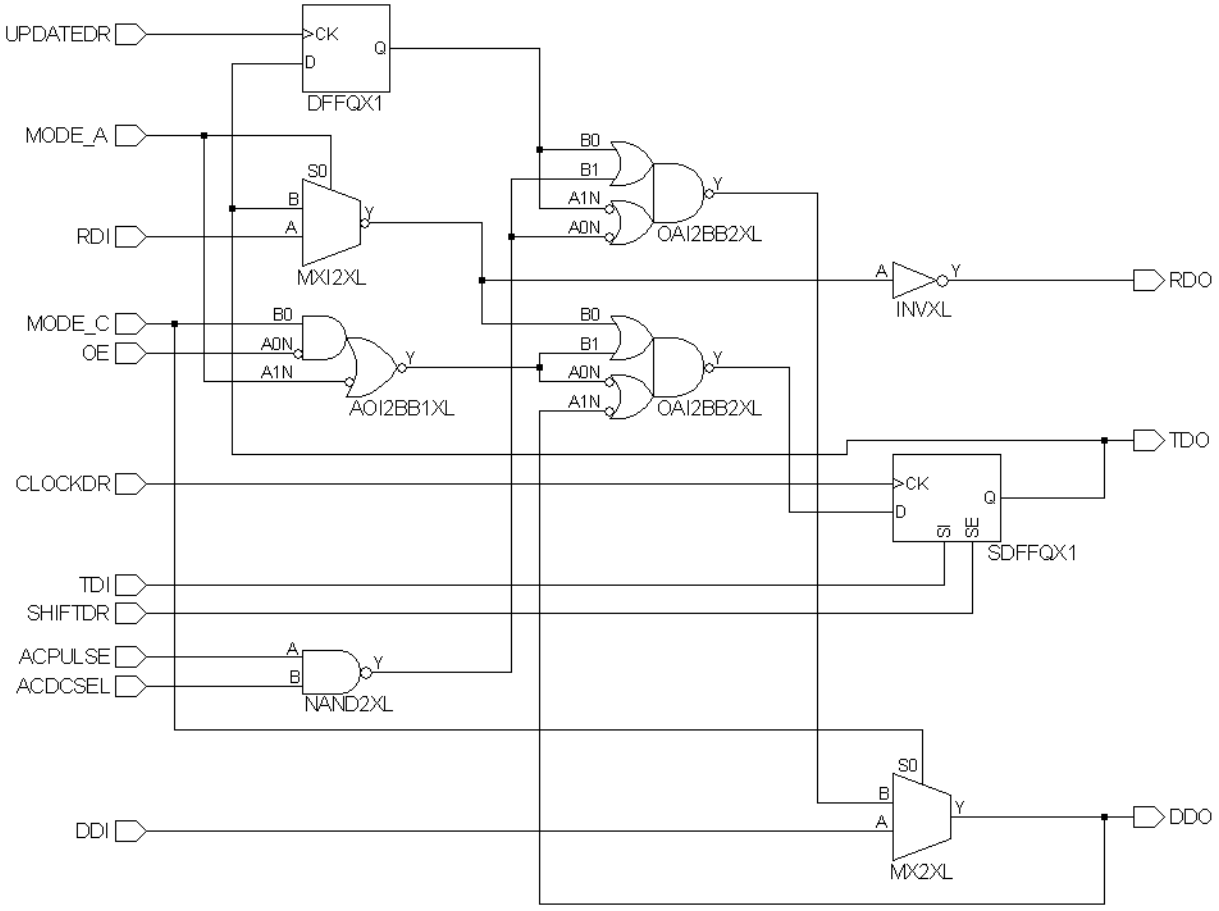


Figure A-37 BC_11496_BIDIR_TI

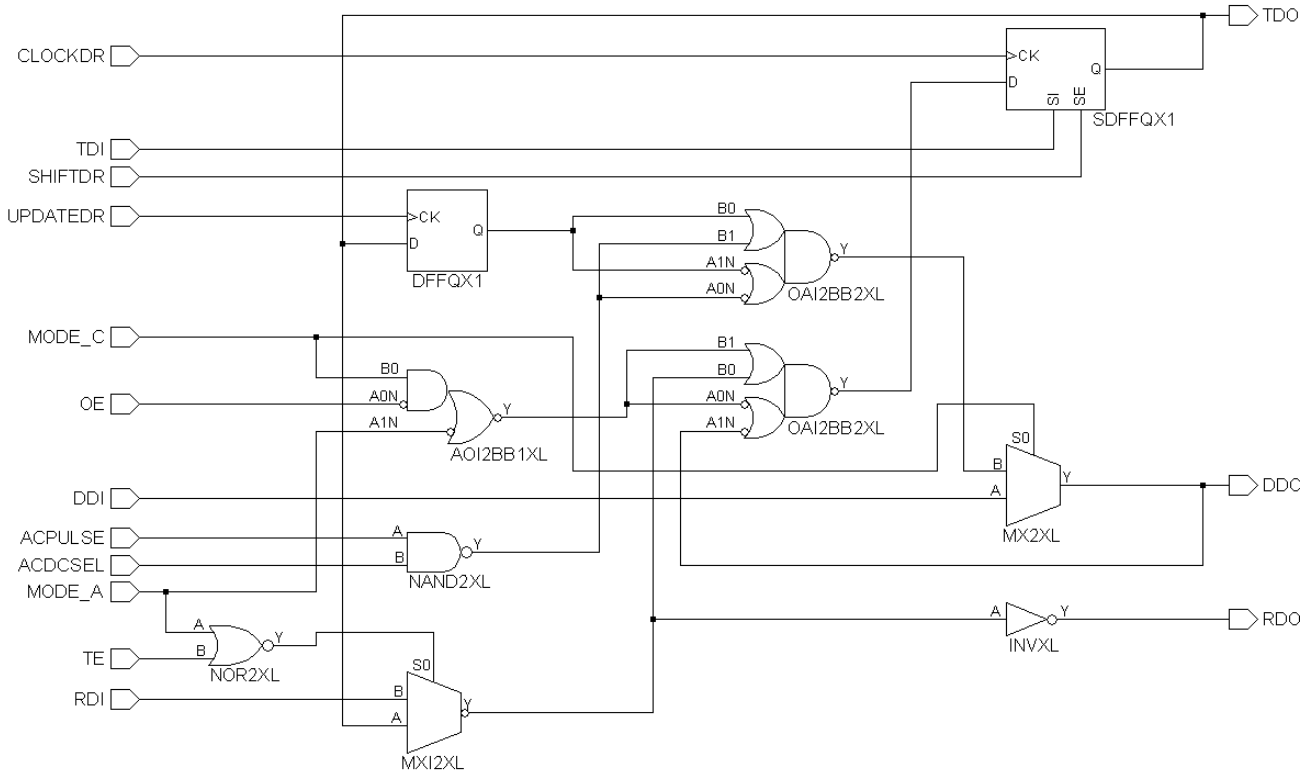


Figure A-38 BC_11496_BIDIR_TO

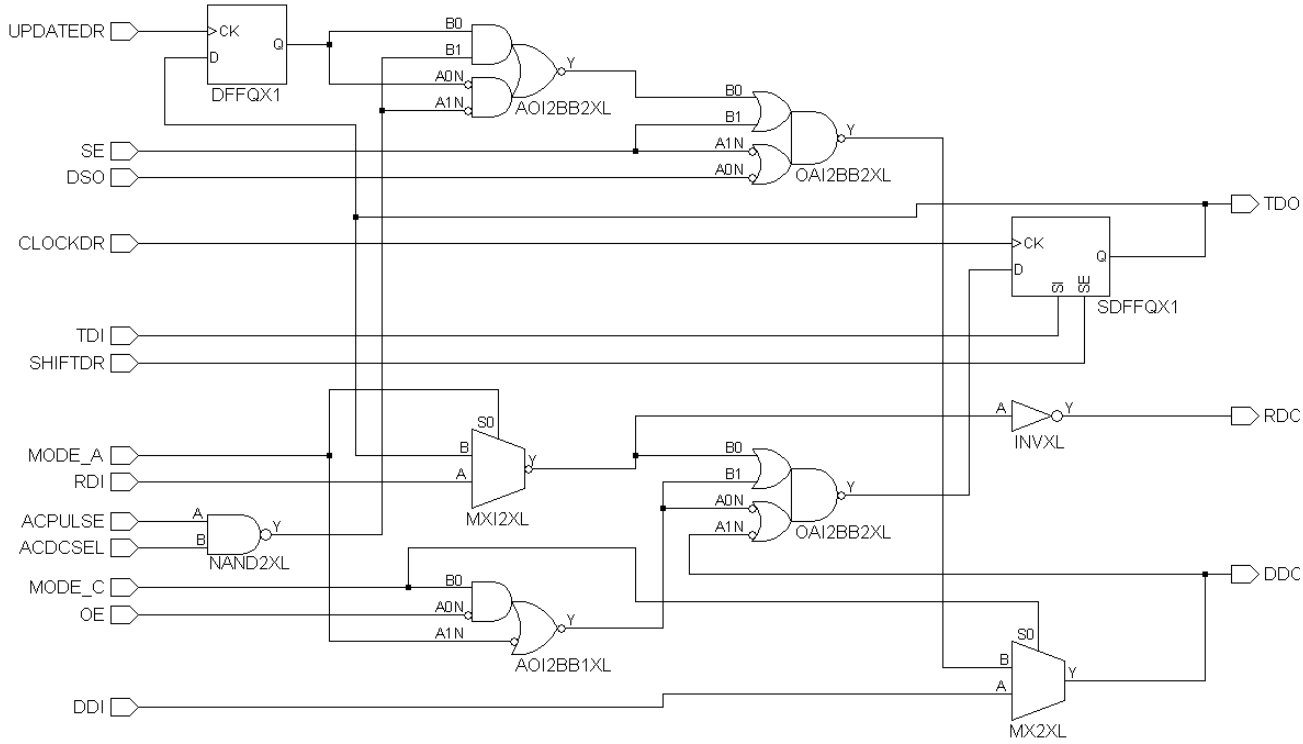


Figure A-39 BC_11496_BIDIR_TO_OO

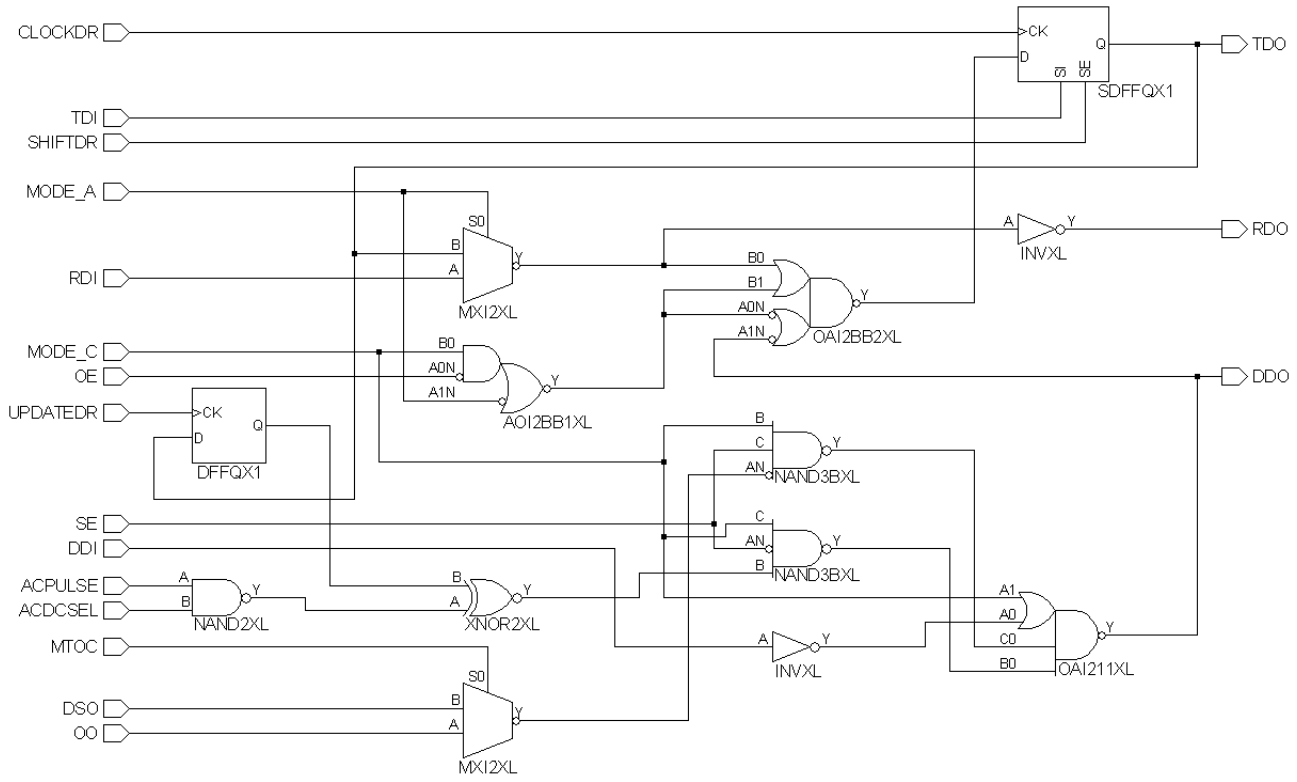


Figure A-40 BC_11496_OUT

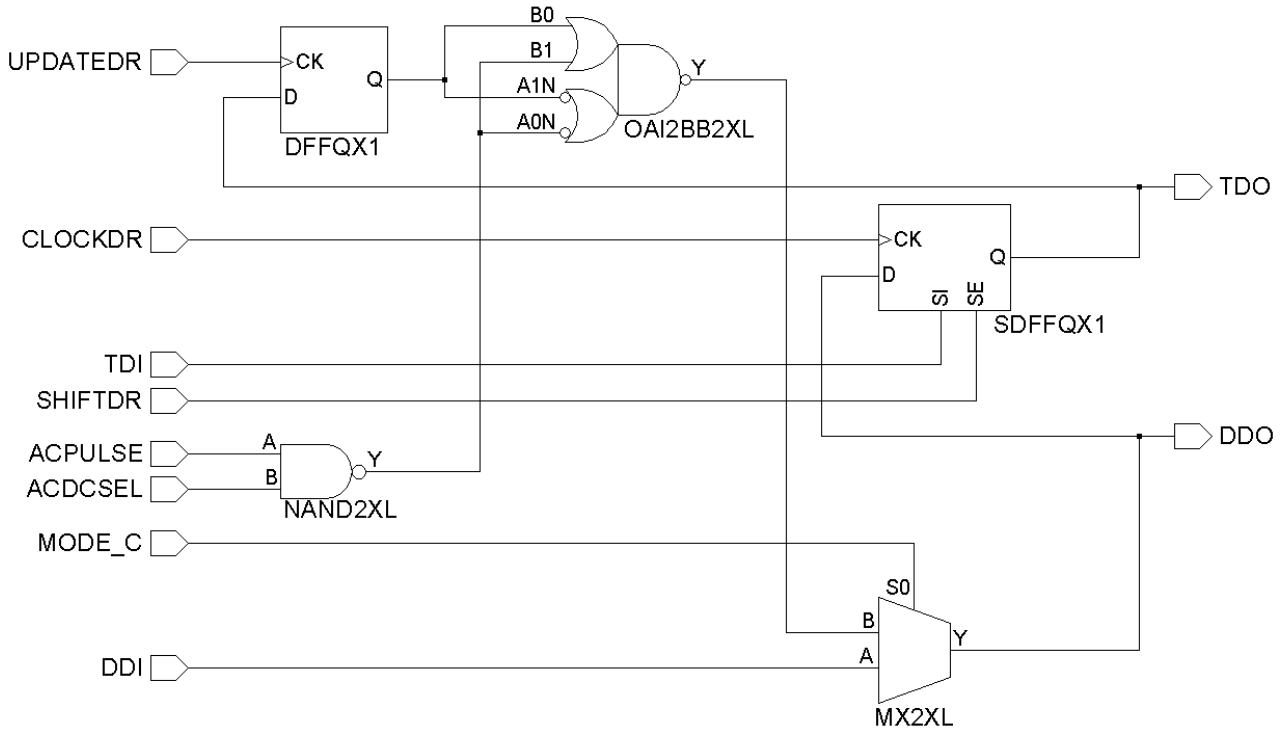


Figure A-41 BC_11496_OUT_NT

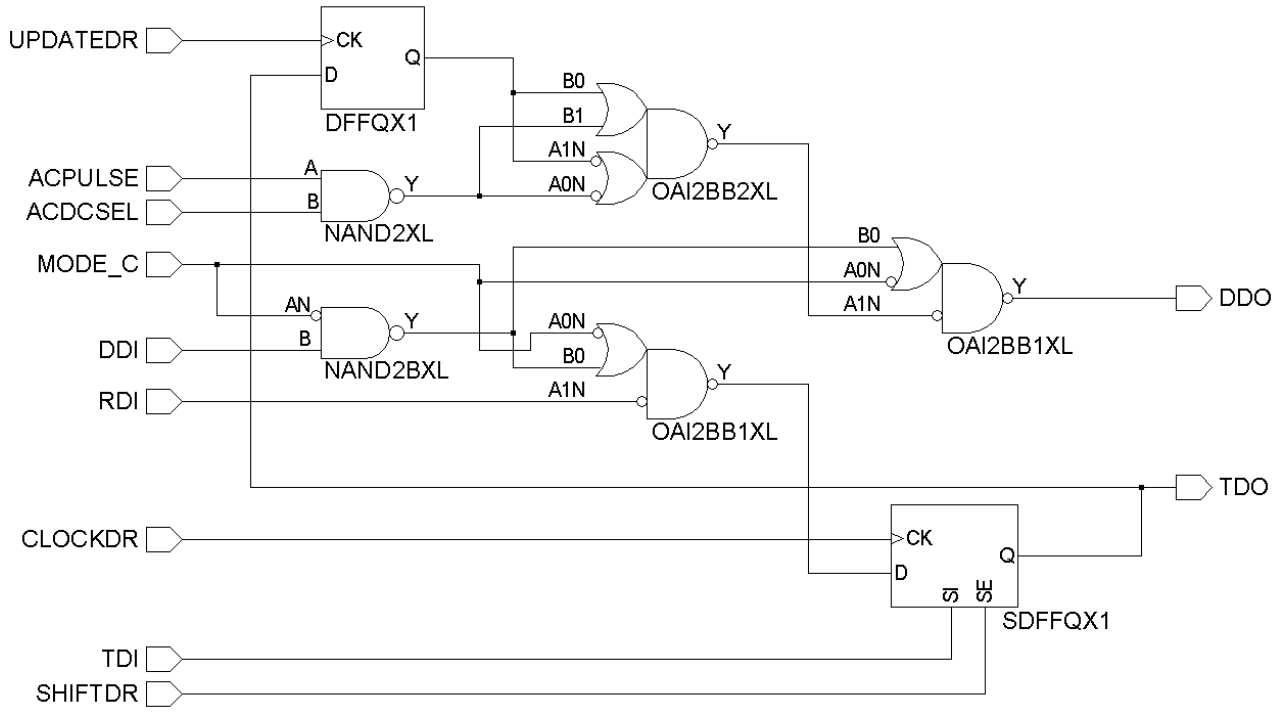


Figure A-42 BC_11496_OUT_TI

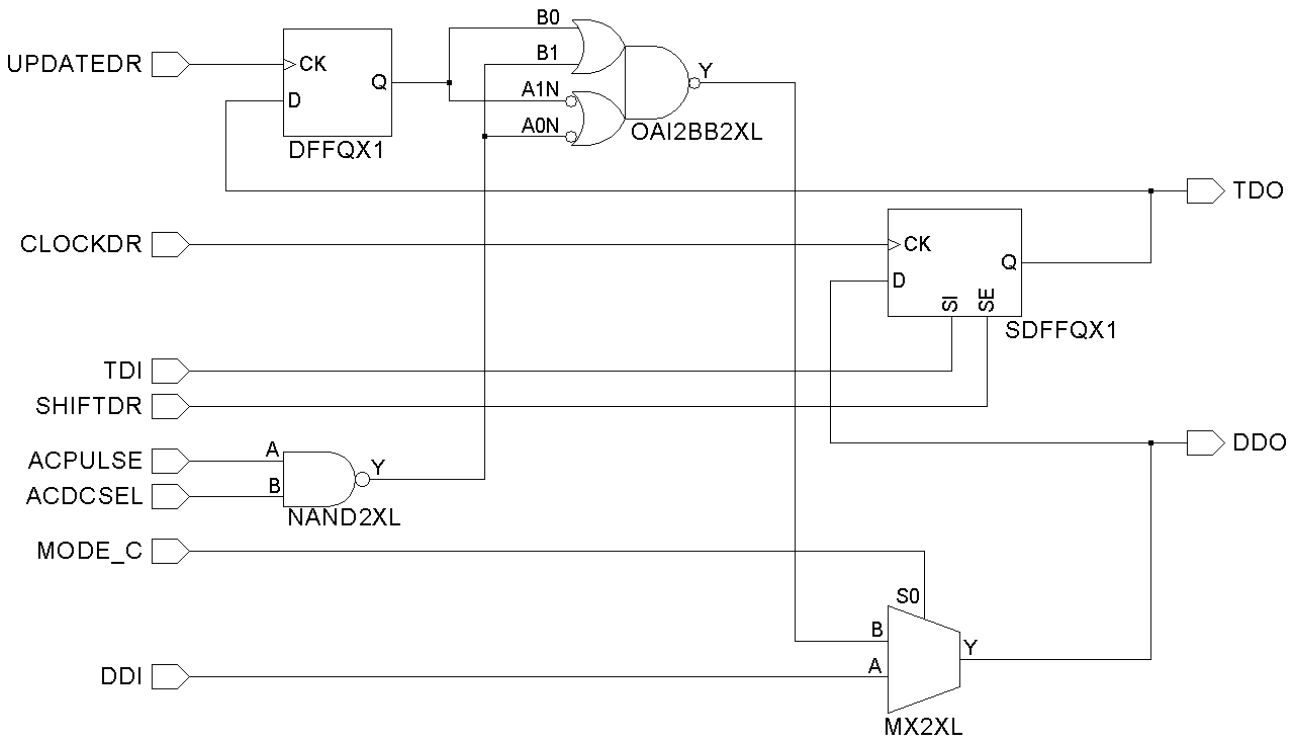


Figure A-43 BC_11496_OUT_TO

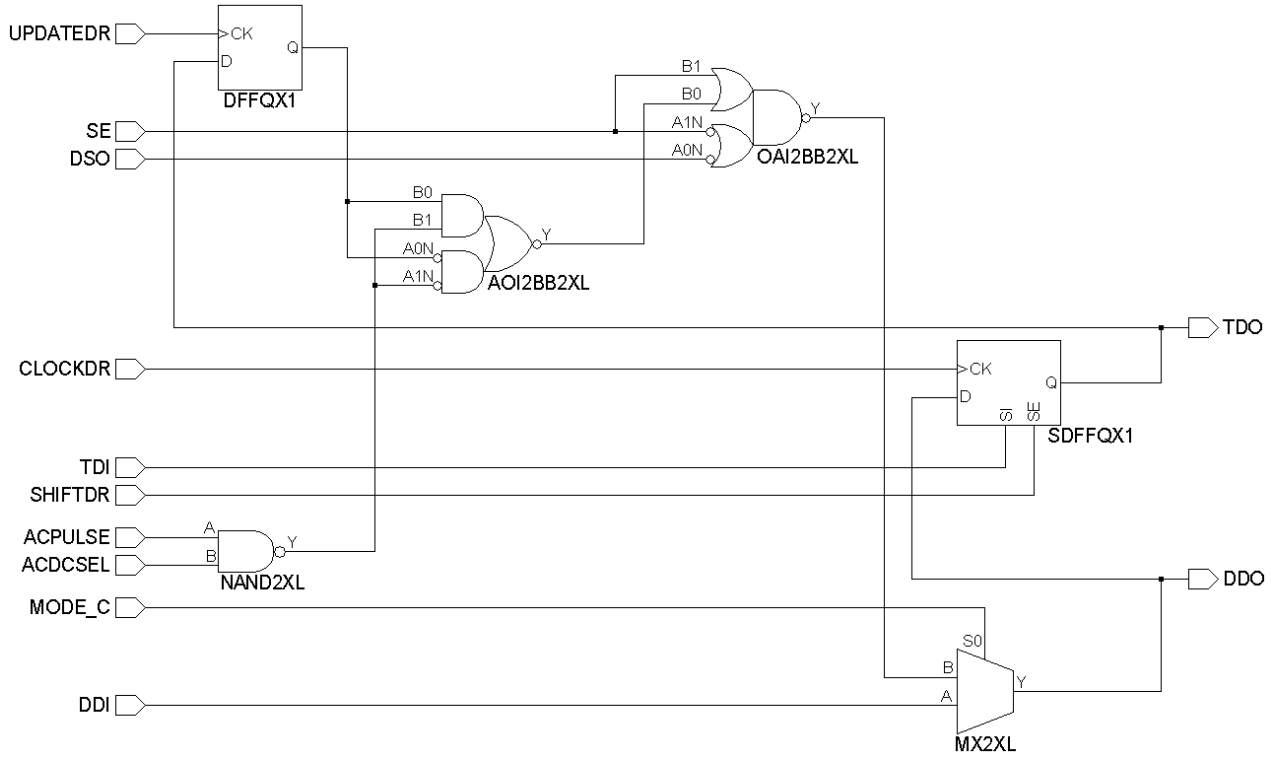
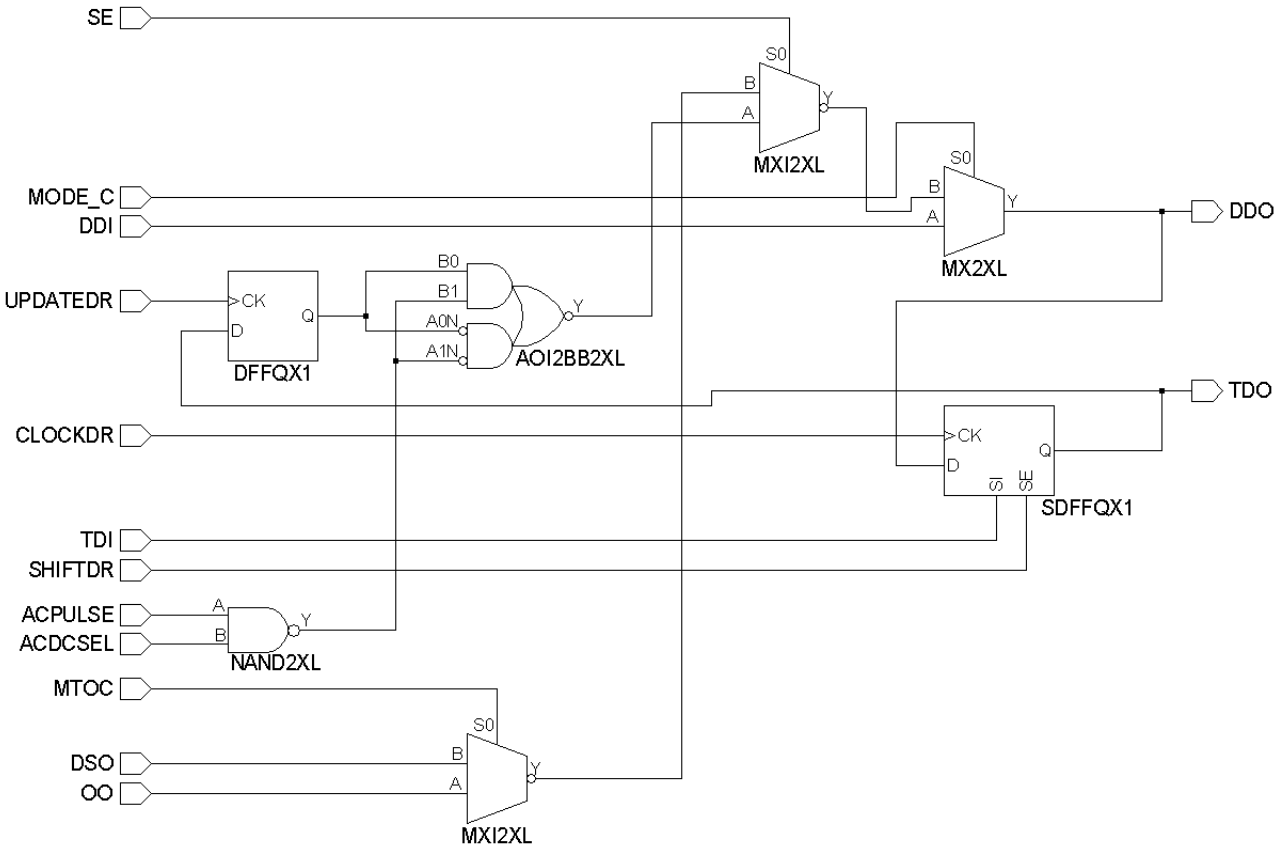


Figure A-44 BC_11496_OUT_TO_OO



Verifying the Boundary Scan Implementation

Due to the backward compatibility of the IEEE 1149.6 specification to IEEE 1149.1, a substantial portion of the validation can be performed by using the existing 1149.1 Boundary Scan Verification support. The selection of the boundary register during `EXTEST_PULSE` and `EXTEST_TRAIN` is also verified. Currently no explicit checks verify the pure 1149.6 extensions. The following lists the specific limitations:

1. Verification of the connection of `AC`, `PC`, `PD` and other 1149.6 control pins is not performed.
2. Verification of the additional JTAG TAP structures added in support of 1149.6 is not performed.

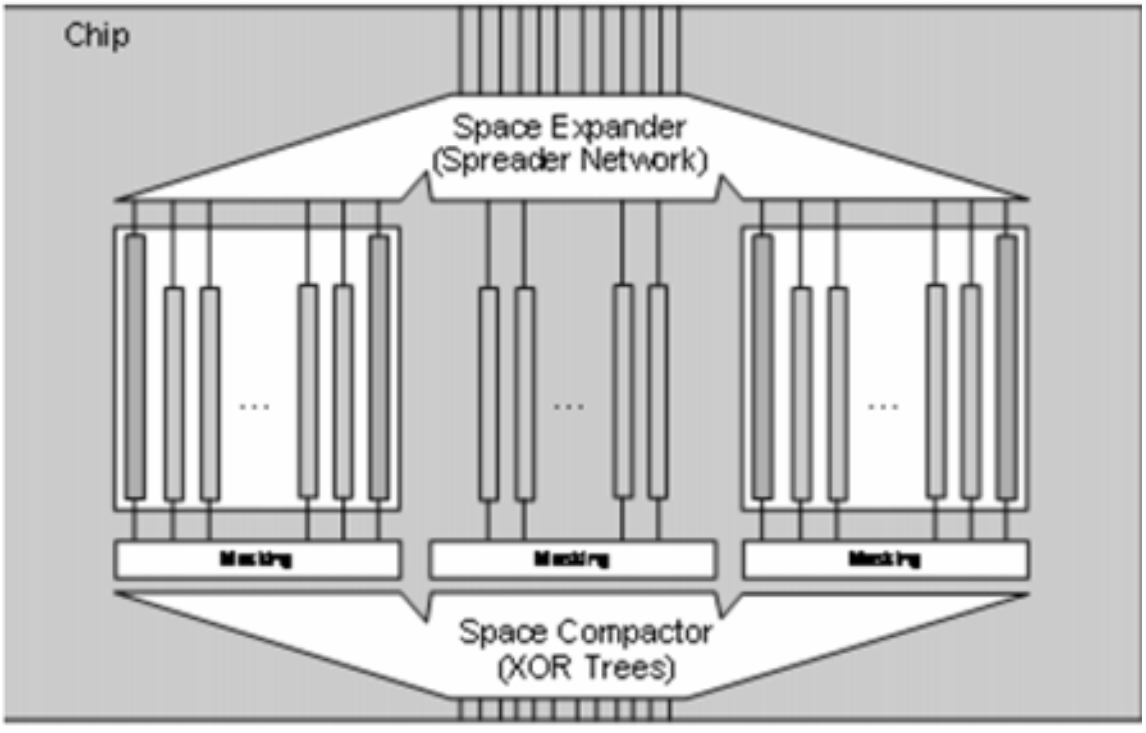
Refer to "Verify 1149.1 Boundary Scan" in the *Encounter Test: Guide 3: Test Structures* for details on how to verify the 1149.1 and 1149.6 implementation.

On-Product XOR Compression

Test Synthesis adds specialized circuitry, called a *Compression Macro,* which allows for reduction in scan chain lengths and leads to reduced test time and test data volume. XOR compression is an on-product test data compression method based on the use of combinational XOR structures. An XOR-tree compactor is used for test response compression (that is, scan-outs) and an optional XOR based test input (that is, scan-ins) spreader can be used for test input data decompression.

The following subsections describe an alternative structure for On-product test data compression based on the use of combinational XOR structures. An XOR-tree spreader is used for test response compression and an optional XOR based test input spreader can be used for test input data de-compression. Figure A-45 on page 173 shows a high-level diagram of the on-product XOR compression architecture. A stream of compressed test data from the tester is fed to the N scan input pins of the chip under test. A space expander based on an XOR based spreader network internally distributes the test data to a large number of internal scan channels which is a multiple (for example, M) of the number of scan input pins N. The input side test data spreader therefore feeds M*N scan channels.

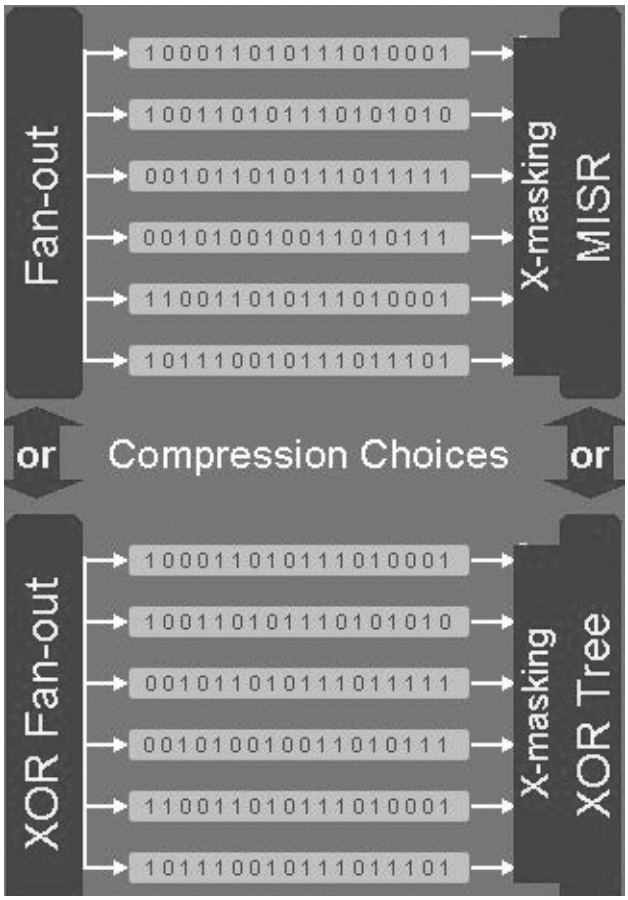
Figure A-45 On-Product Test Data Compression Architecture



On the output side, the test data response is compressed by a space compactor to create an N-wide output test response data stream. The space compactor is based on a combinational XOR-tree. Similar to the case of OPMISR and OPMISR+, optional X-masking logic can be added between the scan channel tails and the input to the XOR based space compactor. The masking logic is optional in the case of XOR-based compression since it has better tolerance for capturing X-states than a MISR however it is highly recommended. It is difficult to predict the number of X-states that may be captured on a *clean* design once it is fully implemented and put on a tester.

While the XOR-based space compactor is needed on the output side, a simpler input side decompressor based on scan fanout can also be used in this architecture. The four compression options are summarized in Figure A-46 . On the input side, the space expander can be based on either scan fanout or an XOR spreader. On the output side, the space compactor can be based on either a MISR or XOR tree. Encounter Test ATPG and Diagnostics support all four combinations shown in Figure A-46 . Test Synthesis does not currently support the combination of an XOR spreader with a MISR space compactor.

Figure A-46 Encounter Test Compression Options



The XOR Compression Macro

Table A-8 describes the external view of a basic XOR-compression macro configured for a design with *N* scan input/output pins and *M*N* internal scan channels. Where possible, the pins names and connectivity are very similar to that of OPMISR and OPMISR+. The test input data is brought in via the *N* scan input pins and fed the N-wide RSI_SI pins. The *M*N* wide decompressed data is fed to the *M*N* wide internal scan channel heads via the SWBOX_SI pins. The test response is gathered from the *M*N* scan channel tails by the SWBOX_SO pins. The compressed test response is fed back to the tester by the *N*-wide DSO SO pins. The control signals SCOMP and SPREAD control the operation of the compression and spreading operations.

Table A-8 XOR Compression Macro Pin Connectivity

Pin Name	Direction	Connection	Shareable	Purpose
RSI_SI[0:N-1]	Input	Chip-level Scan Input receiver	With any functional top-level port	Source of test data from tester
DSO_SO[0:N-1]	Output	Chip-level Scan Output driver	With any functional top-level port	Passes test response to tester
SCOMP	Input	Chip-level Test Mode control	With any functional top-level port. Functional compliance value=0	Active when output test response is being compressed
SPREAD	Input	Chip-level Test Mode control	With any functional top-level port. Functional compliance value=0	Active when input test date is being decompressed. Note that this pin is optional and will not exist when the user chooses to not use the XOR-spreader when creating the compression macro.

Pin Name	Direction	Connection	Shareable	Purpose
SWBOX_SI[0:M*N-1]	Output	Head of internal scan channels	Not applicable	Feeds decompressed test stimuli to internal scan channels
SWBOX_SO[0:M*N-1]	Input	Tail of internal scan channels	Not applicable	Feeds decompressed test stimuli to internal scan channels

Table A-9 describes the additional pins required for X-tolerance using the Channel masking method described in "OPMISR Test Modes" on page 111. These pins and their purpose are identical to that of the OPMISR and OPMISR+ masking logic.

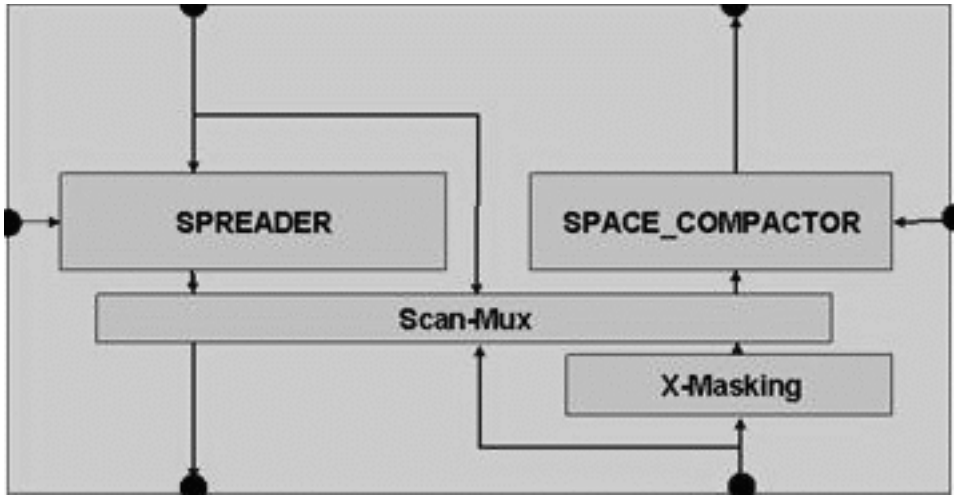
Table A-9 XOR Compression Macro Pin Connectivity for X-Masking

Pin Name	Direction	Connection	Shareable	Purpose
CK	Input	Chip-level Test clock	With any functional top-level port. Functional compliance value=0	Clock to enable loading of mask data during channel mask load state
CME	Input	Chip-level Test data input	With any functional top-level port	WIDE1 Mask enable data from tester. Exists only when WIDE1 masking is selected. Operational during scan load/unload. One bit per scan cycle.
CME0, CME1	Input	Chip-level Test Mode control	With any functional top-level port.	WIDE2 Mask enable data from tester. Exists only when WIDE2 masking is selected. Similar purpose as above.

Pin Name	Direction	Connection	Shareable	Purpose
CMLE	Input	Chip-level Test Mode control	With any functional top-level port. Functional compliance value=0	Enables loading of mask data during channel mask load state.

Figure A-47 on page 177 shows an internal conceptual view of the XOR-Compression Macro. There are 4 conceptual blocks composed of the XOR-Spreader, XOR-Compactor, Scan Multiplexing Logic and the optional X-Masking logic.

Figure A-47 Internal View of XOR-Compression Macro



Modes of Operation

Figure A-48 on page 178 shows an external view of the XOR Compression Macro. The XOR compression macro has the following modes of test operation.

1. Compression Mode with both XOR-spreader and XOR-compactor active (See Figure A-49 on page 179). This is set up when both SPREAD and SCOMP are active.
2. Compression Mode with scan fanout spreader and XOR-compactor (See Figure A-50 on page 179). This is set up when SPREAD is active and SCOMP is inactive. Another possibility is that the XOR-Compression Macro is configured without the SPREAD pin in which case this is the only available Compression Mode. In this case the Scan Input pin

X will feed the scan channels connected to scan channels X, X+M, X+2M, X+3M etc., where M is the scan fanout.

3. Full-scan mode (see Figure A-51 on page 180) is established when SCOMP is inactive and SPREAD is inactive or absent. In this case multiple internal scan channels are concatenated to form full scan chains. The scan chain X for example will be affiliated with SCANIN(X), RSI_SI(X), and scan channels X, X+M, X+2M, X+3M... and finally DSO_SO(X) and SCANOUT(X). Each internal scan channel (i) is affiliated with the pins SWBOX_SI(i) and SWBOX_SO(i).

Figure A-48 XOR Compression Macro Connection to I/O Pins and Scan Channels of Design

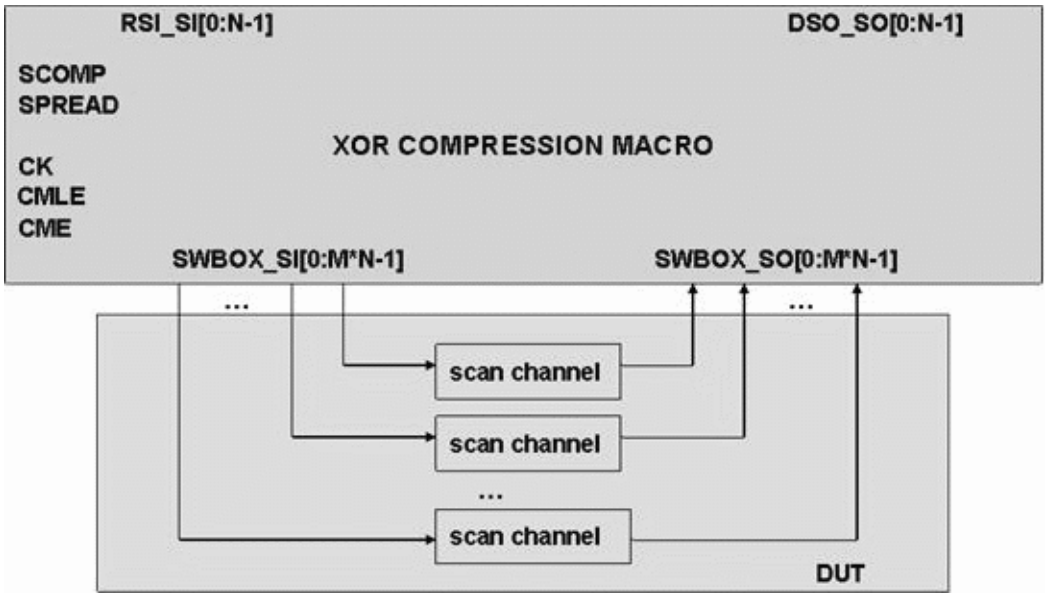


Figure A-49 Compression Mode with Both Spreader and Compactor Active

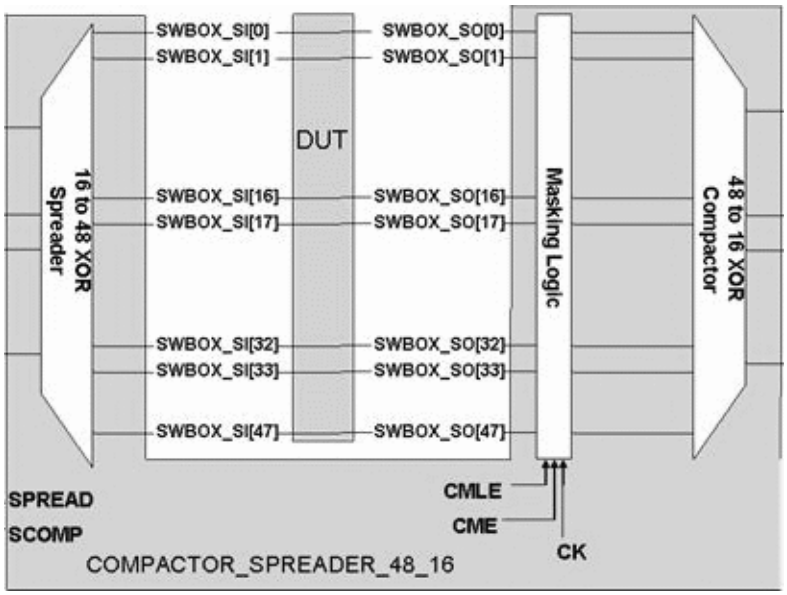


Figure A-50 Compression Mode with Scan Fanout and Compactor Active

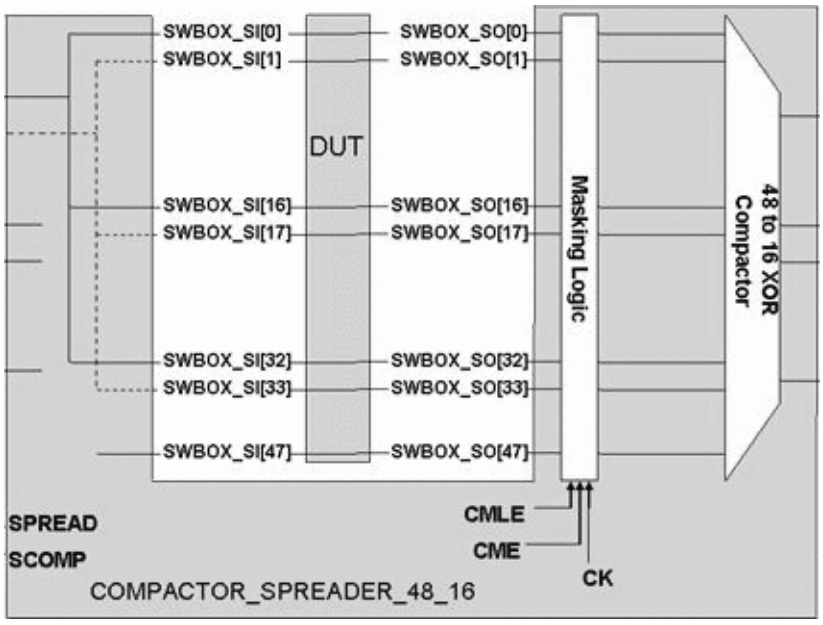
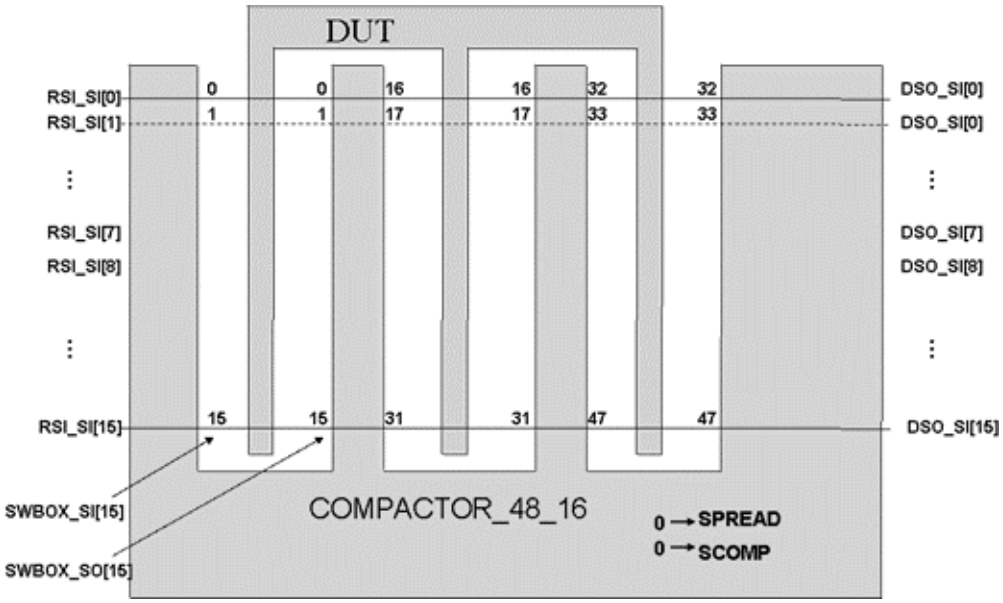


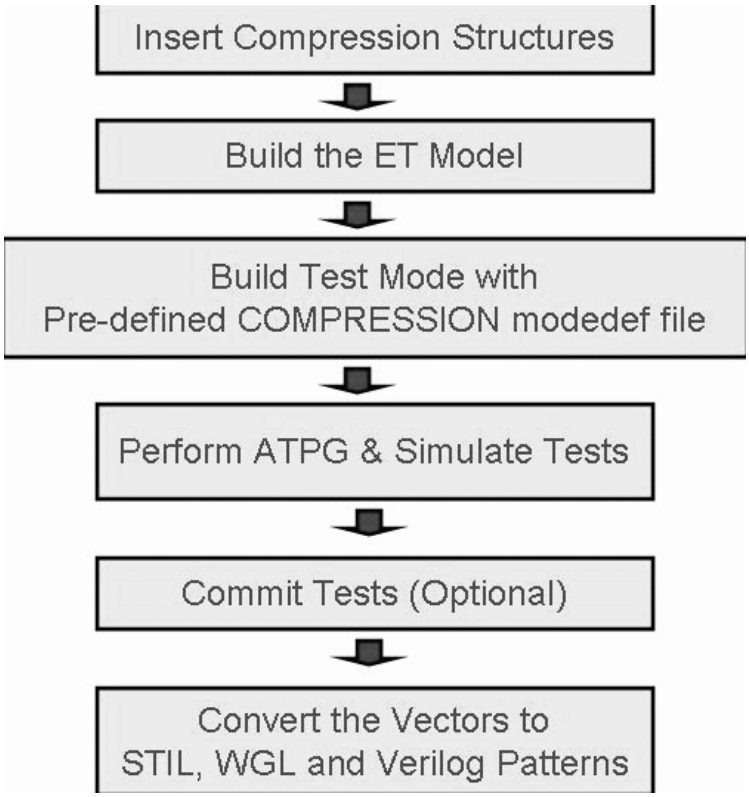
Figure A-51 XOR Compression in Full Scan Mode



XOR Compression Design Flow

Figure A-52 shows the design flow starting from compression macro insertion through ATPG.

Figure A-52 Design Flow for XOR Compression



XOR Compression Limitations

Consider the following limitations:

- The `WIDE0` masking option is not allowed for XOR compression. This restriction is enforced since the extra `WIDE0` logic gives no advantage over the no masking option. With `WIDE0` masking, every channel is masked when the mask enable is asserted. Therefore, no useful data can be obtained from cycles that assert the mask enable. Unlike MISR compression, that cycle's X bits can instead be ignored by the tester. This makes the no masking option better since it saves the additional mask enable pin and additional on chip masking logic.
- When the `numchannels` option is not an integer multiple of the `numchains` option, the scan chains in `FULLSCAN` mode may not be balanced since some of the `FULLSCAN` chains will contain one more scan channel than the others.
- When masking is used, the specified value for `numchannels` must be at least twice as large as the specified value for `numchains`.

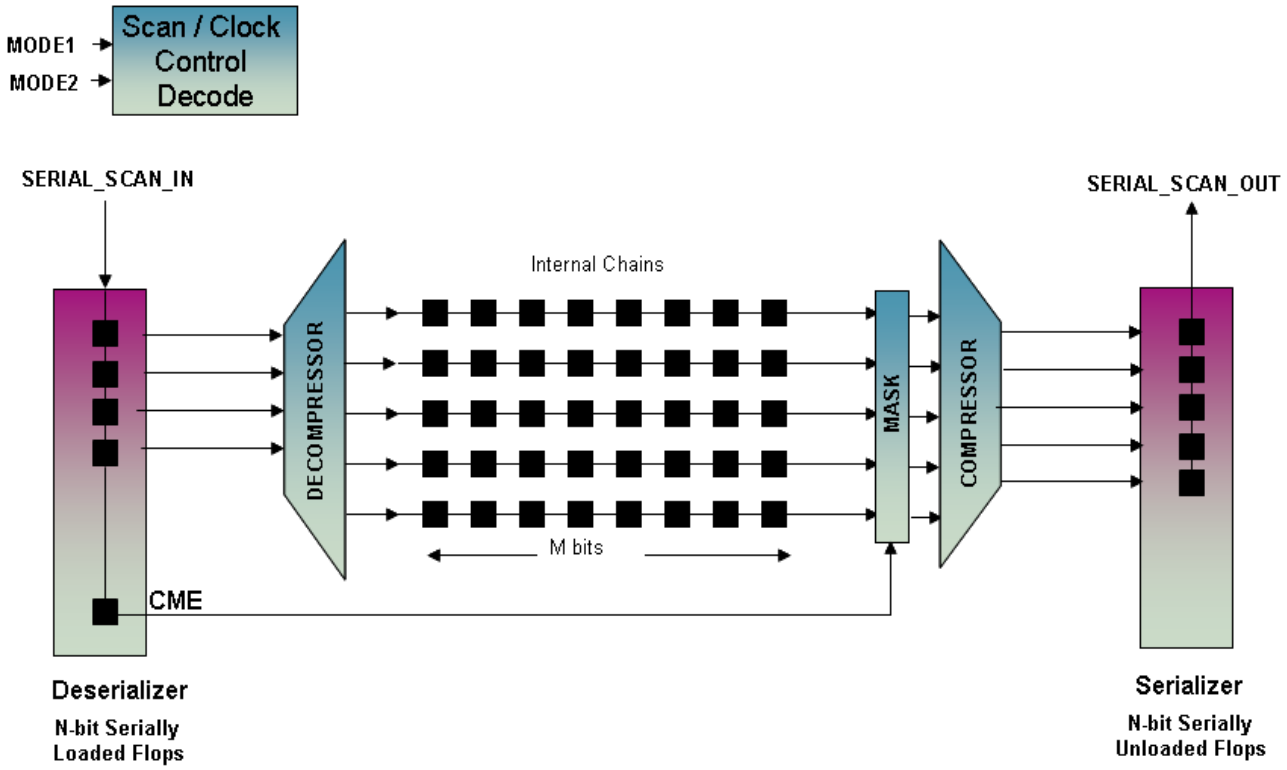
- There is no (reasonable) upper limit on `numchains` or `numchannels`, however, if the `numchannels`/`numchains` multiple is too large, the ability to diagnose the resulting data will be affected. A warning is issued when this condition occurs.

SmartScan Compression

SmartScan is a low pin compression solution that supports as few as 1 scanin and 1 scanout pin while still allowing a reasonable amount of compression and diagnostics. This is useful for designs where limited pins are available for testing purposes, For example, when performing multi-site or system-level testing, the number of contacted test pins can be very limited. An efficient solution to meeting this requirement is to reduce the number of scanin and scanout pins on the design.

The following figure gives an overview of the SmartScan compression architecture:

Figure A-53 SmartScan Compression Architecture



In the SmartScan compression architecture, each scanin feeds an N-bit serial shift register (also known as Deserializer) and each scanout is similarly fed by an N-bit serial shift register (also known as Serializer). This is typically known as the Serial interface. To load data into each channel, the Deserializer first needs to be completely loaded with the test data that would normally be applied to the Decompressor directly from multiple scanin pins. After the Deserializer is loaded, the clock to the internal scan chains starts and the test data is shifted into a channel. On the output side, all the bits within the Serializer simultaneously capture

data from the last flops of the channels and then serially shift it out through a single scanout pin. The Deserializer and Serializer operations are overlapped such that while new data is shifted in through the SERIAL_SCAN_IN pin, the response data loaded into the Serializer (from the scan chains) is simultaneously shifted out through the SERIAL_SCAN_OUT pin.

RTL Compiler supports generation and insertion of the SmartScan compression macro. This includes the Deserializer and Serializer registers, the clock control logic, and the optional mask registers. RC also generates the necessary interface files required in the Encounter Test design flow. Refer to Figure A-54 on page 185 and Figure A-56 on page 193 for more information on these files.

Refer to the *Inserting Scan Chain Compression Logic* chapter in *Design for Test in RTL Compiler Guide* for more information on SmartScan compression architecture and insertion of SmartScan compression macro.

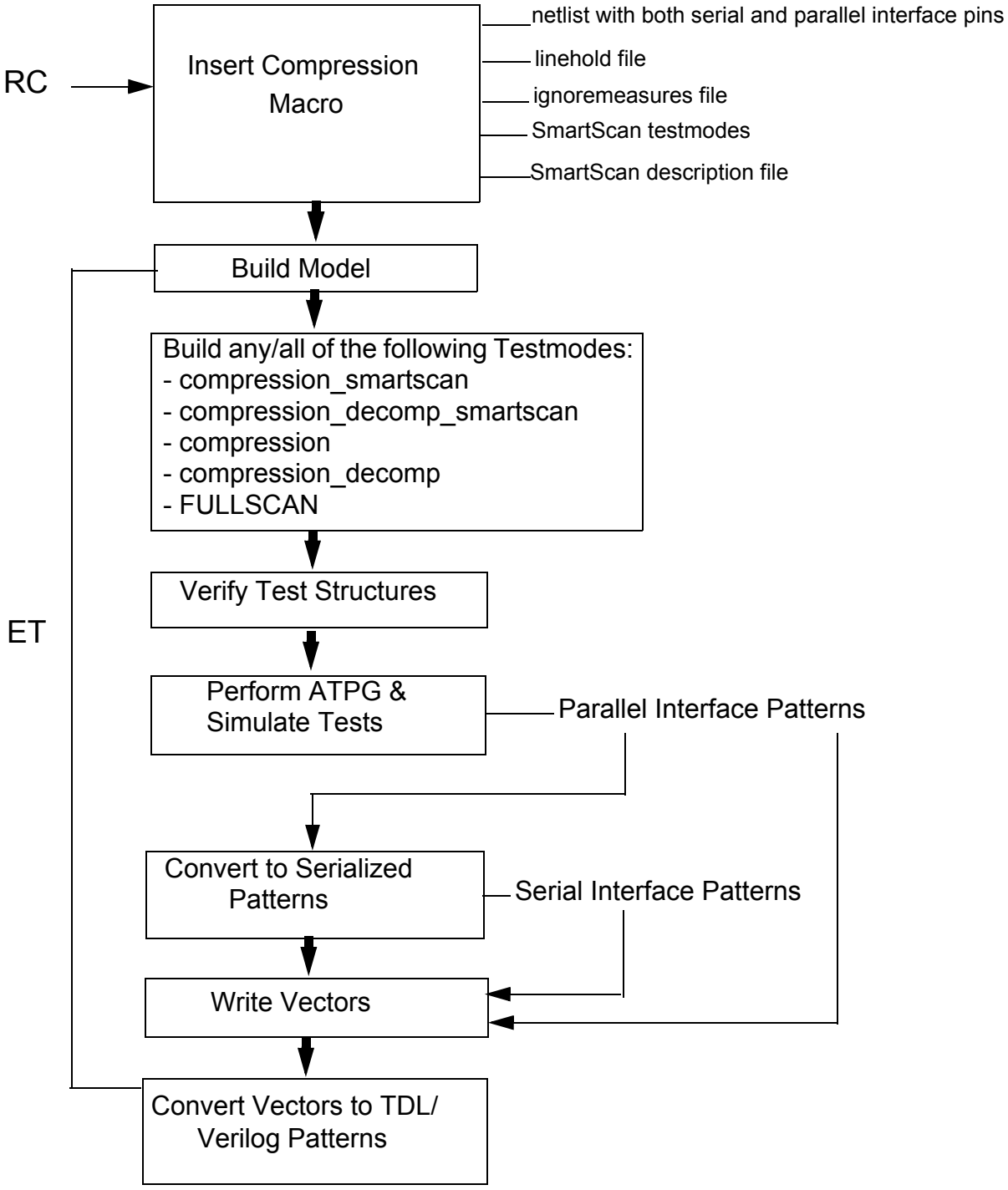
Encounter Test supports the SmartScan compression inserted by RC for both serial and parallel interfaces.

- Parallel Interface is where several scanin and scanout pins are available and these are directly connected to the XOR compression network.
- Serial only Interface has only a few serial scanin and scanout pins that connect to the Deserializer and Serializer registers, which in turn are connected to the XOR compression network.

Compression Serial and Parallel Interfaces

The following figure depicts the design flow for this scenario:

Figure A-54 Design Flow for SmartScan Compression with Parallel and Serial Interface



Note: As of now, `write_vectors` command converts SmartScan vectors only to TDL/ Verilog format.

In this scenario, the Verilog netlist generated by RC contains several scanin and scanout pins (parallel interface) for parallel access to the XOR compression network. One or more of these pins is also shared with the serial interface.

This scenario allows the patterns to be applied either via the Parallel or the Serial interface. For example, the parallel interface can be used during manufacturing test, while the Serial interface can be used to apply patterns during system test.

SmartScan Testmodes

With SmartScan compression, RC generates two SmartScan test modes, `compression_smartscan` and `compression_decomp_smartscan` for test generation. If there is no XOR spreader on the input side, then only the `compression_smartscan` testmode is generated.

Two control signals are required for SmartScan operation. These can be PI controlled or can be internally generated test signals:

- `SMARTSCAN_ENABLE`
  - Will be at Active High value in SmartScan Testmodes
  - Inactive value will cause SmartScan flops to be included within the scan chains
- `SMARTSCAN_PARALLEL_ACCESS`
  - Active High value will select Parallel interface; inactive value selects Serial interface

Performing ATPG

Pattern generation will be done using the Parallel interface, and will be post-processed to also be applied through the Serial interface.

ATPG Constraints in SmartScan Testmodes

The following constraints must be applied during ATPG in SmartScan testmodes. Faults untestable due to these constraints will be targeted in non-SmartScan testmodes.

- ATPG will not stim Scanins or measure Scanouts during capture cycles
  - Increases complexity of serialized patterns

- No assumption is made as to whether all the scan pins will be contacted when applying the serialized patterns at the tester or on the board.

- Scan Enable must be inactive during capture cycles within Logic Tests
  - When Scan Enable is Active, SmartScan Controller allows clock to scan chains only every Nth pulse of the top level clock (CLK)
  - Will allow Scan Enable being active for first (launch) Pulse in LOS tests

Linehold File

ATPG uses the linehold file generated by RC for SmartScan testmodes. This file must list the parallel Scan in pins (real or pseudo), the channel mask enable pin (if present, to its inactive value) and the scan enable pin (to its inactive value) for `create_logic_tests`.

A sample linehold file is given below:

```
Hold Pin PSI1 = X;
Hold Pin PSI2 = X;
Hold Pin SE = 0;
Hold Pin CMLE = 0;
```

Ignoremeasures File

ATPG uses the ignoremeasure file generated by RC for SmartScan testmodes. This file lists the parallel Scan out pins (real or pseudo) and is used to prevent test generation for data from SO pins during capture.

A sample ignoremeasures file is given below:

```
PSO1
PSO2
PSO3
```

Converting Parallel Interface Patterns to Serialized Patterns

After pattern generation is complete, the generated parallel patterns are converted into serial patterns so that they can be applied through the SmartScan interface, that is, using SERIAL_SCAN_IN and SERIAL_SCAN_OUT pins. This gives the flexibility to apply the patterns either through parallel or the serial interface. For example, you might want to apply patterns using parallel interface during manufacturing test and using serial interface during system test.

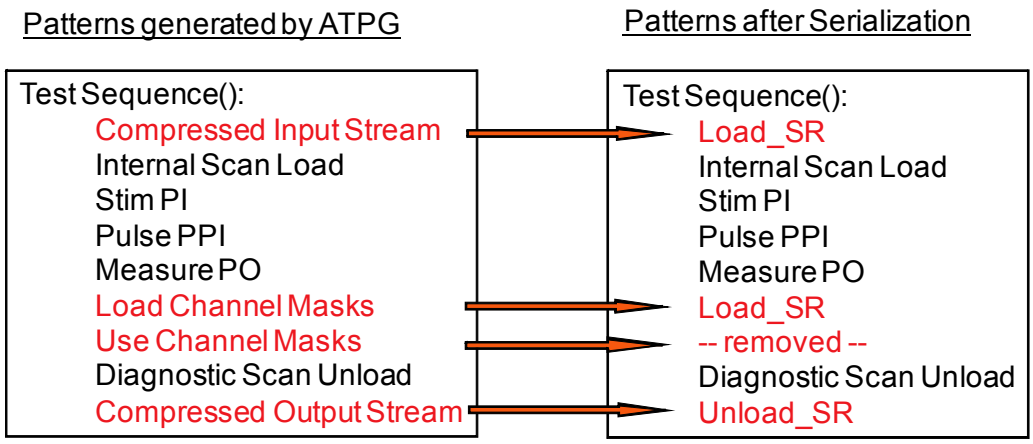
The `convert_vectors_to_smartscan` command requires the following input:

- SmartScan description file - this defines serial SI-SO pins and correlate bits of Serializer/Deserializer registers to the Parallel pins
- SmartScan sequence file - this is used to set the `SMARTSCAN_PARALLEL_ACCESS` signal to zero
- Test sequence name - this is the name of the test sequence defined in the SmartScan initialization sequence file.

Refer to convert_vectors_to_smartscan in the *Encounter Test: Reference: Commands* for the syntax of the command.

The following figure shows the changes to the test pattern made by `convert_vectors_to_smartscan`:

Figure A-55 Update to ATPG Pattern by convert_vectors_to_smartscan



Compressed Input/Output Stream is replaced with Load_SR / Unload_SR, which contain the serialized data. Loading of the mask registers also is done using the Deserializer.

Note: The `Use_Channel_Masks` event is removed and the data within it is combined with the `Load_SR` of the next Test Sequence. Hence, the converted patterns cannot be reordered, as the CME data for a Test Sequence is present in the sequence following it.

The pattern generation is done only once, using the many parallel scanin and scanout pins (called parallel interface). Once these patterns are converted to use the few serial scanin and scanout pins (called serial interface), the user has the flexibility of using either set of patterns.

Note: Separate experiment names need to be used when generating scan chain and logic tests in a Smartscan testmode. User cannot use the same experiment name and append the two ATPG runs. This is because the `convert_vectors_to_smartscan` command cannot handle an input pattern file that contains both types of patterns.

SmartScan Initialization Sequence File

For SmartScan testmodes, RC generates a sequence file that sets the `SMARTSCAN_PARALLEL_ACCESS` control signal to its inactive value, which implies that the Deserializer should drive the compression channels. The `Scan_Enable` signal should also be set to its inactive value to ensure the SmartScan registers are reset.

The `convert_vectors_to_smartscan` command prepends the initialization sequence in the sequence file to the converted test patterns so that the Deserializer and Serializer logic is used for application of the converted ATPG patterns. Otherwise, the top-level Parallel Scanin/Scanouts (real or pseudo) would continue to bypass the Deserializer/Serializer registers and feed the Channels.

Note that this sequence will replace the mode initialization sequence of the SmartScan testmode. Therefore, this sequence must perform the same operations as the testmode modeinit sequence, with the exception of setting the `SMARTSCAN_PARALLEL_ACCESS`, `Scan Enable`, and `Channel Mask Enable` signals to their inactive values.

SmartScan Description File

A SmartScan Description file contains information on the SmartScan structures present in the netlist, that is, mapping of serializer and deserializer flops to the corresponding primary inputs/outputs. Each bit (flop) in the Deserializer will map to a primary input pin (or pseudo primary input pin added by edit-model) with test function SI, CME or CMI. Similarly, each bit of the Serializer will map to a primary output pin or pseudo primary output pin. The file also provides the mapping between the deserializer/serializer bits and the scan-in/scan-out used to serially shift data into the deserializer/from the serializer.

The `write_et` command in RC generates the SmartScan description file in the ASCII format. When using `write_compression_macro` to generate the SmartScan macro, this file must be created manually.

An Update register can also be optionally present between the deserializer and the decompressor. The update register is also a shift register and is of the same length as the deserializer register. The SmartScan description file will provide the mapping of flops/bits in update register with the corresponding primary inputs pins.

The `convert_vectors_to_smartscan` command uses the SmartScan description file through the `smartscanfile` keyword.

SmartScan Description Syntax

The following syntax statements will be supported initially for comments:

Line comments

```
//

--

#
```

Block comments

The block comments can span multiple lines. These are as shown below:

```
/*block of comments*/ - "/*" to start and "*/" to end block comments.
```

Comments spanning multiple lines per the following example:

```
/* comment line1
   comment line 2
   Comment line 3 */
```

Header

The SmartScan description file can optionally also have an header present, which can contain some comments for the user to understand what this file contains. The complete header will be treated as comment by the parser and it must be enclosed using the line comments syntax shown above.

Smart Scan Macro Version

The SmartScan description file should have the smart scan macro version specified that is used to generate this file. The syntax for this is as below:

`SMARTSCAN_MACRO_VERSION=<`*`version_number`*`>;`

Example:

```
SMARTSCAN_MACRO_VERSION=1.0;
```

The smart scan macro version can start with 1.0 and increment as and when we make changes to the smart scan hardware (serializer, deserializer, clock-controller etc) inserted by the RC. This will help ET understand which version of hardware it is dealing with, if there are any incremental updates to hardware.

Statement Syntax

A statement would define a serializer, deserializer or an update register and the mapping of its flop with the primary input (or pseudo primary input) pins. It will have the following syntax:

```
SMARTSCAN_REG = <Reg_type> {
     [Serial_Primary_PIN = <serial_pin_name>;]
    REG_BIT_CORRESPONDENCE = (
         <Hierarchial_flop_name>, BIT_INDEX =<bit_index>, PIN= <primary_pin_name>;
         <Hierarchial_flop_name>, BIT_INDEX =<bit_index>, PIN= <primary_pin_name>;
            .
            .
      )
  };
```

Here:

`"Reg_type` : Specifies the type of register and can be of following types

- DESERIALIZER_SHIFT_REG
- SERIALIZER_SHIFT_REG
- DESERIALIZER_UPDATE_REG

`Serial_Primary_PIN` : Specifies the primary (or pseudo) input or output pin. This will not be specified for `Reg_type= DESERIALIZER_UPDATE_REG`. It can be one of following:

- SERIAL_SCAN_IN
- SERIAL_SCAN_OUT

`serial_pin_name` : Name of the top level Serial Scan IN/OUT pin.

`Hierarchial_flop_name` : Hierarchical name for the flop in the shift register.

The flopname can also be enclosed in double quotes (""), to support names having special characters or escaped names. The use of double quotes is not mandatory for simple names which do not have any special characters.

`Bit_Index` : This specifies the position of flop in the shift register.

For deserializer and update register: In serial mode, the flop closest to SSI pin has bit_index 1 and next closest has bit_index 2 and so on.

For serializer: In serial mode, the flop closest to SSO pin has bit_index 1 and the next closest flop has bit_index 2 and so on.

`Primary_pin_name :` Equivalent parallel scan in or scanout pin corresponding to the specified flop. This is the pin which feeds in the data directly into the corresponding mux.

Statements

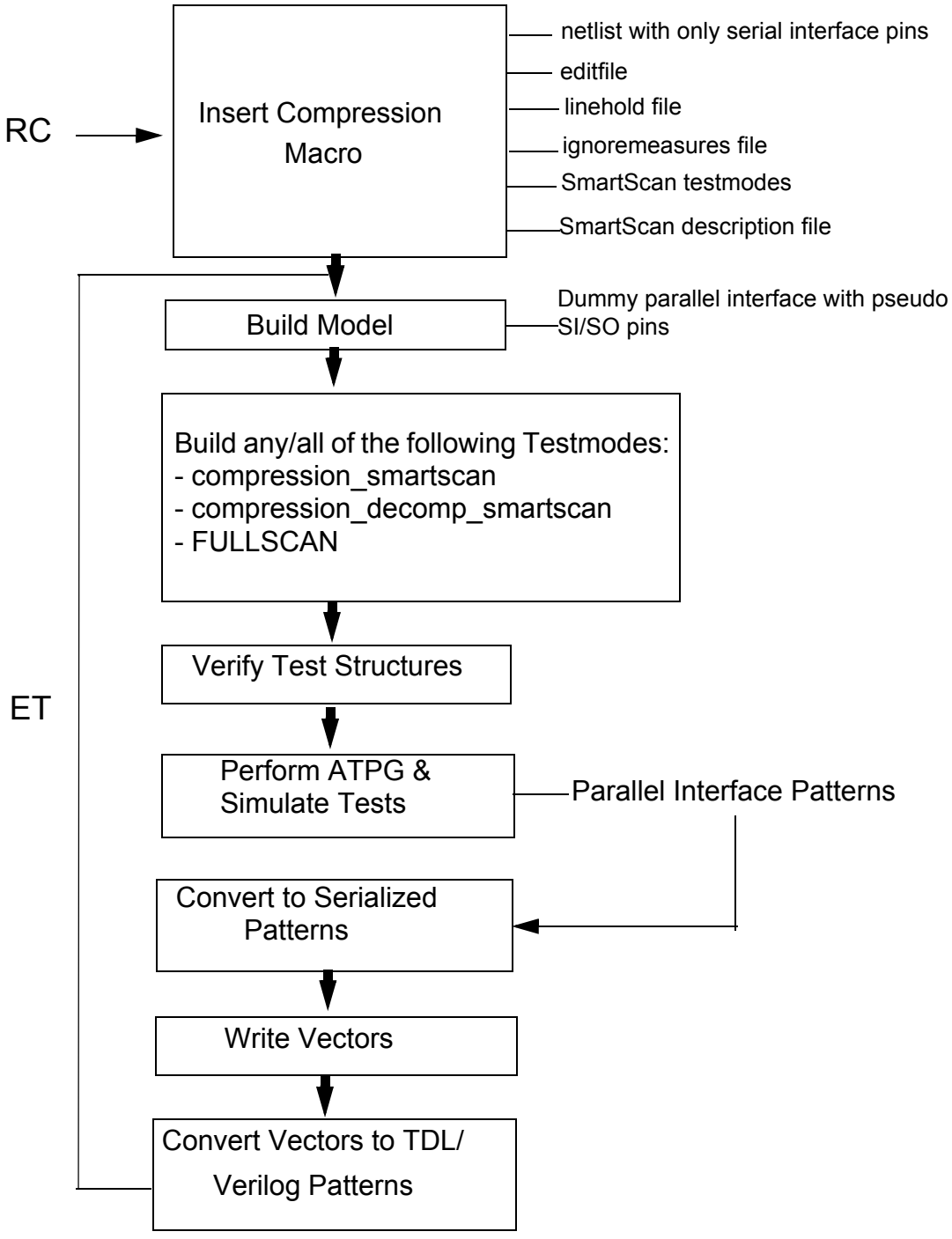
The language is case insensitive. Therefore, although the above shows the statement elements in uppercase, they really can be entered in any case. Note that the names of cells, instance, pins, etc must be in the same case as they are in the model.

- The statement must end with a semicolon (;).
- Use of braces '{' and brackets '(' is compulsory as shown above
- Use of '=' is mandatory, where needed, as shown in syntax above
- Use of comma ',' is mandatory, where needed, as shown in syntax above
- The comments may appear any place white space may appear. The white space is either a blank or a new line character

Compression with Serial Only Interface

The following figure depicts the design flow for this scenario:

Figure A-56 Design Flow for SmartScan Compression with Serial Only Interface



In this scenario, the RC-generated Verilog netlist contains only few scan pins for the Serial interface.

While building the model for Encounter Test, the editfile generated by RC is used to add pesudo SI/SO pins to the model to create a dummy Parallel interface to facilitate test generation. Build Testmode and test generation assume the presence of N scanin and N scanout pins (Parallel interface), but only a few of those pins actually exist in the hardware. The pseudo pins are added by specifying the `editfile` keyword for `build_model`.

Pattern generation is done using the Parallel interface and the `convert_vectors_to_smartscan` command converts the patterns to be applied through the Serial interface. The pseudo pins are deleted from the patterns to be used for simulation and at the tester.

In this case, only serial interface can be used when applying the patterns either at the tester or on the board.

B

Mode Definition File Syntax

Mode Definition File

Each test mode that you create is described by a mode definition file, used as input to the *Build Test Mode* function on the *File* pull-down menu. This is a text file that can be created from scratch by using your own text editor or by editing a copy of one of the test mode definition files included with Encounter Test. These test mode definition files are in the directory named *$Install_Dir*/defaults/rules/modedef.

Mode Definition Syntax

The syntax for the Mode Definition Statements is shown in “railroad-style” syntax diagram format.

You may have as many comments as you wish in your file. The supported comment characters are:

line comments:

- //
- --
- #

block comments:

- */*block of comments*/* - “/*” to start and “*/” to end block comments.

Comments spanning multiple lines per the following example:

```
/* comment line1
   comment line 2
   comment line 3 */
```

A comment may precede, follow, or be imbedded within a statement, or it can be on a separate line. There cannot be more than one comment on the same line. For example, the following line is not supported:

```
/*comment*/ data /*comment*/
```

Mode Definition Statements

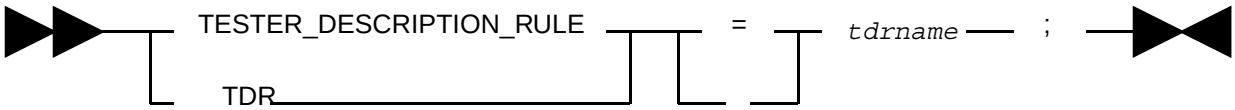
The following mode definition statement syntax diagrams and keyword descriptions are listed in this section:

- "ASSIGN" on page 210
- "CLOCKDOMAIN" on page 238
- "COMETS" on page 233
- "CORRELATE" on page 239
- "CUTPOINTS" on page 236
- "DELETE" on page 236
- "DIAGNOSTIC_MODE" on page 241
- "FAULTS" on page 208
- "FIXED_VALUE_DEFAULT" on page 210
- "IMPLICIT CHOPPERS" on page 241
- "MACRO_MODE" on page 239
- "MISR_PROPERTIES" on page 242
- "ON_BOARD_MISR" on page 232
- "ON_BOARD_PRPG" on page 231
- "OPCG" on page 242
- "RAM_INITIALIZE_VALUE" on page 239
- "SCAN" on page 197
- "SEQUENCE_DEFINITION" on page 230
- "SIGNATURE_OBSERVATION_MODE" on page 237
- "TEST_FUNCTION_PIN_ATTRIBUTES" on page 209

- “TEST_TYPES” on page 202
- “TESTER_DESCRIPTION_RULE” on page 197

TESTER_DESCRIPTION_RULE

This required statement specifies the name of the tester description rule (TDR) to be used in processing this test mode. Refer to “Introduction to Tester Description Rules (TDRs)” on page 259.



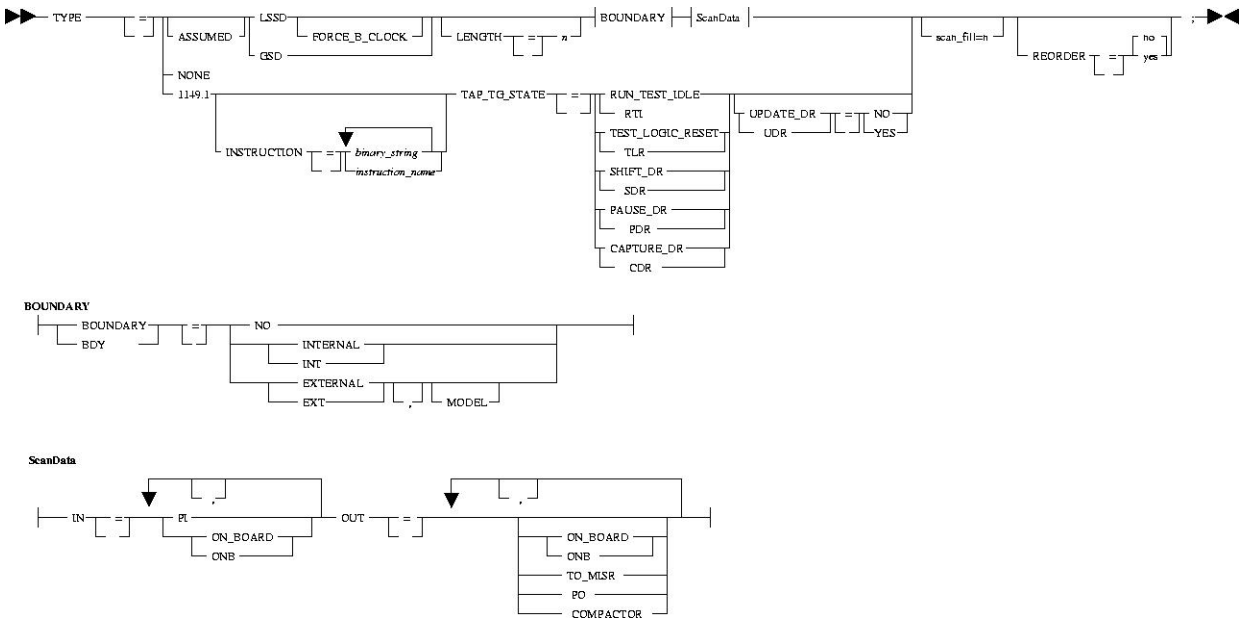
In the syntax:

tdrname - The file name of the TDR. The TDR specifies the characteristics of the tester to which this test mode's data is targeted.

Additional related information is available in “Resolving Internal and External Logic Values due to Termination” in the *Automatic Test Pattern Test Generation User Guide*.

SCAN

This required statement specifies whether the design is to be scannable in this test mode, and what form of scan design is used.



This statement uses the following keywords:

- `TYPE` - This keyword indicates that the following keyword will specify the basic type of scan operation to be used in this test mode. It can have the following values:
  - `NONE`

    Specifies that the design does not use scan design in this mode.
  - `ASSUMED`

    Specifies that an assumed scan chain test mode is to be used.

    Note: The following are disallowed if specifying assumed scan:

    `IN=ON_BOARD/ONB` and `OUT=ON_BOARD/ONB/TO_MISR` on the `SCAN` statement

    `ON_BOARD_PRPG`, `ON_BOARD_MISR`, and `SIGNATURE_OBSERVATION_MODE` mode definition statements
  - `LSSD`

Specifies that in this test mode the design should conform to the Encounter Test Level Sensitive Scan Design guidelines.

- `FORCE_B_CLOCK`

  Specifies that B_SHIFT_CLOCKs are to be pulsed after any non-B_SHIFT_CLOCK is pulsed.

- `LENGTH`

  Specifies the length of the scan sequence. This defaults to the length of the longest scan chain in the test mode, and *n* must be larger than this or it will be changed to the length of the longest scan chain.

- `GSD`

  Specifies that in this test mode the design uses “general scan design” and should follow the Encounter Test General Scan Design guidelines.

- `1149.1`

  Specifies that in this test mode the design uses the IEEE standard 1149.1, which incorporates a TAP controller and boundary scan.

  - `INSTRUCTION`

    Specify instructions to be loaded into the IR to select test data registers (TDRs) for scanning through the TAP. For stored pattern test generation this instruction configures the design so that the test generator can work effectively in the TAP controller state designated by `TAP_TG_STATE`, and analyze and determine which latches are scanned by the set of instructions selected by the specified instructions.

    The instructions to be loaded are specified using one of the following methods:

  - `binary_string` - specify the binary bit string to be loaded into the IR, with the bit closest to TDI being the left-most bit and the bit closes to TDO being the right-most bit.

    `Example: INSTRUCTION=10` (The IR is two bits long.)

  - `instruction_name` - specify the name of the instruction whose bit encoding is defined in the BSDL used as input to Build Test Mode.

  - `TAP_TG_STATE`

    This is an optional parameter used to specify the TAP controller state in which test generation is to be performed. Do not specify this parameter if the test

mode is not for the purpose of test generation or if test generation is to be performed with LSSD or GSD scan. Acceptable values are:

- `RUN_TEST_IDLE` (RTI) - test generation is to take place with the TAP controller in the Run-Test/Idle state.
- `TEST_LOGIC_RESET` (TLR) - test generation is to take place with the TAP controller in the Test-Logic-Reset state.
- `SHIFT_DR` (SDR) - test generation is to take place with the TAP controller in the Shift-DR state.
- `PAUSE_DR` (PDR) - test generation is to take place with the TAP controller in the Pause-DR state.
- `CAPTURE_DR` (CDR) - this option is intended for use only if you have implemented parallel capture clocking via the `CAPTURE_DR` state. This is not a recommended way to implement internal scan via an 1149.1 interface, but Encounter Test will support it in a limited fashion.
- `UPDATE_DR` (UDR) - this option is intended for use only if you have implemented internal scan load via the `UPDATE_DR` state. This is not a recommended way to implement internal scan via an 1149.1 interface, but Encounter Test will support it in a limited fashion. See "1149.1 Mode Initialization Example with User-Supplied Custom Scan Sequence" in the *Test Pattern Data Reference* for additional information.

- `scan_fill=`*n*

  This keyword is used to load non-scan latches and flip-flops are expected to be loaded from scan bits during scan.

  Specify the number of scan cycles required to completely load non-scan latches or flip-flops. The default is zero (0). If a value greater than zero is specified, a `scan_fill` sequence type is automatically generated and applied after every Scan_Load even with ATPG generated tests. Refer to the definition for sequence type `scan_fill` in the "Define_Sequence" section of the *Test Pattern Data Reference*.

- `REORDER`

  Specify whether converted tests can be reordered without need for resimulation. The default is `no` if `OUT=COMPACTOR` is specified; otherwise the default is `yes`.

- `ScanData`
  - `IN`

Indicates that the allowed sources of scan data are to follow.

- `PI`

  Specifies that the scan chains may be sourced from primary inputs.

- `ON_BOARD`

  Specifies that the scan chains may be sourced from pseudo-random pattern generators resident on the device under test.

- `OUT`

  Indicates that the allowed destinations of scan output data are to follow.

  - `PO`

    Specifies that the scan chains may feed primary outputs of the design.

  - `ON_BOARD`

    Specifies that the scan chains may feed signature registers resident on the device under test.

  - `TO_MISR`

    Indicates an On-Product MISR test mode is being used. Refer to On-Product MISR (OPMISR) Test Mode for related information.

  - `COMPACTOR`

    Indicates that scan chain outputs feed through a linear compactor before their scan out data appears at the scan output pins.

- `BOUNDARY`

  This keyword indicates that the boundary scan attributes of the test mode are to follow.

  - `NO`

    Specifies that no boundary scan processing is to be done in this test mode. Any tests generated in this mode will not assume boundary scan.

  - `INTERNAL`

    This keyword indicates that the design uses boundary scan, and only internal test data is to be generated in this test mode. For mixed analog/digital designs, specify this keyword in conjunction with `assign pin=<accessible non-test control digital pin>` and `test_function=bdy` (refer to ASSIGN on page 210 and "RPCT Boundary Controls" on page 48 for more information) to identify the digital pins that are accessible beyond test control signals.

- `EXTERNAL`

  This keyword indicates that the design uses boundary scan, and only external test data is to be generated in this test mode.

- `MODEL`

  This keyword is required if this mode is to be used in building a boundary model for this structure.

- `EMBEDDED_PIPELINES`

  This keyword provides a method to identify compression testmodes which pipelines embedded in the compression and decompression logic. The default value is `no`.

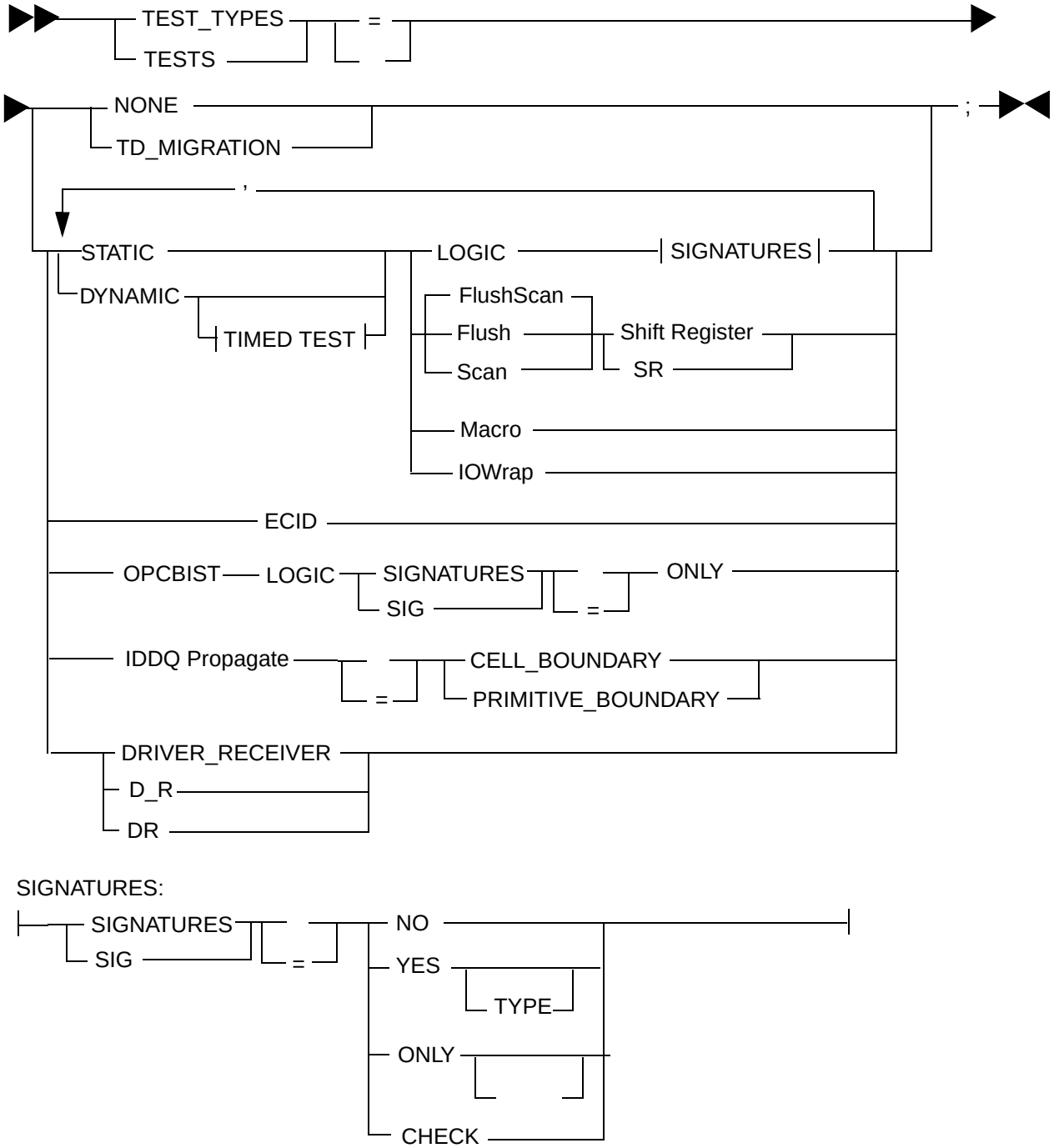
TEST_TYPES

This required statement specifies the kinds of tests that are to be generated and applied in this test mode.

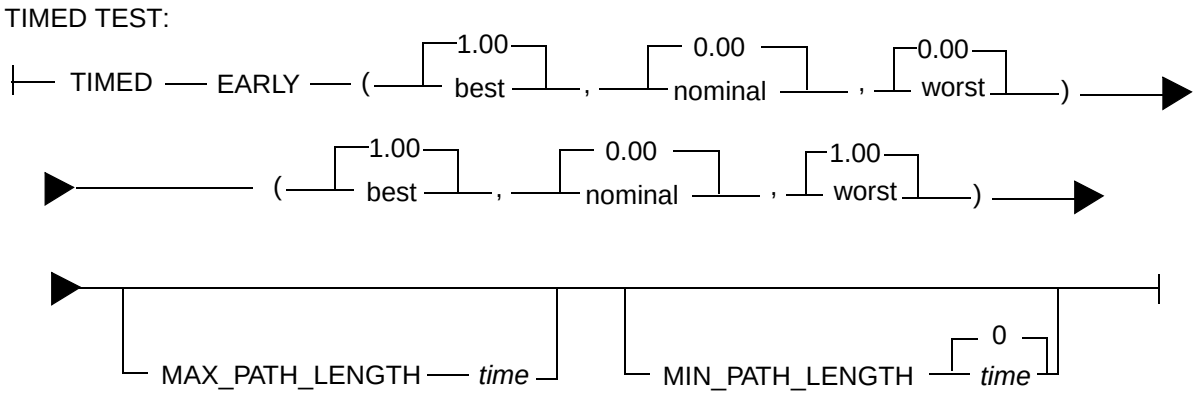
Note: The `time` keyword is specified with a positive decimal number followed by one of these parameters:

- `ns`
- `ps`
- `us`
- `ms`

❑ s



SIGNATURES:



- `NONE`

  Specifies that no test data is to be generated in this mode.

- `TD_MIGRATION`

  Specifies that this test mode exists solely for the purpose of migrating pre-existing tests from the boundary of an embedded macro to the package boundary.

- `LOGIC`

  Specifies that this test mode may be used to generate logic test data (for example, stuck faults or transition faults).

- `STATIC`

  When used with the `LOGIC` or `I/O wrap` keywords, specifies that this test mode may be used to generate tests that are largely delay-independent (for example, stuck-fault testing). When used with the `MACRO` keyword, specifies that the design contains macros that are delay-independent (design “settles” between patterns).

  - `MACRO`

    Specifies that the design may contain embedded macros for which in-place tests are to be generated. This is commonly done for embedded RAM or ROM.

  - `I/O wrap`

    Specifies that static stuck driver tests are to be created for bi-directional chip I/Os.

- `DYNAMIC`

When used with the `LOGIC` or `I/O wrap` keywords, specifies that this test mode may be used to generate tests to be applied in a carefully timed sequence (for example, delay testing). When used with the `MACRO` keyword, specifies that the test portion of the macro contains events that occur in a carefully timed sequence.

- `TIMED_TEST`

  Dynamic Timed Tests are dynamic tests that have associated timings. The `TIMED_TEST` keyword controls whether timed tests can be imported or automatically generated. This keyword also controls whether the test sequences developed by the test generators will be timed. If the timings are already present for the sequence, whether they were automatically or manually derived, new timings are not created.

  - `EARLY`

    This path specifies to check for paths that are too fast.

  - `LATE`

    This path specifies to check for paths that are too slow.

  - `best`

    Specifies the delay through a chip or net under the best or fastest conditions. Typically this delay is desired for early mode analysis.

  - `nominal`

    This delay is usually an average between the worst and best case delays and thus, will give an average delay through the logic design.

  - `worst`

    The delay across a chip or net under the worst or slowest conditions. This is typically the delay preferred for late mode analysis.

- TIMED TEST OPTIONS `MAX_PATH_LENGTH`

  Specifies the maximum path length for which path timings are to be generated. Paths on the product that are larger than the specified maximum are identified as unobservable to the test generator and fault simulator. The paths that are considered are latch to latch, PI to PO, PI to latch, and latch to PO.

- TIMED TEST OPTIONS `MIN_PATH_LENGTH`

  Specifies the minimum path length for which path timings are to be generated. Paths on the product that are smaller than the specified minimum will not have timings generated for them.

- SIGNATURES

  Indicates that the allowed usage of signature analysis for tests generated in this mode is to follow.

  - NO

    Specifies that signature analysis is not permitted in this test mode.

  - YES

    Specifies that signature analysis is permitted in this test mode.

    - TYPE - specify one of the following:

      TEST_RESET to provide independent signatures for each test. This is the default for OPMISR and OPMISRplus test modes.

      RUNNING to indicate that the signature will be accumulated across all tests with no resets between tests and that the current MISR state (signature) should be provided at the end of each test.

      Final to indicate that a final signature accumulated across all tests within a Tester_Loop should be computed and be devoid of resets between tests.

      If the test mode is not OPMISR or OPMISRplus, the default is RUNNING.

    Note: Specifying SIGNATURES=YES and ECID within the same test mode is an improper combination.

  - ONLY

    Specifies that signature analysis is the only output observation method permitted in this test mode (individual primary output or scan elements cannot be measured).

  - CHECK

    This is a “checking only” option that invokes *Logic Test Structure Verification* X-state checking in this mode (check to insure that X-sources cannot be observed). WRP test generation is not permitted in this mode if this option is specified.

- ECID

  Specifies that Electronic Chip ID (ECID) tests are to be generated that enable encoding of manufacturing information on each chip. Refer to “Electronic Chip ID (ECID) Test” in the *Encounter Test: Reference: Legacy Functions* for additional information.

- OPCBIST

This keyword is specified for a test mode in which tests will be applied under the control of an on-product finite-state machine such as a BIST controller, where the only required external stimuli after initialization is a string of clock or oscillator pulses. OPCBIST can be specified in combination with STATIC, DYNAMIC, DRIVER_RECEIVER, and IDDQ. It must be accompanied by the modifier keywords LOGIC and SIGNATURES only.

- `SHIFT_REGISTER`

  Specifies that tests are to be generated that focus on verifying the integrity of the scan operations for a GSD, LSSD, or 1149.1 design. This test type is not valid if `SCAN TYPE=NONE`.

  - `FlushScan`

    Include both LSSD Flush and Scan Chain test portions of scan chain test.

  - `Flush`

    Include only LSSD flush test portion of scan chain test.

  - `Scan`

    Include only scan chain test portion of scan chain test.

- `IDDq PROPAGATE`

  Specifies the valid “observation points” for tests generated to be used for making power supply current measurements (IDDq).

  - `CELL_BOUNDARY`

    Specifies that for IDDq tests, the fault effect must be propagated to the output of a cell to be considered detected by an IDDq test. For purposes of generating IDDq tests, the lowest level entities (above the primitive blocks) in the model hierarchy are treated as cells. Use this option if your typical cell is designed such that the detection of a defect by current measurement depends upon the state of cell inputs other than those connected to the primitive block on which the defect is modeled. This is commonly true for complex CMOS designs.

  - `PRIMITIVE_BOUNDARY`

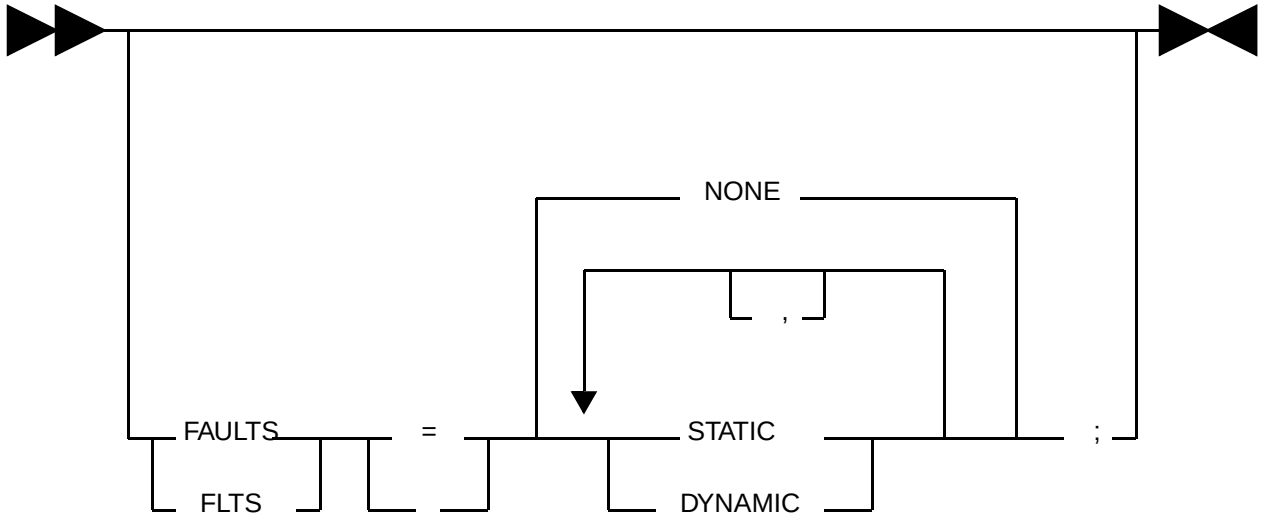
    Specifies that for IDDq tests, the fault effect must be propagated to the output of the primitive block to be considered detected by an IDDq test. Use this option if the detection of defects by current measurement depends only upon the state of the inputs of the primitive with which the defect is associated. You should specify this option also if you imported a non-hierarchical model, because in that case Encounter Test would treat the entire design as a cell.

- `DRIVER_RECEIVER`

Specifies that this test mode may be used to generate tests targeted specifically at faults on drivers and receivers connected to the primary I/O pins of the design.

FAULTS

This statement specifies the types of faults to be used for test generation in this test mode. The default is to not include any faults.

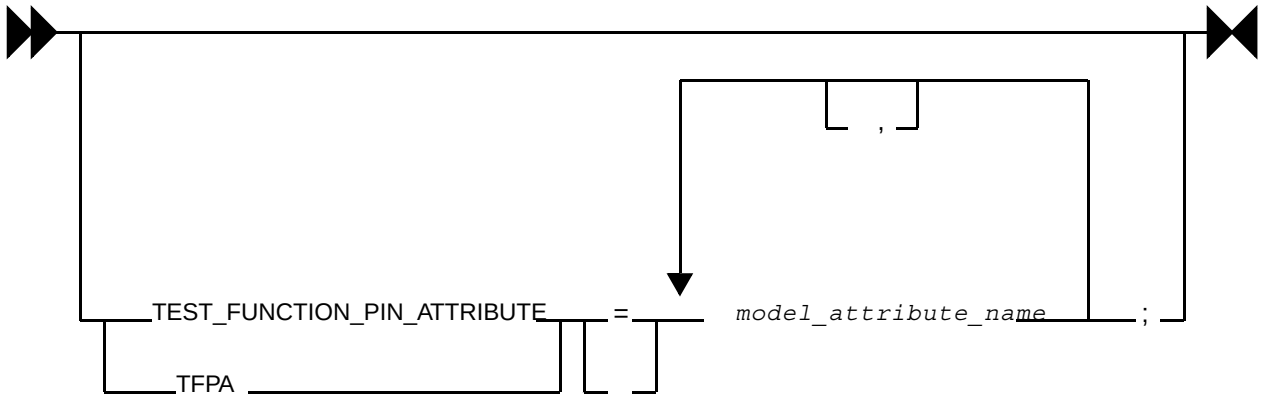


In this statement:

`NONE`	Specifies that no fault model is to be created for this test mode. This is the default.
`STATIC`	Specifies that the fault model for this test mode is to include static (for example, stuck-at) faults.
`DYNAMIC`	Specifies that the fault model for this test mode is to include dynamic (for example, transition or delay) faults. Even if dynamic faults, for example, are defined for the design by *Build Fault Model*, they cannot be processed in the test mode unless `FAULTS=DYNAMIC` is specified here.

TEST_FUNCTION_PIN_ATTRIBUTES

This optional statement specifies the names of any model attributes that carry the test functions of the pins in this test mode. The default is to not get test function information from the model data, but from the mode definition only.



In this statement:

model_attribute_name - The name of a pin attribute (keyword or property) used in the model source that carries test function information.

If conflicting test functions are found, the one specified by the earlier appearing *model_attribute_name* in the list is used. For example, if you specify

```
TFPA = TF_A, TF_B, TF_C
```

and some primary input has attributes

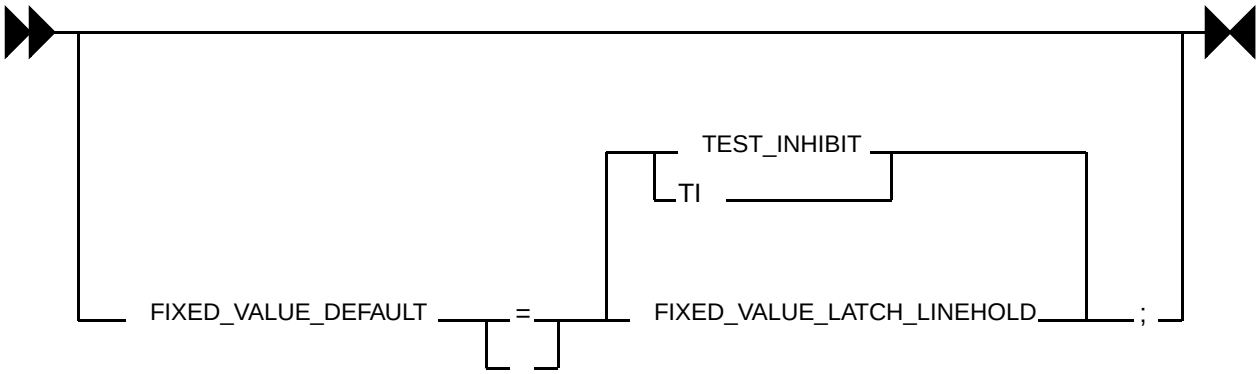
```
TF_D = +TI, TF_C = BDY, TF_B = +SE, TF_A = SI
```

then the test function attributes SI and BDY are kept for this PI in this test mode. The +TI is ignored completely because TF_D was not included on the TFPA statement. +SE and SI conflict, but TF_A appears first in the TFPA statement, so SI is used. BDY does not conflict, so it is used also, even though TF_C was the last appearing name on the TFPA statement.

Refer to "Test Function Attribute Definition" on page 32 for descriptions of test function pins.

FIXED_VALUE_DEFAULT

This optional statement specifies the test function for all fixed value latches that do not have a test function explicitly specified for them. If this statement is omitted, and such fixed value latches exist, then they will be assigned a test function of test_inhibit (TI).



ASSIGN

This optional statement specifies test function information. The specified test function is placed on a primary I/O pin, a pseudo primary input (PPI), a latch, or in rare instances, an internal node in Encounter Test's internal model. If a test function was specified on this pin or block by a model attribute, that designation is replaced by the information from this mode definition statement.

Three test functions, TI, FLH, and FSM, are recognized for latches.

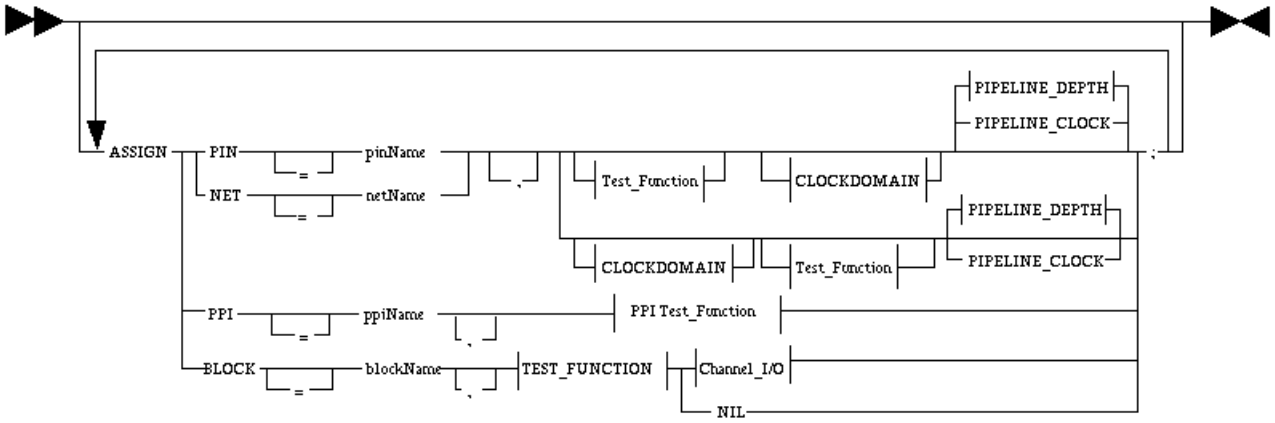
When specifying a test function for a latch, the designated pin or net must be one of the following:

- Output pin of the latch primitive;
- Net connected to the output pin of the latch primitive, but only if there are no other latches directly connected to this net;
- Output pin of a usage block which contains the latch, but only if exactly one latch within this usage block can be reached by a backtrace from this pin;
- Any net that is driven directly or indirectly by the latch, but only if the latch is contained (possibly at a lower hierarchical level) within the same entity in which the net is visible (i.e., the lowest level hierarchical entity that contains the net), and there is no other latch that can be associated in this manner with the designated net.

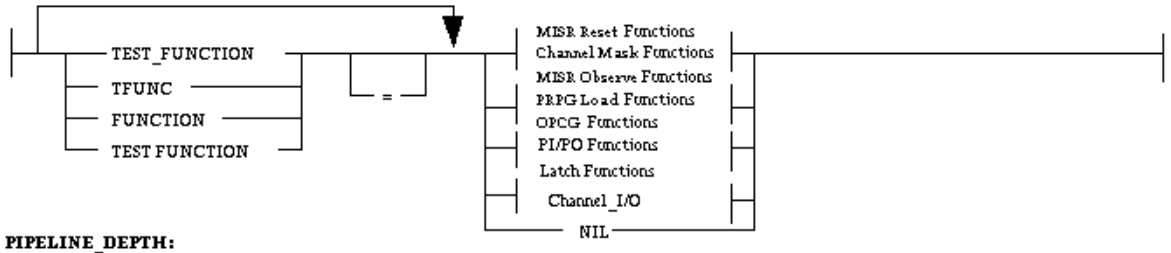
- Results of multiple ASSIGN statements are not cumulative for a given PIN. If two ASSIGN statements are used for one PIN, the last statement seen will assign the pin flag values.

Refer to "ASSIGN Statement Syntax Techniques" on page 227 for details on specifying ASSIGN parameters and to view example syntax.

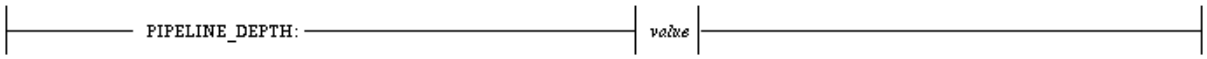
Note: Test function pins for an 1149.1 test mode are defined in the Boundary Scan Description Language (BSDL). Refer to "Test Function Pins for an 1149.1 Mode" on page 50 for more information.



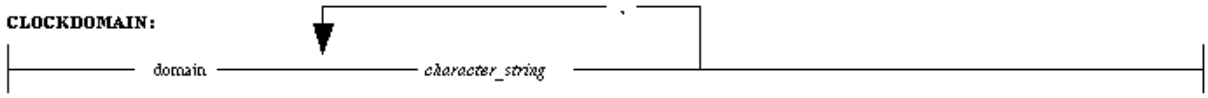
Test_Function:



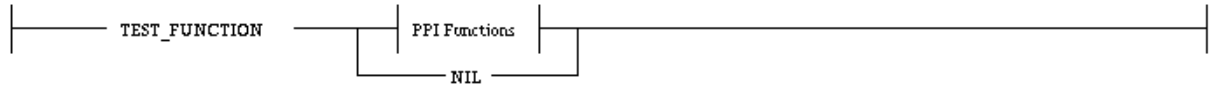
PIPELINE_DEPTH:



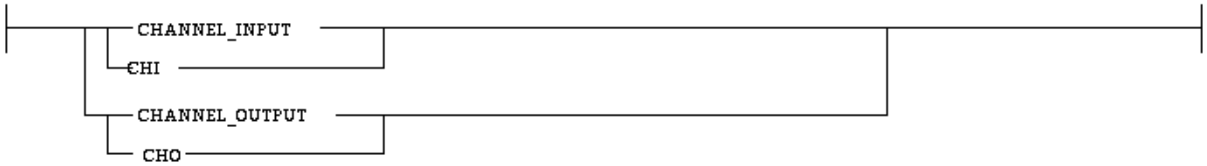
CLOCKDOMAIN:



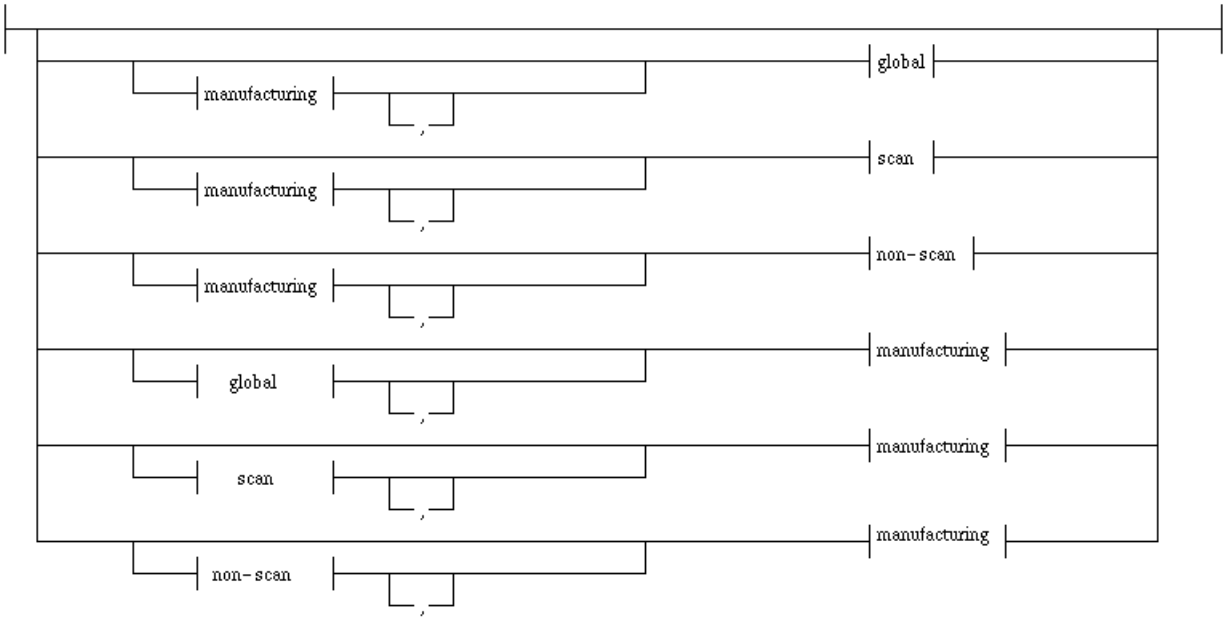
PPI Test_Function:



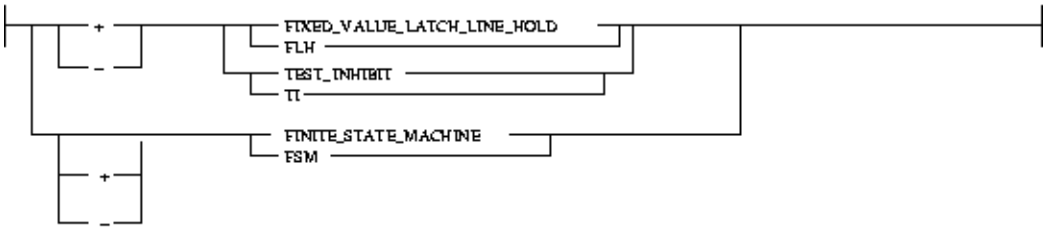
Channel_I/O:



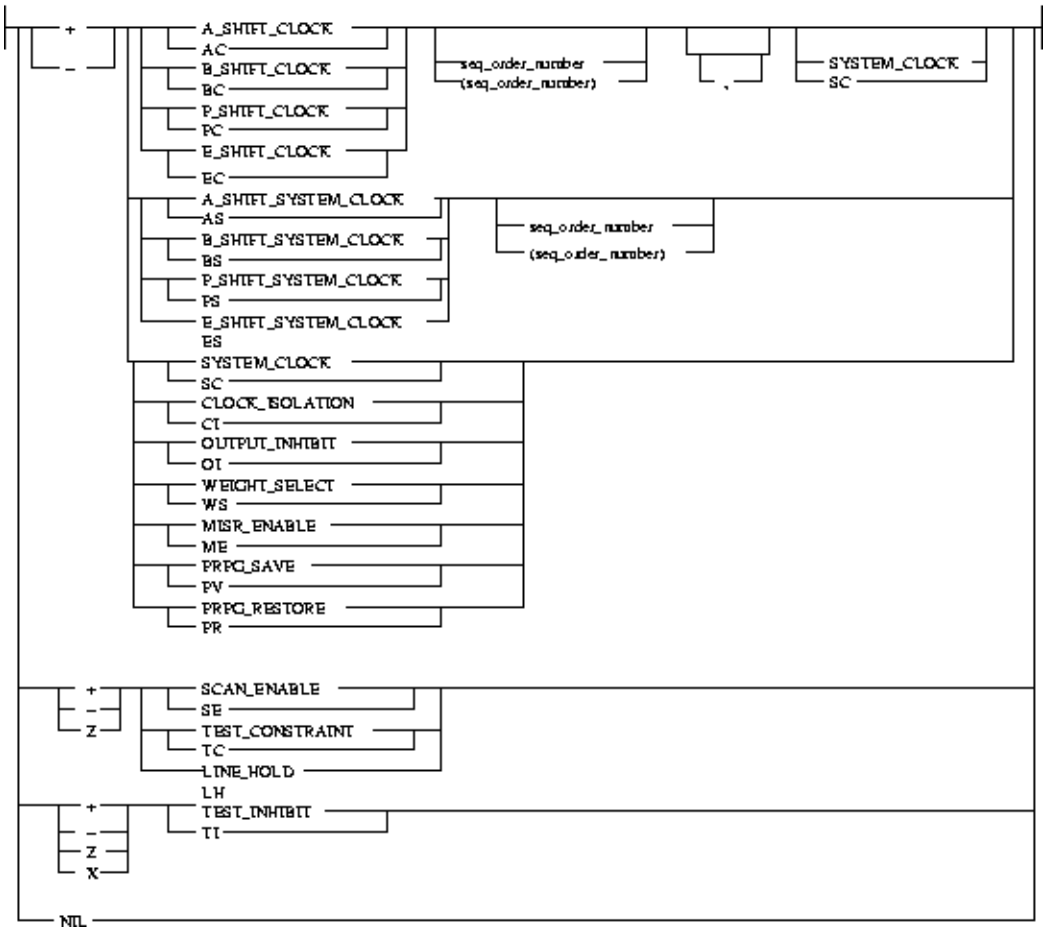
PI/PO Functions:



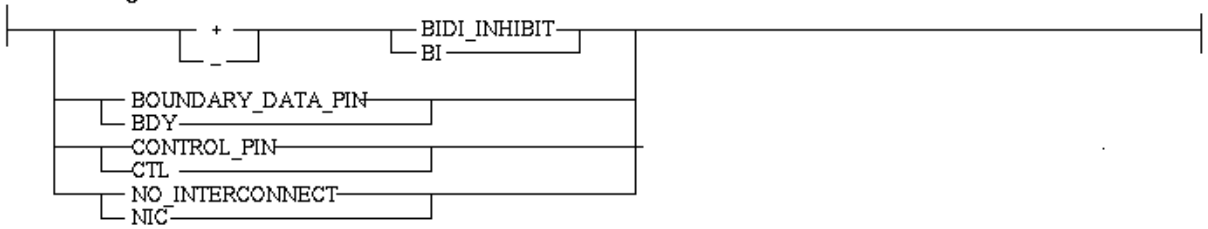
Latch Functions:



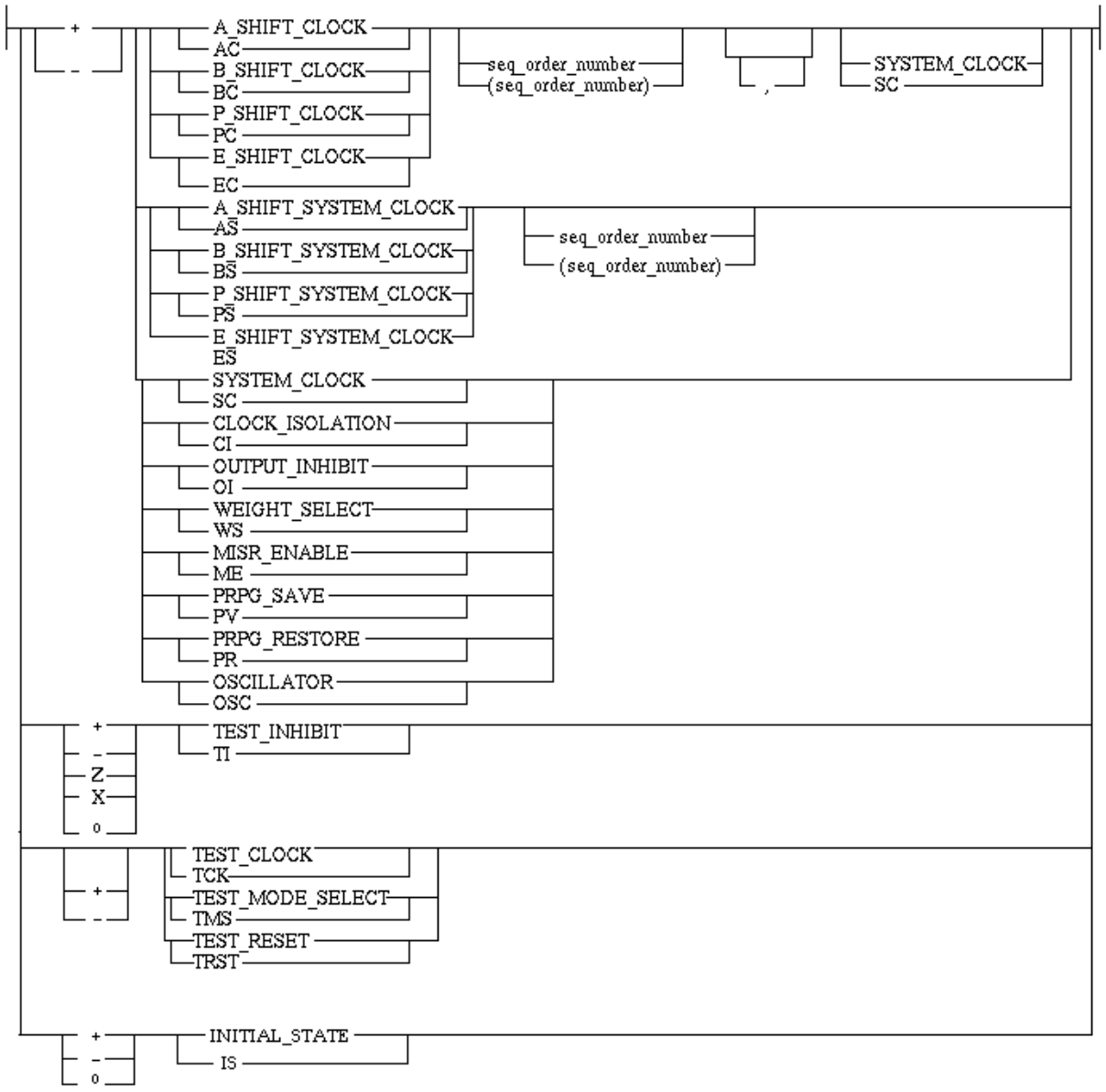
PPI Functions:



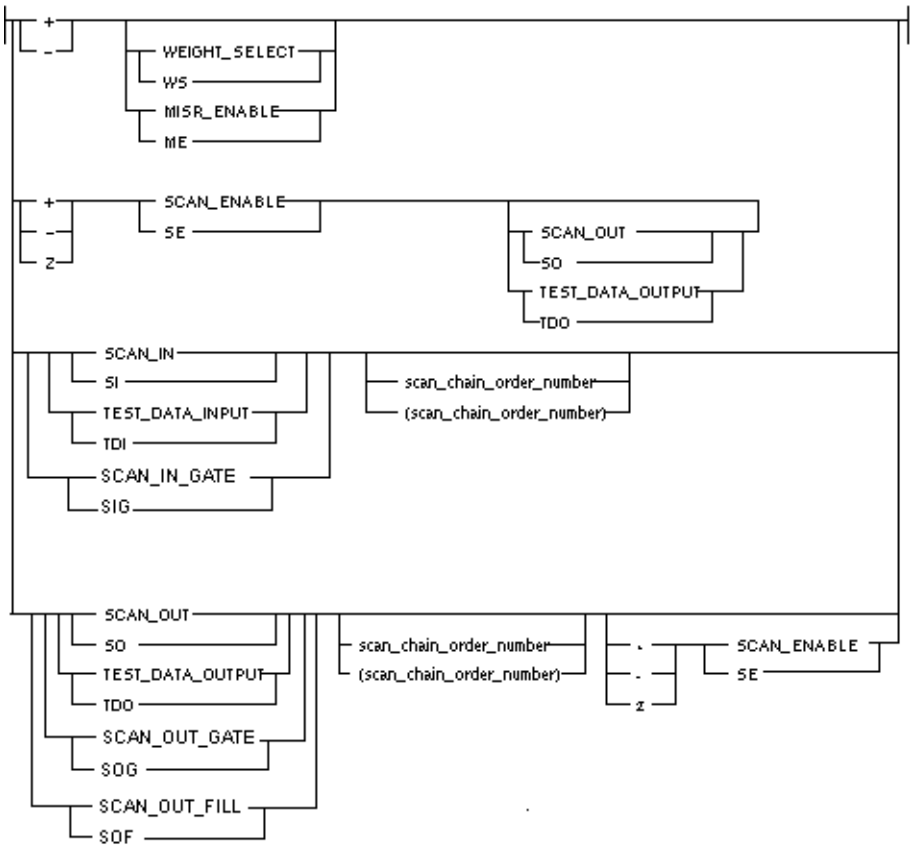
manufacturing:



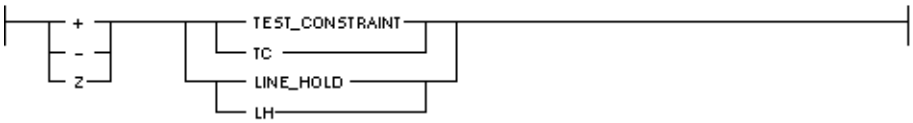
global:



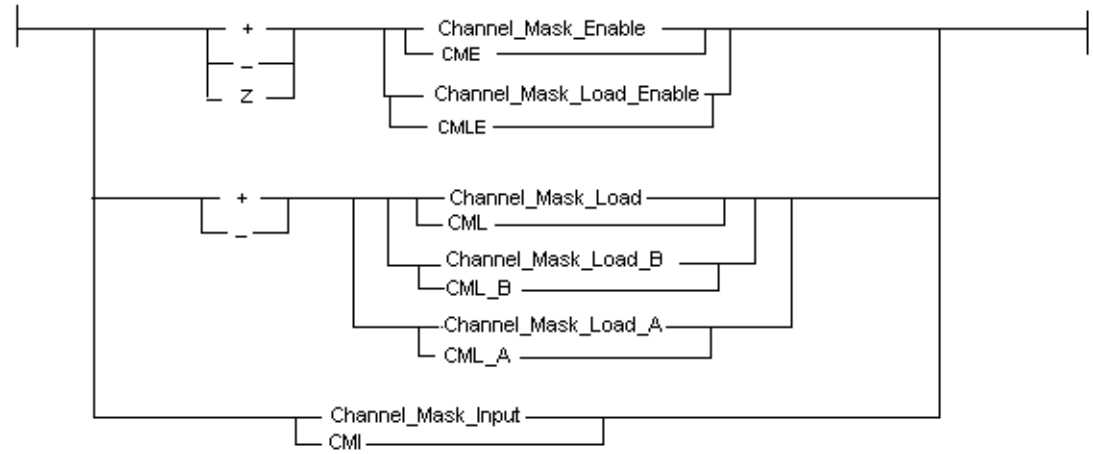
scan:



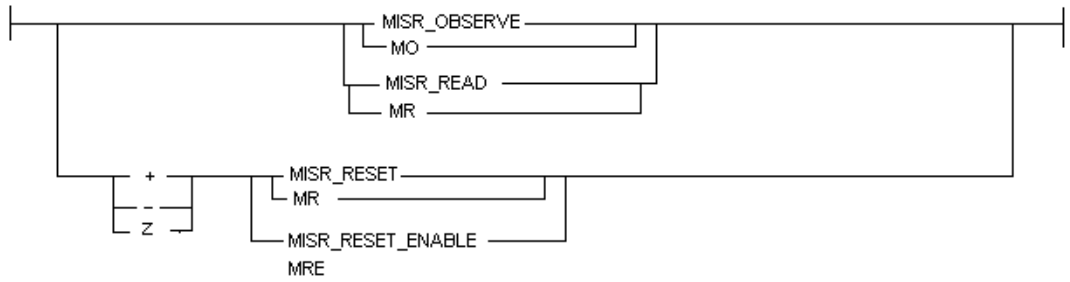
non-scan:



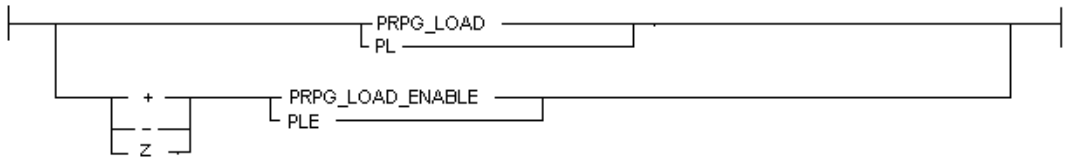
Channel Mask Functions:

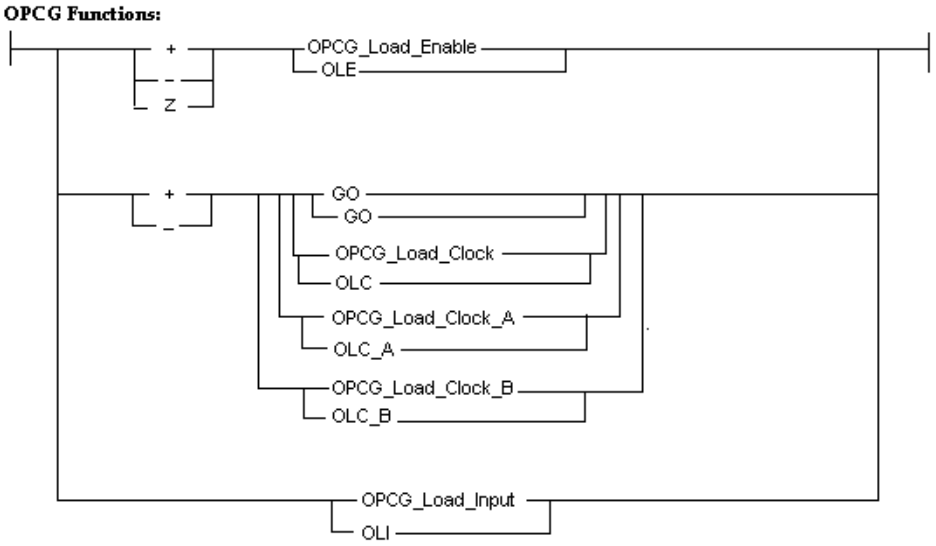


MISR Functions:



PRPG Functions:





In this statement:

- `PIN -` Indicates that the specified test function, that is, the data following the `TEST_FUNCTION` keyword, is to be put on a specific pin identified by *`pinName`*.

  - *`pinName`*

    Identifies the pin whose test function is being specified.

  Note: You must enclose pin names using a double quotes escaped identifier if the first character is something other than:

  ```
  A-Z or a-z
  0-9
  ```

  Or subsequent characters are something other than:

  ```
  A-Z or a-z
  0-9
  _
  ( )
  #
  +
  -
  &
  /
  :
  .
  { }
  |
  [ ]
  @
  ```

For example:

```
abc
abc_
abc(0)
"&abc"
"01"
01(a)
"net a"
are all legal pin names.

&abc
abc?
a%5
abc ef
```

are not legal unless enclosed in double quotes. Pin names themselves can never contain a double quote (") or a newline (ln) character.

- NET - Indicates that the specified test function, that is, the data following the TEST_FUNCTION keyword, is to be put on a specific net identified by *netName*.

  - *netname*

    Identifies the net whose test function is being specified.

  Note: You must enclose netNames using a double quotes escaped identifier if the first character is something other than:

  ```
  A-Z or a-z
  0-9
  ```

  Or subsequent characters are something other than:

  ```
  A-Z or a-z
  0-9

  _
  ( )
  #
  +
  -
  &
  /
  :
  .
  { }
  |
  [ ]
  @
  ```

  For example:

  ```
  abc
  abc_
  abc(0)
  "&abc"
  "01"
  ```

```
01(a)
"net a"
```

are all legal net names.

```
&abc
abc?
a%5
abc ef
```

are not legal unless enclosed in double quotes. Netnames themselves can never contain a double quote (") or a newline (ln) character.

- `PPI -` Indicates that the specified test function, that is, the data following the `TEST_FUNCTION` keyword, is to be put on a specified ppi identified by *`ppiName`*.
- `BLOCK -` Indicates that the specified test function, that is, the data following the `TEST_FUNCTION` keyword, is to be put on a specified block identified by *`blockName`*.
- `PIPELINE_DEPTH -` A pipeline depth value may be specified on any of the following test function pins:
  - `BDY`, refer to BOUNDARY_DATA_PIN
  - `CTL`, refer to CONTROL_PIN
  - `CI`, refer to CLOCK_ISOLATION
  - `CME`, refer to CHANNEL_MASK_ENABLE
  - `CMI`, refer to CHANNEL_MASK_INPUT
  - `WS`, refer to WEIGHT_SELECT
  - `ME`, refer to MISR_ENABLE
  - `SE`, refer to SCAN_ENABLE
  - `SIG`, refer to SCAN_IN_GATE
  - `SIS`, refer to SCAN_IN_SELECTOR
  - `SI`, refer to SCAN_IN
  - `SO`, refer to SCAN_OUT
  - `SOG`, refer to SCAN_OUT_GATE
  - `SOS`, refer to SCAN_OUT_SELECTOR
  - `TC`, refer to TEST_CONSTRAINT
  - `LH`, refer to LINEHOLD

- `PIPELINE_CLOCK -` Specify the pipeline clock attribute on a scan clock. This attribute indicates which of the scan clocks operates the control pipelines. Refer to "Clock Functions" on page 34 for additional information.

- `DOMAIN -` Indicates that clock domains will follow. More than one clock domain can be specified in a comma-separated list. The list will replace any clock domain attribute value for this object that was specified in the model source. The syntax is:

  `domain=`*`character_string`*`[,`*`character_string`*`]...`

  where *`character_string`* is an arbitrary name assigned by the user.

  This keyword is accepted for nets and pins. Be careful about specifying a multi-source net or a pin that is a source on a multi-source net, as this could cause unexpected results.

  Aliases for this keyword are `clk_domain` and `clk domain`.

  The clockdomain attribute specified in the model source cannot be erased, it can only be replaced. Replacing it with an artificial clock domain will yield similar checking results as removing it entirely.

- `TEST_FUNCTION -` Indicates that the test function will follow. The use of this keyword is only for readability.

  Note:

  - You may use `1` and `0` in place of `+` and `-`, respectively, for any flag that has a sign preceding it.
  - A `1` or a `0` can have either a blank or non-blank following it, for example, 1 AC or 1AC (+AC is also valid). This is also true for the Z (high impedance) logic value as a test function pin assignment, for instance, either Z SE or ZSE.
  - The values are case-insensitive.

  The test function is one or more of the following, and the allowed combinations are illustrated in the ASSIGN syntax diagram.

  - `NIL`

    Specifies that any existing test function should be removed from the pin.

  - `NO_INTERCONNECT`

    See NO_INTERCONNECT (NIC) on page 48 for detail.

  - `BIDI_INHIBIT`

    See `BIDI_INHIBIT` (BI) on page 42 for detail.

  - `BOUNDARY_DATA_PIN`

See BOUNDARY_DATA_PIN (BDY) on page 48 for detail.

- CONTROL_PIN

  See CONTROL_PIN (CTL) on page 48 for detail.

- A_SHIFT_CLOCK

  See A_SHIFT_CLOCK (AC) on page 35 for detail.

- B_SHIFT_CLOCK

  See B_SHIFT_CLOCK (BC) on page 35 for detail.

- P_SHIFT_CLOCK

  See P_SHIFT_CLOCK (PC) on page 35 for detail.

- E_SHIFT_CLOCK

  See E_SHIFT_CLOCK (EC) on page 36 for detail.

- seq_order_number

  See Scan Clock Sequence Number on page 36 for detail.

- SYSTEM_CLOCK

  See SYSTEM_CLOCK (SC) on page 34for detail.

- A_SHIFT_SYSTEM_CLOCK

  See A_SHIFT_SYSTEM_CLOCK (AS) on page 35 for detail.

- B_SHIFT_SYSTEM_CLOCK

  See B_SHIFT_SYSTEM_CLOCK (BS) on page 35 for detail.

- P_SHIFT_SYSTEM_CLOCK

  See P_SHIFT_SYSTEM_CLOCK (PS) on page 35 for detail.

- E_SHIFT_SYSTEM_CLOCK

  See E_SHIFT_SYSTEM_CLOCK (ES) on page 36 for detail.

- CLOCK_ISOLATION

  See CLOCK_ISOLATION (CI) on page 38 for detail.

- OUTPUT_INHIBIT

  See OUTPUT_INHIBIT (OI) on page 42 for detail.

- WEIGHT_SELECT

  See WEIGHT_SELECT (WS) on page 43 for detail.

- MISR_ENABLE

  See MISR_RESET_ENABLE (MRE) on page 44 for detail.

- PRPG_SAVE

  See PRPG_SAVE (PV) on page 43 for detail.

- PRPG_RESTORE

  See PRPG_RESTORE (PR) on page 43 for detail.

- OSCILLATOR

  See OSCILLATOR (OSC) on page 37 for detail.

- SCAN_ENABLE

  See SCAN_ENABLE (SE) on page 38 for detail.

- TEST_INHIBIT

  See TEST_INHIBIT (TI) on page 39 for detail.

- TEST_CONSTRAINT

  See TEST_CONSTRAINT (TC) on page 40 for detail.

- TEST_CLOCK

  See TEST_CLOCK (TCK) on page 49 for detail.

- TEST_MODE_SELECT

  See TEST_MODE_SELECT (TMS) on page 49 for detail.

- TEST_RESET

  See TEST_RESET (TRST) on page 49 for detail.

- FINITE_STATE_MACHINE

  See FINITE_STATE_MACHINE (FSM) on page 41 for detail.

- LINEHOLD

  See LINEHOLD (LH) on page 40 for detail.

- FIXED_VALUE_LATCH_LINEHOLD

See FIXED_VALUE_LATCH_LINE_HOLD (FLH) on page 41 for detail.

- SCAN_IN

  See SCAN_IN (SI) on page 32 for detail.

- TEST_DATA_INPUT

  See TEST_DATA_INPUT (TDI) on page 49 for detail.

- Scan Chain Order Number

  See Scan Chain Sequence Order on page 33 for detail.

- SCAN_OUT

  See SCAN_OUT (SO) on page 33 for detail.

- TEST_DATA_OUTPUT

  See TEST_DATA_OUTPUT (TDO) on page 49 for detail.

- CHANNEL_INPUT

  See CHANNEL_INPUT (CHI) on page 33 for detail.

- CHANNEL_OUTPUT

  See CHANNEL_OUTPUT (CHO) on page 33 for detail.

- MISR_OBSERVE

  See MISR_OBSERVE (MO) on page 44 for detail.

- MISR_RESET

  See MISR_RESET (MRST) on page 44 for detail.

- MISR_RESET_ENABLE

  See MISR_RESET_ENABLE (MRE) on page 44 for detail.

- MISR_READ

  See MISR_READ (MRD) on page 44 for detail.

- SCAN_IN_GATE

  See SCAN_IN_GATE (SIG) on page 45 for detail.

- SCAN_OUT_GATE

  See SCAN_OUT_GATE (SOG) on page 45 for detail.

- SCAN_OUT_FILL

  See SCAN_OUT_FILL (SOF) on page 45 for detail.

- PRPG_LOAD

  See PRPG_LOAD (PLD) on page 43 for detail.

- PRPG_LOAD_ENABLE

  See PRPG_LOAD_ENABLE (PGE) on page 43 for detail.

- Channel_Mask_Input

  See Channel_Mask_Input (CMI) on page 45 for detail.

- Channel_Mask_Enable

  See Channel_Mask_Enable (CME) on page 45 for detail.

- Channel_Mask_Load_Enable

  See Channel_Mask_Load_Enable (CMLE) on page 46 for detail.

- Channel_Mask_Load

  See Channel_Mask_Load (CML) on page 46 for detail.

- Channel_Mask_Load_A

  See Channel_Mask_Load_A (CML_A) on page 46 for detail.

- Channel_Mask_Load_B

  See Channel_Mask_Load_B (CML_B) on page 46 for detail.

- Initial_State

  See INITIAL_STATE (IS) on page 52 for detail.

- GO

  See GO on page 47 for detail.

- OPCG_Load_Input

  See OPCG_Load_Input (OLI) on page 47 for detail.

- OPCG_Load_Enable

  See OPCG_Load_Enable (OLE) on page 47 for detail.

- OPCG_Load_Clock

See OPCG_Load_Clock (OLC) on page 47 for detail.

- `OPGG_Load_Clock_A`

  See `OPCG_Load_Clock_A (OLC_A)` on page 47 for detail.

- `OPCG_Load_Clock_B`

  See `OPCG_Load_Clock_B (OLC_B)` on page 47 for detail.

ASSIGN Statement Syntax Techniques

This section lists techniques, available syntax options, and examples for the ASSIGN statement.

`ASSIGN` *`object_type`* `=` *`objectname`* *`assignmentInfo`* `;`

Note: A semi-colon (`;`) is required for all statements.

`object_type` is REQUIRED and must be one of the following:

- `PIN` - Must be fully spelled (mixed case is allowed)
- `NET` - Must be fully spelled (mixed case is allowed)
- `BLOCK` - Must be fully spelled (mixed case is allowed)
- `PPI` - Must be fully spelled (mixed case is allowed)

Tip

For Verify Test Structures (TSV), objects that are vectors must be referenced as individual bits. For example:

`ASSIGN PIN=SCAN_OUT[7] ...` is valid

`ASSIGN PIN=SCAN_OUT[7:0] ...` is invalid

Equals (=) is OPTIONAL. If specified, the "`=`" character must be used.

`objectname` is REQUIRED and must be one of the following:

- A quoted name - Special characters are allowed.
- An unquoted name - Special characters are disallowed.

  Note:

  - The *`objectname`* should be a pin name if the object_type is `PIN`.

- The *objectname* should be a net name if the object_type is NET.
- The *objectname* should be a block name if the object_type is BLOCK.
- The *objectname* should be a ppi name if the object_type is PPI.

assignmentInfo can be ONE OR MORE assignment groups whereas assignment group can be one of the following:

- Test Functions assignment
- Clock Domain
- Pipeline Depth
- Pipeline Clock designation

For assignment of Test Functions:

TEST_FUNCTION = *test function list*

Specify Test Function using of the following forms:

- TEST_FUNCTION (mixed case is allowed)
- TEST FUNCTION (mixed case is allowed)
- FUNCTION (mixed case is allowed)
- TFUNC (mixed case is allowed)
- (blank)
- If specified (non-blank)
- Mixed case, i.e. Tfunc,TfUnC, etc.)

Equals (=) is OPTIONAL but cannot be specified if the test function is blank. If specified, use the "=" character.

test function list consists of one or more test functions applicable for the object type. Each test function may be separated by a space or comma. A test function MAY REQUIRE a polarity (sign/stability) value, i.e. +SG; or it may not require a polarity, i.e. SO.

For a Clock Domain:

CLK_DOMAIN = *clock domain data*

The Clock Domain specification must start with one of the following:

- `CLK_DOMAIN` (mixed case is allowed)
- `CLK DOMAIN` (mixed case is allowed)
- `CLKDOMAIN` (mixed case is allowed)
- `DOMAIN` (mixed case is allowed)

`Equals (=)` is OPTIONAL. If specified, use the "=" character

`clock domain data` is one OR more comma-separated values. A value may be a string, a quoted string, or a number. The comma is required if more than one value is specified.

For Pipeline Depth:

`PIPELINE_DEPTH =` *`pipeline depth data`*

Pipeline Depth must start with one of the following:

- `PIPELINEDEPTH` (mixed case is allowed)
- `PIPELINE_DEPTH` (mixed case is allowed)
- `PIPELINE DEPTH` (mixed case is allowed)

`Equals (=)` is OPTIONAL. If specified. use the "=" character.

`pipeline depth data` can be a string, a quoted string, or a number.

Pipeline clock designation:

`PIPELINE_CLOCK`

Specify a pipeline clock using one of the following conventions:

- `PIPELINECLOCK` (mix case is allowed)
- `PIPELINE_CLOCK` (mix case is allowed)
- `PIPELINE` CLOCK (mix case is allowed)

The following examples assign information to a pin. Blocks, nets, ppis, or symbols may be assigned the same way.

Test Function examples:

```
ASSIGN PIN = A1 Test_Function=-TC;
ASSIGN PIN A1 -TC; Note: shortest form: optional =, optional test_function
```

Other Test Function examples:

```
ASSIGN PIN = A1 Test_Function=-TC;
ASSIGN PIN = A1 TESt_FunCtion=-TC;
ASSIGN PIN = A1 Test_Function=-TC;
ASSIGN PIN = A1 Tfunc=-TC;
ASSIGN PIN = A1 Tfunc=-TC;
ASSIGN PIN A1 Tfunc=-TC;
ASSIGN PIN = A1 -TC;

ASSIGN PIN = A1 Test_Function= SI;
ASSIGN PIN = 01 Test_Function= SO;

ASSIGN PIN = A1 -TC -CI; Note:shortest form with two test functions
```

Clock Domain examples:

```
ASSIGN PIN A1 -TC CLK_DOMAIN=2; Note: has test function and clock domain
ASSIGN PIN A1 CLK DOMAIN=2;     Note: no test function
ASSIGN PIN A1 CLKDOMAIN=2;
ASSIGN PIN A1 DOMAIN=2;
```

Pipeline Depth examples:

```
ASSIGN PIN A1 -TC PIPELINEDEPTH=2; Note: has test function and pipeline depth
ASSIGN PIN A1 PIPELINE_DEPTH=2;    Note: no test function
ASSIGN PIN A1 PIPELINE DEPTH=2;
```

Pipeline Clock examples:

```
ASSIGN PIN A1 -ES PIPELINECLOCK; Note: has test function and pipeline clock
ASSIGN PIN A1 PIPELINE_CLOCK;    Note: no test function
ASSIGN PIN A1 PIPELINE CLOCK;
```

Other examples

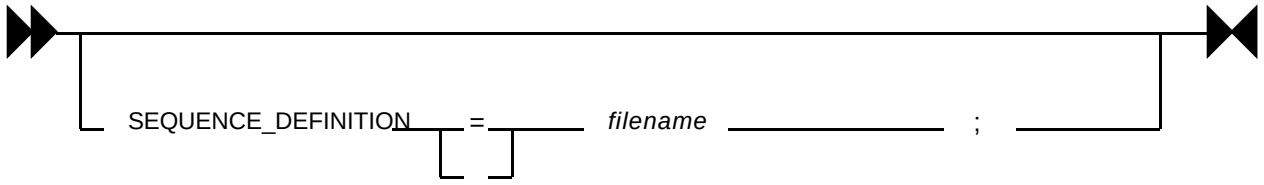
```
ASSIGN PIN = "A01" TEST_FUNCTION = +ES +MRST;
ASSIGN PIN "A02"  pipeline_depth=12 DOMAIN=12 -TC;
```

SEQUENCE_DEFINITION

This statement specifies a file name that contains predefined input pattern sequences for a test mode.

An example is the definition of a custom scan sequence for a part. Another is the specification of a mode initialization sequence definition required for test modes that have non-scannable latches that must be initialized at the beginning of the test mode. Such latches would include fixed-value latches and the latches that make up the LFSRs used in LBIST. Refer to “Mode

Initialization Sequences (Advanced)" on page 66 for more information about coding sequence definitions. Also refer to "Custom Scan Sequences (Advanced)" on page 70.

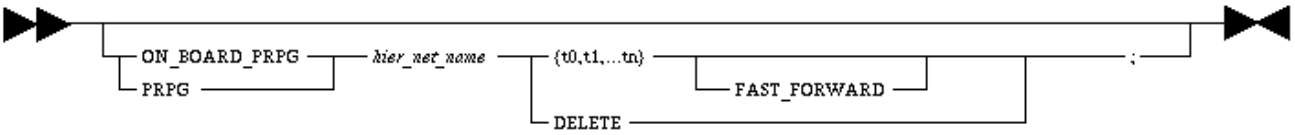


In this statement:

filename The name of the file containing the sequence definition statements for the test mode. If the file name starts with a slash, it is assumed to be a fully specified path; otherwise, it is assumed a file of this name is found in the path specified in the SEQPATH parameter.

ON_BOARD_PRPG

This optional statement may either define an on-design pseudo-random pattern generator (PRPG) or disable a PRPG that is defined in the model. PRPGs are used as test pattern sources for LBIST.



In this statement:

- *hier_net_name*

  The name of any net driven only by the latch primitive that models the last cell of the PRPG. The last cell is considered to be the right-most cell where the PRPG shifts from left to right. The *Build Test Mode* function traces the design to identify the other latches comprising the PRPG. Refer to "Self Test Structures" in the *Encounter Test: Reference: Legacy Functions* for additional information.

- t0,t1,...,tn

  LFSR feedbacks (tap positions). Each *ti* (0 <=i <=n) is an integer denoting a cell in the LFSR. The LFSR cells are numbered with the left-most cell as cell 1 and the right-most cell as cell m, where m is the total number of cells and the LFSR shifts from left to right.

Encounter Test supports two LFSR implementations, called multiplier LFSR and divisor LFSR. The implementation is implied by the value of t_0; t_0 must be zero for a divisor LFSR and >0 for a multiplier LFSR. For both implementations, the length m is defined by t_n, where t_n = m. The intermediate cell numbers, t_1, t_2,... must be specified in order of increasing value. For a multiplier LFSR, the cells $t_0, t_1, ..., t_n$ are the inputs to an XOR function that feeds the first cell of the LFSR. For a divisor LFSR, cell t_n (that is, cell m) is fed directly into the input of cell 1 and the cells $t_1, t_2, ... t_{n-1}$ are each individually XORed with cell t_n and the XOR(t_i, t_n) is fed into cell t_i +1. The VIM syntax of this attribute is `PRPG = (t 0, t 1,..., t n)`.

- `FAST_FORWARD`

  Specify whether the design contains registers that are used to save the PRPG seed during each scan operation and restore it for the next scan operation. This technique allows unproductive tests to be skipped, thereby saving the time that would be required to do the scan, without invalidating the computed fault coverage.

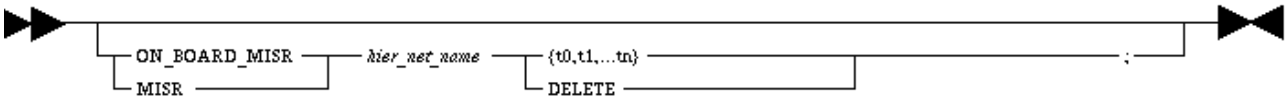
  Note: If you use `FAST_FORWARD` for one PRPG, you must use it for all of them.

- `DELETE`

  This keyword specifies that this latch is not to be considered part of a PRPG. This form of the statement would be used where the logic model contains a PRPG definition at this latch, and the PRPG definition is to be ignored.

ON_BOARD_MISR

This optional statement may either define an on-design signature register MISR or disable a MISR that is defined in the model. MISRs are used to compress test pattern responses for LBIST.



In this statement:

- *`hier_net_name`*

  The name of any net driven only by the latch primitive that models the last cell of the MISR. The last cell is considered to be the right-most cell where the MISR shifts from left to right. The *Build Test Mode* function traces the design to identify the other latches comprising the MISR. Refer to "Modeling Self Test Structures" in the *Encounter Test: Reference: Legacy Functions* for additional information.

- `t0,t1,...,tn`

  LFSR feedbacks (tap positions). Each t_i (0 <=i <=n) is an integer denoting a cell in the LFSR. The LFSR cells are numbered with the left-most cell as cell 1 and the right-most cell as cell m, where m is the total number of cells and the LFSR shifts from left to right. Encounter Test supports two LFSR implementations, called multiplier LFSR and divisor LFSR. The implementation is implied by the value of t_0; t_0 must be zero for a divisor LFSR and > 0 for a multiplier LFSR. For both implementations, the length m is defined by t_n, where t_n = m. The intermediate cell numbers, t_1, t_2,... must be specified in order of increasing value. For a multiplier LFSR, the cells $t_0, t_1, ..., t_n$ are the inputs to an XOR function that feeds the first cell of the LFSR. For a divisor LFSR, cell t_n (that is, cell m) is fed directly into the input of cell 1 and the cells $t_1, t_2, ... t_{n-1}$ are each individually XORed with cell t_n and the XOR(t_i, t_n) is fed into cell t_i + 1. The VIM syntax of this attribute is MISR = (t_0, t_1,..., t_n).

- `DELETE`

  This keyword specifies that this latch is not to be considered part of a MISR. This form of the statement would be used where the logic model contains a MISR definition at this latch, and the MISR definition is to be ignored.

  `DELETE` overrides the `MISR`, `TB_MISR`, and any properties specified by the `MISR_PROPERTIES` statement.

COMETS

This optional statement specifies that the test mode belongs to one or more groups, called comets (collection of modes for estimating test coverage). Two types of comets are defined. One type, called a STATS_ONLY comet, allows the reporting of test coverage summaries for the cumulative effect of all the tests for the member test modes. The other type, called a MARKOFF comet, has all the features of a `STATS_ONLY` comet but also allows cross-mode fault markoff to occur. Cross-mode fault markoff is a time-saving technique used when faults can be tested in more than one test mode, to prevent unnecessary work to test those faults in every test mode. When a MARKOFF comet is defined, then the testing of a fault in one of its member test modes causes it to appear as tested in other test modes belonging to that comet if the fault is marked *tested* in some test mode for each markoff comet that the other test mode belongs to.

See section Report Fault Coverlage Statistics in *Encounter Test: Guide 4: Faults* for a more detailed explanation of markoff comets.

A test mode may belong to several comets, although the maximum number of comets allowed for a design is 64. By default, a test mode is always assigned to a markoff comet that has the same name as the TDR for the test mode, unless `FAULTS=NONE` is specified or defaulted.

Care must be exercised in defining markoff comets or you will not get the results you expect. Two of the points mentioned above often cause confusion:

- *tested in another mode* applies only if for every MARKOFF comet the mode belongs to, the fault is tested in some mode that belongs to that comet. Thus, in a complicated case where there are multiple MARKOFF comets with overlapping member test modes, a fault may have to be tested in two or more modes before the cross-mode markoff can take effect for that fault in the remaining test modes.
- By default, there is a MARKOFF comet defined for each TDR, whose members are all test modes that point to that TDR. This can get in the way, and the remedy is to disable cross-mode markoff by defining a `STATS_ONLY` comet with the same name as the TDR.

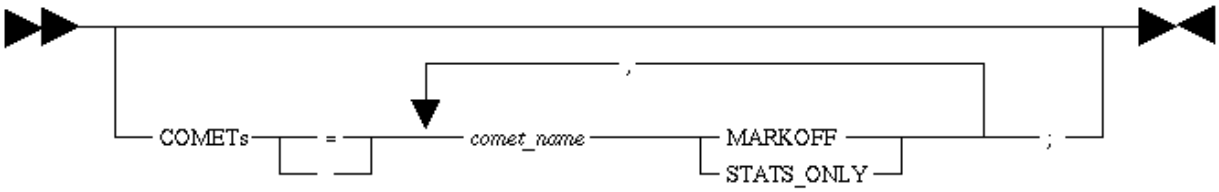
To use cross-mode fault markoff, do the following while laying out the methodology and before any test modes are actually built:

1. Make a list of the TDRs you need.
2. Make a list of the MARKOFF comets you need.
3. Make a list of the test modes you need.
4. Create a matrix to define which MARKOFF comet(s) each test mode belongs to, and another matrix to define which TDR each test mode points to.
5. For each TDR, determine whether it is congruent with some MARKOFF comet (they have exactly the same list of member test modes). If a match is found, that TDR comet can be the MARKOFF comet. If no match is found, then that TDR comet must be defined as `STATS_ONLY`.
6. Explicitly define each MARKOFF comet that is not congruent with some TDR.

Example

Suppose a test mode belongs to two MARKOFF comets, “haleys” and “santas”, and uses the “speedy” TDR. The test mode definition should contain the following COMET statement:

```
COMETS = haleys MARKOFF,
    santas MARKOFF,
    speedy STATS_ONLY;
```



In this statement:

comet_name
: The name of the comet. This is an arbitrary name you assign when building the first test mode that is a member of the comet. Every test mode that belongs to the comet will refer to it by this same name.

- MARKOFF

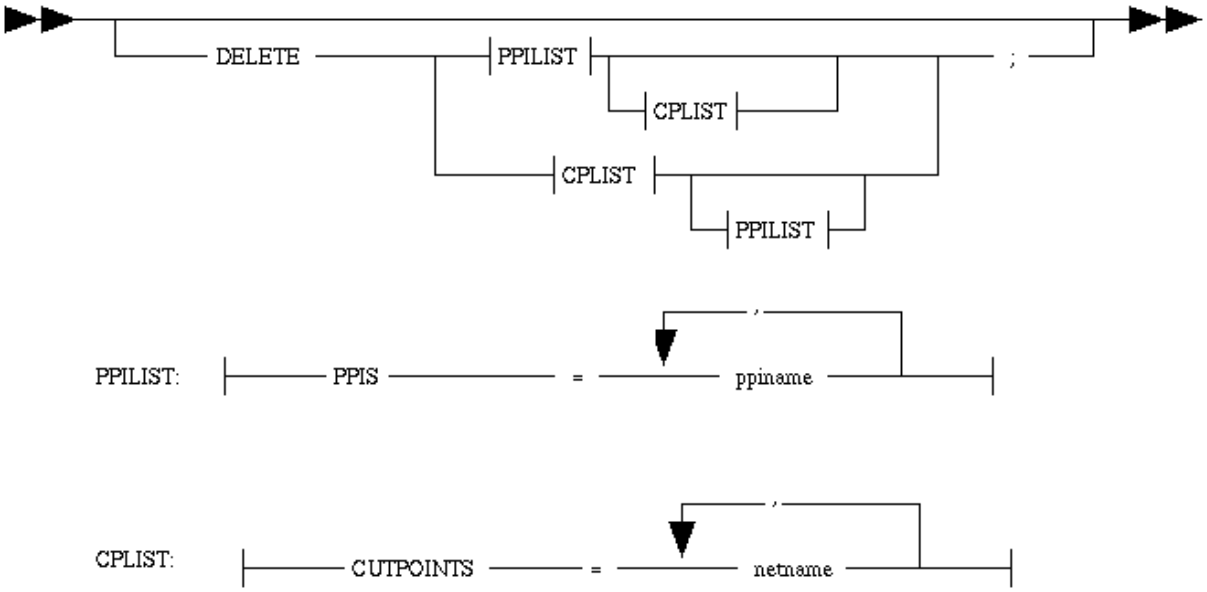
  Specifies that this comet supports cross-mode fault markoff.

- STATS_ONLY

  Specifies that this comet does not support cross-mode fault markoff, and is useful only for test coverage statistics reporting. There is no restriction on the number of STATS_ONLY comets a test mode may belong to. There is also a “virtual” STATS_ONLY comet for the global design.

DELETE

This optional statement specifies properties or objects to be “deleted” or ignored for this test mode. There is no default.



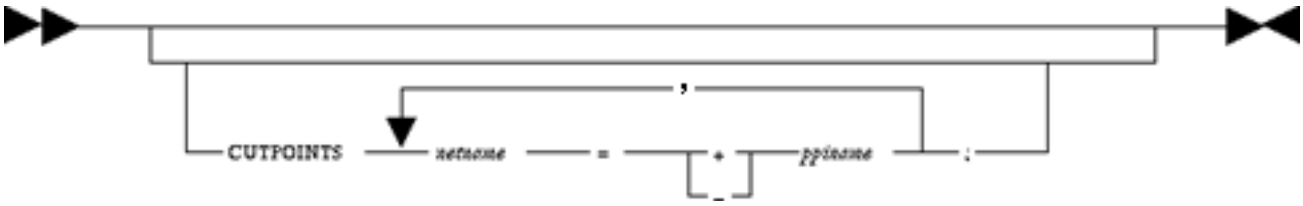
In this statement:

`CPLIST`	Specify a list of netnames that are defined as cut points in the design source and are not to be treated as cut points. CUTPOINT attributes on the listed nets will be ignored.
`PPILIST`	Specify a list of pseudo primary inputs that are defined in the Design source and are not to be used. Deleting a pseudo PI automatically deletes all references to it, so any cut points defined by reference to a deleted pseudo PI will not be considered to be cut points unless they are redefined (through the mode definition CUTPOINTS statement).

CUTPOINTS

This optional statement identifies internal nets in the product that are to be treated by Encounter Test as primary inputs. It associates each such net with a conceptual primary input (pseudo PI) through which the state of the net is specified for all Encounter Test processing. There is a 1:many relationship between pseudo PIs and cut points; several cut point nets that have a common logical signal are tied together by reference to the same pseudo PI. A cut

point is defined so that its state is either the true or the complement of the state of its associated pseudo PI.



In this statement:

netname	The hierarchical name of a net in the design
ppiname	The name of the pseudo primary input (defined either in the design source or implicitly by reference in this statement) which Encounter Test will use to "control" this net. The associated sign tells whether to invert the pseudo primary input value when placing the value on the net.

The following is an example CUTPOINTS statement:

```
CUTPOINTS  "netnameA" = +ppi_name,
"netnameB" = +ppi_name,
"netnameC" = +ppi_name;
assign ppi=ppi_name test_function=+ES;
```

Tip

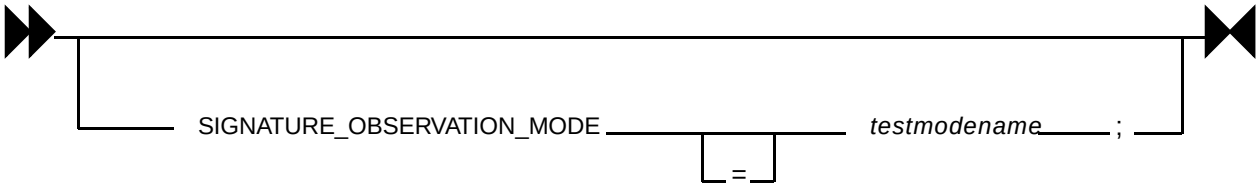
Quotations are required around net names for a multiple cutpoint definition.

SIGNATURE_OBSERVATION_MODE

This statement is used in the automatic creation of a signature observation sequence. It tells Encounter Test the name of the test mode to use for scanning out the results of the BIST. If you are supplying the signature observation sequence definition, then this mode definition statement is not needed. You must supply the signature observation sequence definition if it does not use a scan operation or if switching to the observation test mode requires any sequencing. Encounter Test can automatically create the signature observation sequence only if switching to the scan mode is the trivial operation of switching (in any arbitrary order) a set of control pins labeled as Clock, TI, and TC in the observation mode.

For OPMISR or OPMISRplus test modes, this statement indicates that MISR observation should be done serially by scanning out the MISR using the scan chains defined by the specified test mode.

Refer to "Signature Observation Sequences" in the *Encounter Test: Reference: Legacy Functions* for related information.

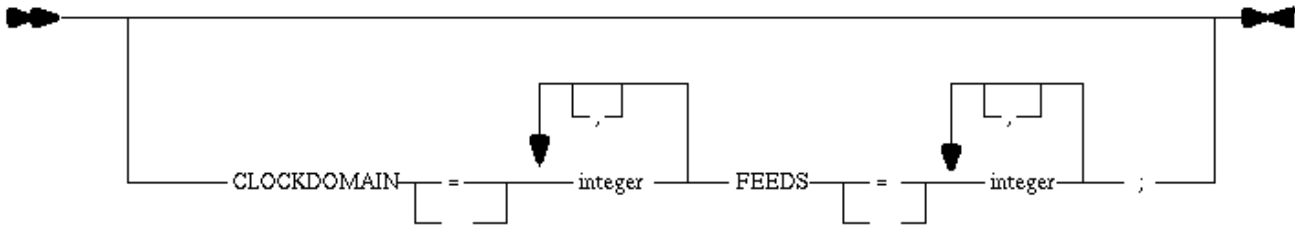


In this statement:

testmodename	Specify the name of the observation test mode to be used for scanning out the BIST results. The BIST results consist of the MISR contents and, for diagnostics, the channel latch states.

CLOCKDOMAIN

This optional statement is used only by the clock domain check in Test Structure Verification.



In this statement:

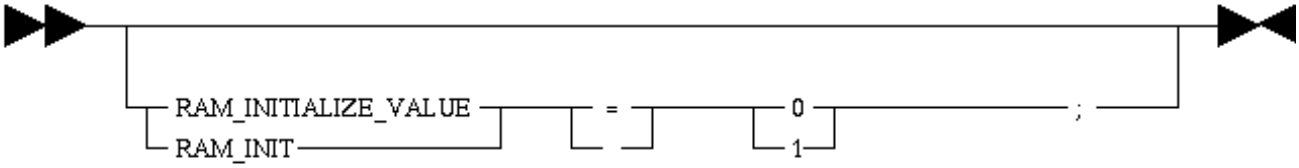
integer	Specify a list of clock domain names.

Each clock domain in the list on the left (following the CLOCKDOMAIN keyword) is allowed to feed, through the data path or clock gating, latches in the clock domains in the right-hand list

(following the `FEEDS` keyword). Use several `clockdomain` statements where you have multiple domains that feed unique sets of other domains.

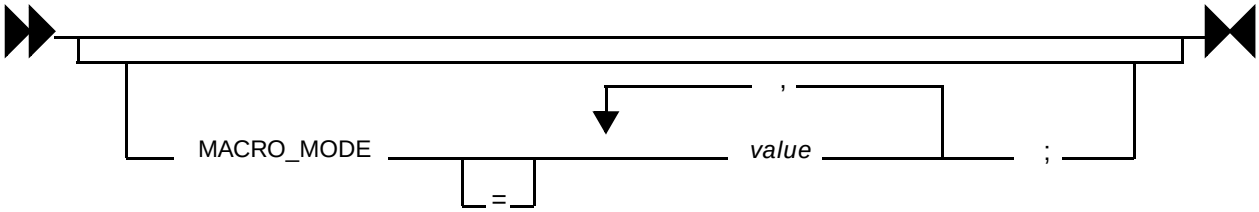
RAM_INITIALIZE_VALUE

This optional statement specifies that all arrays will be initialized to the specified value at the start of the mode initialization sequence. The value used to initialize the arrays during `gp` or `hsscan` simulation will be the value at the end of the simulation of the mode initialization sequence. If this statement is not included, the default is to start the mode initialization sequence simulation with the state of the arrays at X.



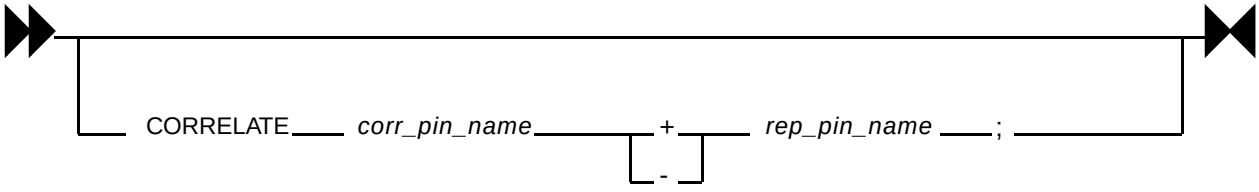
MACRO_MODE

This optional keyword specifies whether a macro should be processed in Verify Core Isolation and Create Core Tests.



CORRELATE

The CORRELATE statement supports the correlation of pins within the mode definition.



Two pin names are specified, for example:

```
CORRELATE corr_pin_name + rep_pin_name
```

- *corr_pin_name*

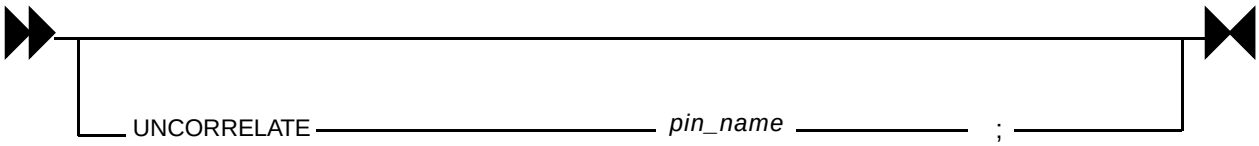
  Specify the name of the correlated pin (also referred to as the dependent pin).

- *rep_pin_name*

  Specify the name of the representative pin.

A “+” between the pin names denotes correlated in-phase; a “-” denotes correlated out-of-phase. There is no default, either “+” or “-” must be specified.

The UNCORRELATE statement removes pin correlation.



Example syntax:

```
CORRELATE in2 +in1;
CORRELATE out2 -out1;
UNCORRELATE in2s;
CORRELATE in799 -in1;
CORRELATE in5c +in5;
CORRELATE in6c +in3;
UNCORRELATE in9;
CORRELATE out2 -out1;
```

Maintain awareness of the following when using the CORRELATE statement:

During test mode creation:

- All pins referenced in the CORRELATE statement must be a PI, PO, or technology I/O cell. If they are not a PI, PO or technology I/O cell, an error message is issued and the statement is ignored.
- The test mode creation process disallows test functions to be defined on newly defined (in the mode definition file) correlated pins. If found in a new CORRELATE statement, an information message is issued and the pin is removed from the Test Function Pin list.
- If a pin is defined to be a dependent pin, it cannot appear as a dependent pin nor a representative pin in any other CORRELATE statement. The first statement sets the pin characteristic as either correlated or representative.

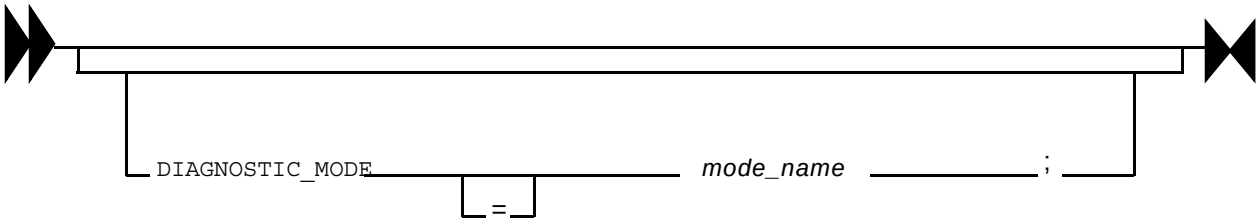
Interaction with the logic model:

- If the dependent pin in the CORRELATE statement is defined as a representative pin through the CORRELATE attribute in the model, then all dependent pins correlated to the old representative pin in the model will be correlated to the representative pin specified in the CORRELATE mode definition statement.
- Conflicts between attribute-defined correlation polarity and mode definition polarity are won by the mode definition, with associated informational messages issued.
- If the dependent pin in the CORRELATE statement is defined as a dependent pin (to a different representative pin) through the CORRELATE attribute in the model, its correlation is changed to comply with the mode definition and an informational message is issued.
- If a representative pin in the CORRELATE statement is defined as a dependent pin through the CORRELATE attribute in the model, an error message is issued and the statement is ignored.

  See Specifying Differential I/O and Other Correlated Pins in *Encounter Test: Guide 1: Models* for details on the CORRELATE attribute.

DIAGNOSTIC_MODE

The `DIAGNOSTIC_MODE` statement identifies the test mode in which diagnostic scan-out is to occur. This statement is used only when defining an On-Product MISR test mode.



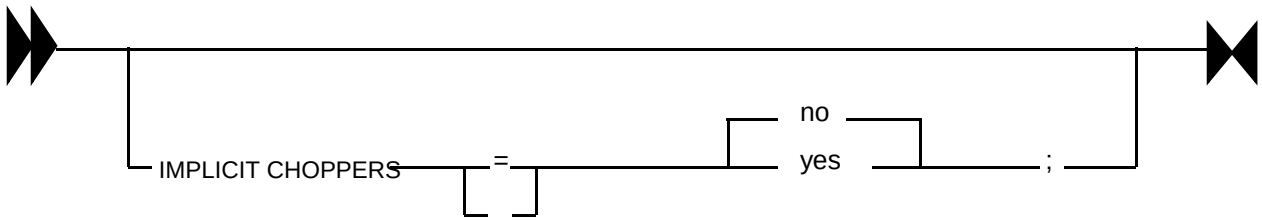
In this statement:

mode_name Specify a chip fullscan chain test mode for use in diagnostics.

IMPLICIT CHOPPERS

The IMPLICIT CHOPPERS statement provides a method to avoid unintentionally defining a chopper in the test mode. An implicit chopper is an otherwise validly defined chopper design that does not contain the normally requisite CHOPL or CHOPT block.

Refer to Clock Chopper Primitives and Implicit Clock Choppers in *Encounter Test: Guide 1: Models* for additional information.



The default is `no`, to disallow implicit choppers.

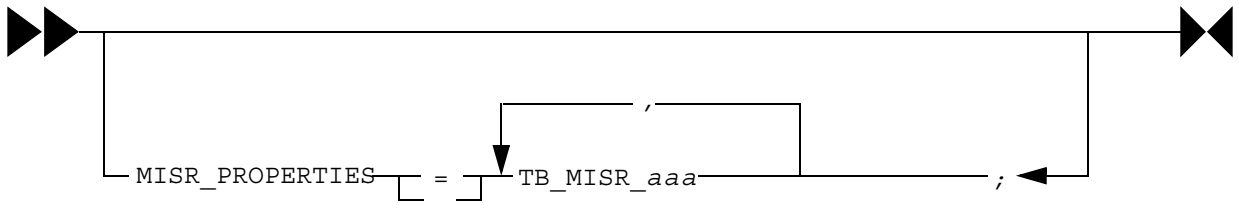
The Implicit Chopper keyword can appear anywhere in the Mode Definition file and only has an effect on the treatment of valid implicit clock choppers. Invalid clock choppers are always treated as errors and require Pessimistic (X-source) simulation. See "InfiniteX Simulation" in the *Automatic Test Pattern Test Generation User Guide* for related information.

MISR_PROPERTIES

Use the optional `MISR_PROPERTIES` statement to specify model attribute names to be used to identify the nets in the product that are to be treated as MISR output nets.

Important

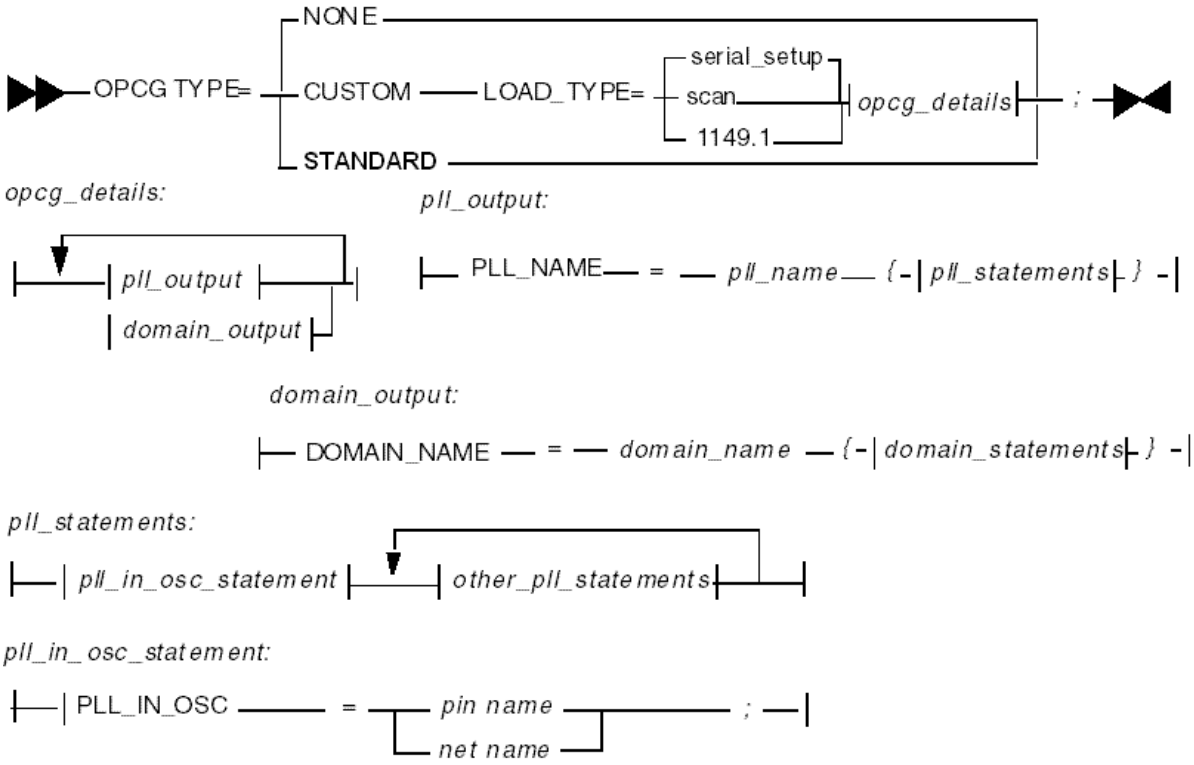
It is strongly recommended to begin the name of specified MISR properties with the prefix `TB_MISR` to avoid conflict with other attribute names.



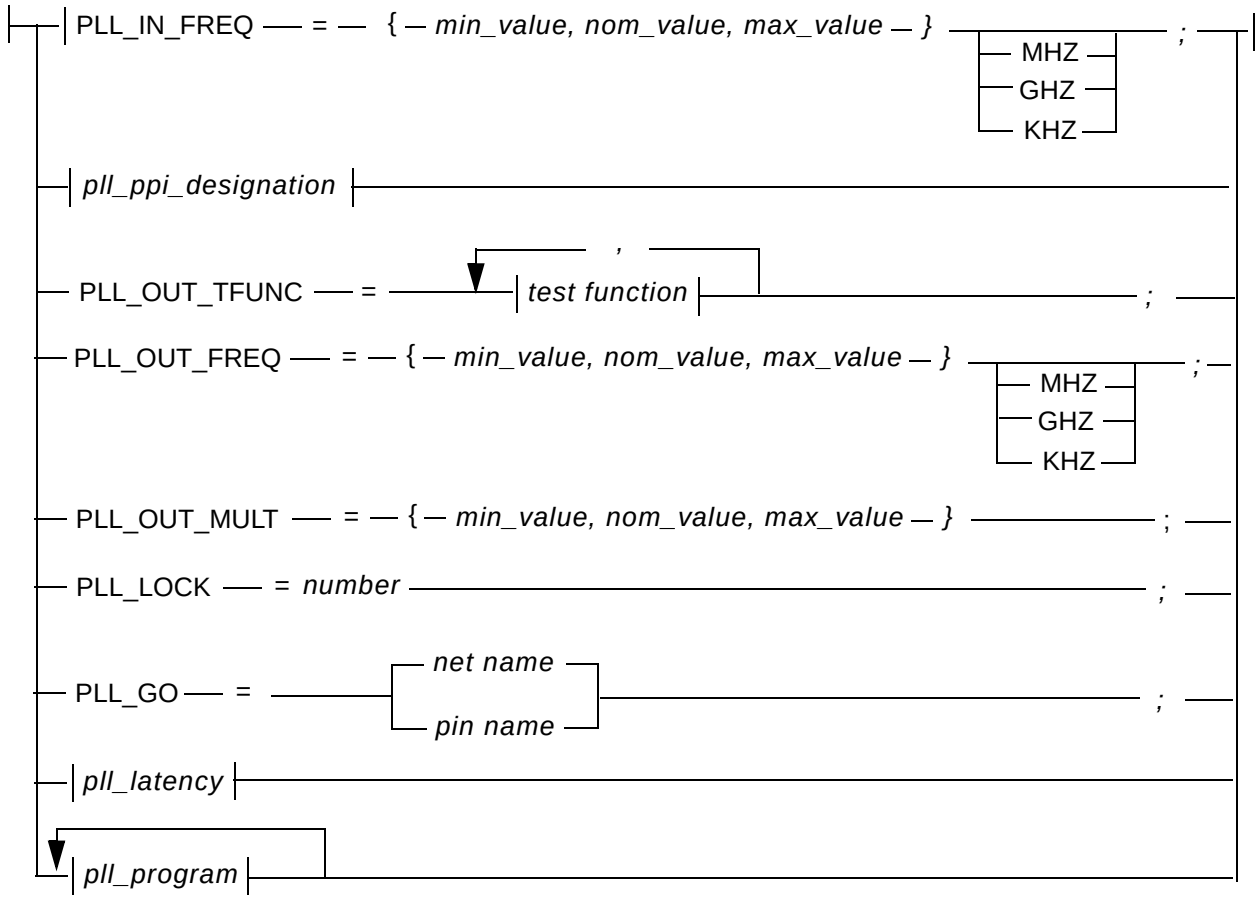
OPCG

The OPCG statement defines OPCG logic to automatically process cut points and collect additional data. Multiple OPCG statements are accepted as long as they do not conflict with

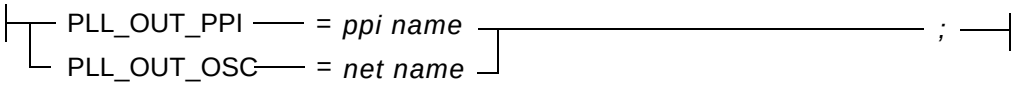
each other. Multiple OPCG statements are composited and applied to the mode definition. Refer to OPCG Test Modes for more information.



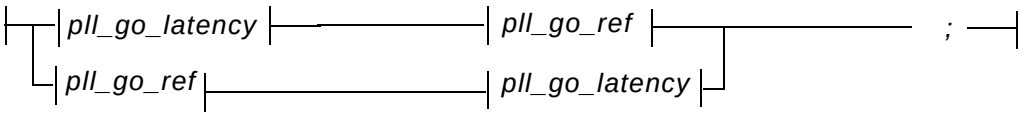
other_pll_statements:



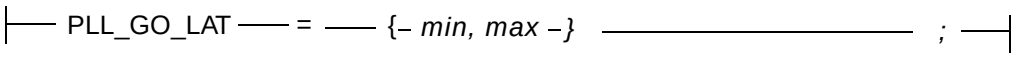
pll_ppi_designation



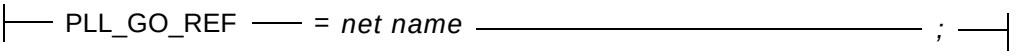
pll_latency:



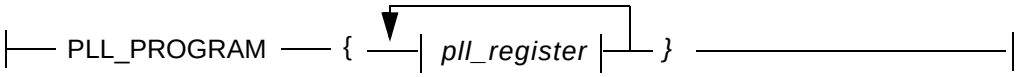
pll_go_latency:



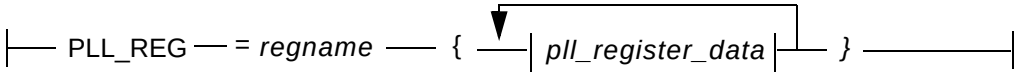
pll_go_ref:



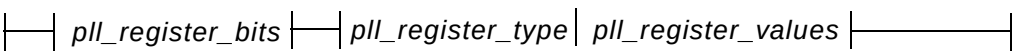
pll_program:



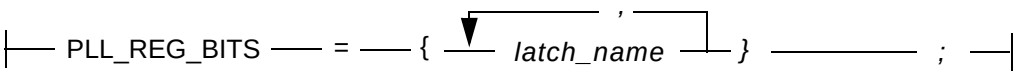
pll_register:



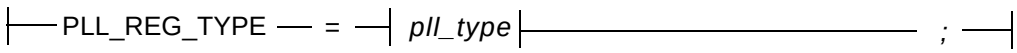
pll_register_data:



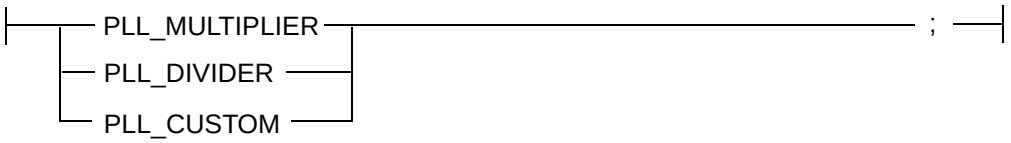
pll_register_bits:



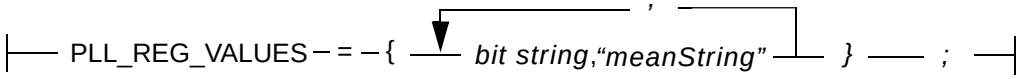
pll_register_type:



pll_type:

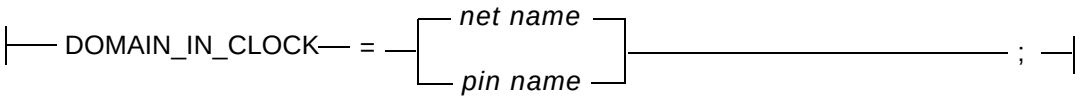


pll_register_values:

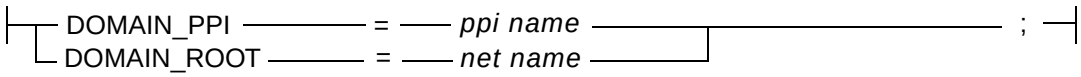


domain_statements:

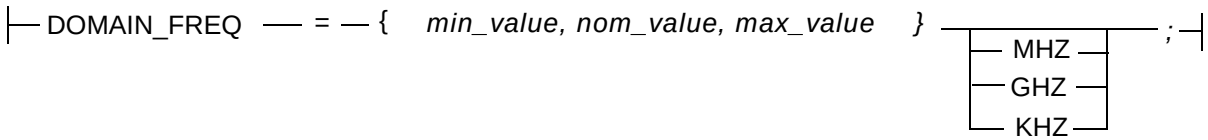
domain_in_clock_statement:



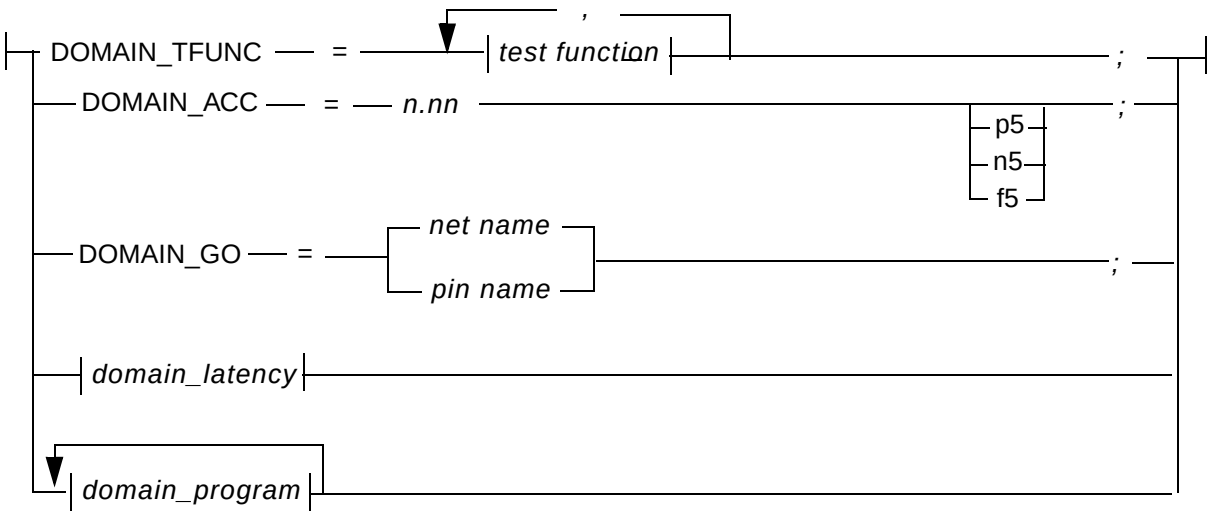
domain_ppi_def:



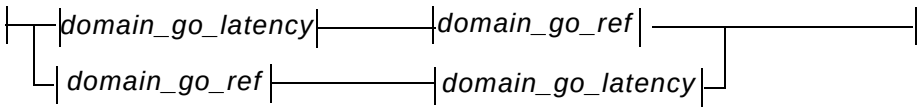
domain_freq_statement:



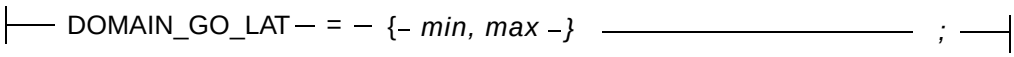
other_domain_statements:



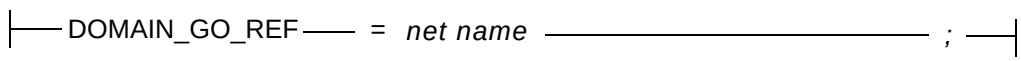
domain_latency:



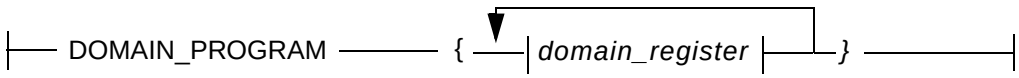
domain_go_latency:



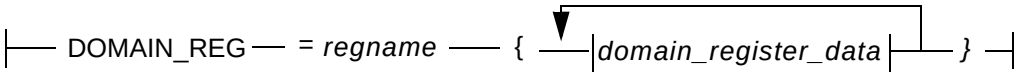
domain_go_ref:



domain_program:

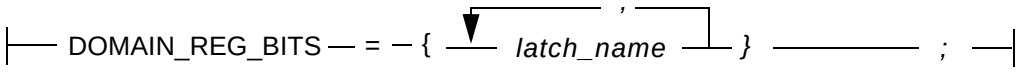


domain_register:

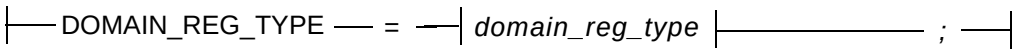


domain_register_data:

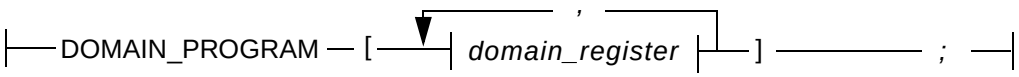
domain_register_bits:



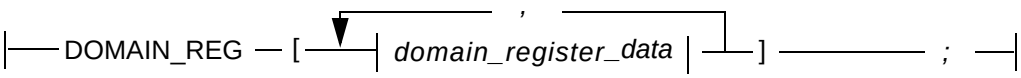
domain_register_type:



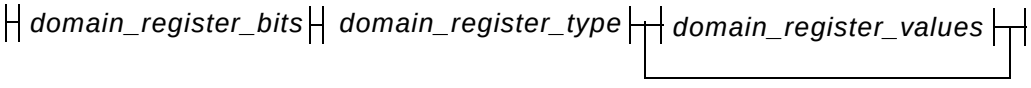
domain_program:



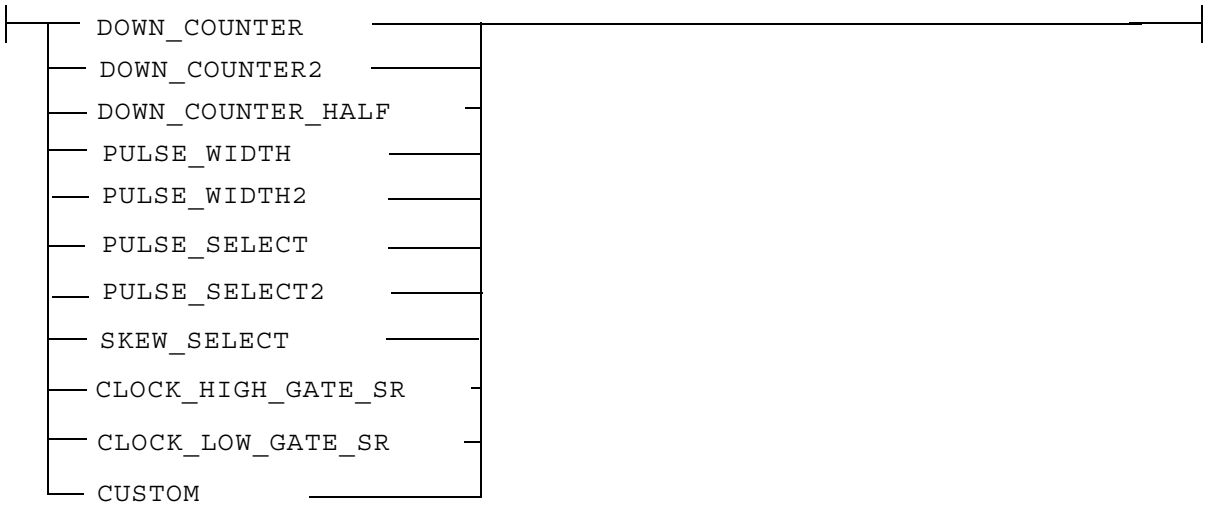
domain_register:



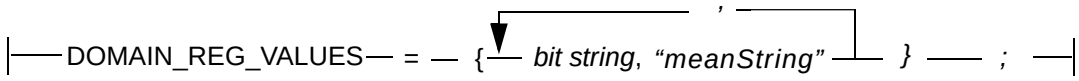
domain_register_data:



domain_reg_type:



domain_register_values:



In this statement:

TYPE

Specify one of the following options to indicate the type of OPCG logic:

- NONE to indicate no OPCG logic is defined
- CUSTOM to indicate that customized OPCG logic is implemented. Custom requires additional input, described for opcg_details on page 249.
- STANDARD to indicate that the OPCG logic is inserted by Cadence RTL Compiler

LOAD_TYPE Specify one of the following methods to load OPCG programming registers:

- `serial_setup` to load registers via a serial scan that is applied during a setup sequence.
- `scan` to indicate that the registers are built into the standard scan chains and will require reload on every `scan_load` event. Refer to "Scan_Load" in the *Test Pattern Data Reference* for additional information.
- `1149.1` to indicate use of an 1149.1 instruction to load the registers. Refer to "Automatically Generated Initialization Sequence" on page 67 for additional information.

  If using an 1149.1 instruction, specify one of the following options to define the type of `INSTRUCTION`:

  - A binary string to specify the binary bit string to be loaded.
  - The name of the 1149.1 instruction.

opcg_details Specify additional OPCG specifics to identify elements such as PLL and clock domain information. Refer to the following:

- "Specifying PLL Data"
- "Specifying Clock Domain Data" on page 252

Specifying PLL Data

Specify PLL details using the syntax `PLL_NAME=`*`pll_name`*. The assigned name to each PLL output must be unique. As a netlist attribute, specify a comma-separated list of names If a specified netlist name begins with an ampersand (&), the name will have the containing cell instance name prepended with an underscore (_).

The following are the `PLL_NAME` keyword options and syntax:

Note: The following keywords are optional unless specifically designated as required.

`PLL_IN_OSC=`*`Input_Oscillator_pin`* `;`

Specify the name of the PLL oscillator input. As a keyword option, the value must be a primary input pin or a net that can be traced back to a PI or another PLL output (`PLL_OUT` for a different PLL definition).

As an attribute, it must specify the input pin of the cell containing the attribute or, if it is an instance attribute, the name of a net connected to the cell instance.

In either case the pin or net must be traceable to locate the PI that will be supplying the oscillator input. When a single physical PLL has multiple outputs, they may all reference the same `IN_OSC`.

`PLL_IN_OSC` is a required `PLL_NAME` keyword.

`PLL_IN_FREQuency=(`*`min,nom,max`*`)[MHz|GHz|KHZ];`

Specify the minimum, nominal and maximum input oscillator frequencies supported by the PLL..

`{PLL_OUT_PPI=`*`Output_Oscillator_PPI`*`}`

Specify a Pseudo-PI (PPI) defined elsewhere that is used to represent the PLL output as a free running clock. If this is specified, OUT_OSC should not also be specified. This keyword is mutually exclusive with `PLL_OUT_OSC`.

`{PLL_OUT_OSC=`*`Output_Oscillator_net`*`} ;`

Specify the name of the net for which a cutpoint and PPI will be automatically defined. The PPI name will be generated from a catenation of the `PPI_NAME` value with `_PLL_PPI` at the end. Note that if the `PLL_NAME` attribute value begins with an ampersand (&), the PPI name will have the instance name for the cell containing the attribute included. This keyword is mutually exclusive with `PLL_OUT_PPI`.

`PLL_OUT_TFUNC =` *`testfunction`* `;`

Specify the test function to be associated with the PPI corresponding to the PLL output. This is the only means of specifying the intended test function for the PPI when generating the PPI given the `OUT_OSC` net. If the PPI is already defined, either this keyword or the ASSIGN statement may be used to specify the PPI test function.

`PLL_OUT_FREQuency = (`*min*`,` *nom*`,` *max*`) [MHz | GHz | KHz] ;`

Specify the minimum, nominal and maximum output frequency supported by the PLL output. The default unit is MHz.

`PLL_OUT_MULT = (`*min*`,` *nom*`,` *max*`) ;`

Specify the minimum, nominal and maximum multiplying factors for the PLL. When a single physical PLL has multiple outputs, only the highest frequency output requires `PLL_OUT_MULT` specified for it.

`PLL_LOCK =` *n* `;`

Specify the maximum number of input oscillator cycles before the PLL is guaranteed to be locked. Specify this attribute only on the PLL output used to do the locking of the PLL (this output is fed back to compare input).

`PLL_GO=` *GO_pin* `;`

Specify the `GO` signal used to control all domains run by the PLL output.

`PLL_GO_LATency = (`*min*`,` *max*`) ; PLL_GO_REF =` *pll_out_name* `;`

Specify the relative minimum and maximum latency between when the `GO` primary input is triggered and the first possible clock edge appears at any clock domain controlled by the PLL output. Specify the minimum and maximum as cycle counts for the `PLL_GO_REF` oscillator signal.

`PLL_GO_REF` is required if `PLL_GO_LATency` is specified. Specify the reference oscillator used to estimate the latency between `GO` switching and clock edges appearing on domain clock tree roots.

```
[PLL_PROGRAM {
  PLL_REG = regname {
       PLL_REG_BITS=(bit_n-1, bit_n-2, ..., bit_0) ;
   PLL_REG_TYPE = plltype ;
    PLL_REG_VALUES = ( val_0, "meaning_0" ,
                ...., val_n, "meaning_n" ) ;
```

Specify the beginning of a block of PLL register definitions in the mode definition or ASSIGN file.

`PLL_REG` - specify the name for the PLL program register. Use a comma separated list of register names to specify as a netlist attribute.

Within the mode definition or ASSIGN files, the following keywords apply to this register up until the next `REG` keyword specifies the start of information for a new register:

- `PLL_REG_BITS` - specify a list of latches or flops that comprise the PLL register, listed from the high-order bit to the low-order bit. The bit ordering is important since values will be assigned to the registers as binary values.
- `PLL_REG_TYPE` - specify the type of the PLL register. The valid PLL register types are: `PLL_multiplier`, `PLL_divider`, and `PLL_custom`.
- `PLL_REG_VALUES` - specifies a list of paired values and meanings. The first value is a binary value representing a value that can be loaded into the bits of the register. Specify binary values (for example, `00101` for a 5-bit register). The second value is a string providing some meaning to the preceding value (for example, `"multiply_by_5"`). Separate the values with commas. Each pair constitutes data for a single value. To specify register values via a netlist attribute, specify the entire list of values and meanings as a single string with commas separating the values from the meanings

Specifying Clock Domain Data

Specify domain details using the syntax `Domain_NAME=`*`domain_name.`* `{`. The assigned name to each domain must be unique. As a netlist attribute, specify a comma-separated list of names If a specified netlist name begins with an ampersand (&), the name associated with this instance of the domain will include a pre-pending of the hierarchical name of the instance of the cell containing the attribute.

The following are the DOMAIN_NAME keyword options and syntax:

Note: The keywords are optional unless specifically designated as required.

DOMAIN_IN_CLOCK = *in_clock* ;

Specify the input oscillator clock used to run the domain's OPCG logic. When specified as a netlist attribute, it must specify a cell pin name or a net name if it is on a cell usage/ instance. Tracing back through ungated logic from this pin or net arrives at either an OSC node (PI or PPI output of a PLL) or the ROOT (PPI) of another clock domain when there is a cascading of domains.

DOMAIN_PPI = *PPI_name* ;

Specify the PPI representing the clock domain. when the PPI is already defined for the test mode. This keyword is mutually exclusive with DOMAIN_ROOT. Use DOMAIN_PPI when there are multiple cutpoints as DOMAIN_ROOT identifies a single net that will be the cutpoint for the PPI.

DOMAIN_ROOT = *domain_root* ;

Specify the name of the net for which a cutpoint and PPI will automatically defined to represent the clock domain. This keyword is mutually exclusive with DOMAIN_PPI.

DOMAIN_FREQ= (*min*, *nom*, *max*) [MHz | GHz | KHz] ;

Specify the frequency range and nominal value. The default unit is MHz.

`DOMAIN_TFUNC = testfunction ;`

Specify the test function to be associated with the PPI corresponding to the Domain Root. This is the only means of specifying the intended test function for the PPI when generating the PPI given the ROOT net. If the PPI is already defined, either this keyword or the ASSIGN statement may be used. Refer to TEST_FUNCTION - Indicates that the test function will follow. The use of this keyword is only for readability. on page 222 for description of the pins.

`DOMAIN_ACCuracy = n.nn [ps | fs | ns] ;`

Specify the relative skew between cutpoints associated with the PPI of the clock domain. This is analogous to the TDR specification of pin to pin accuracy for the tester.

`DOMAIN_GO= GO_pin ;`

Specifies the GO signal used to control all domains run by the domain output.

`DOMAIN_GO_LATency = (min, max) ; DOMAIN_GO_REF = domain_out_name ;`

Specify the relative minimum and maximum latency between when the GO primary input is triggered and the first possible clock edge appears at any clock domain controlled by the domain output. Specify the minimum and maximum as cycle counts for the DOMAIN_GO_REF oscillator signal.

DOMAIN_GO_REF is required if DOMAIN_GO_LATency is specified. Specify the reference oscillator used to estimate the latency between GO switching and clock edges appearing on domain clock tree roots.

```
[DOMAIN_PROGRAM {
  DOMAIN_REG = regname {
       DOMAIN_REG_BITS=(bit_n-1, bit_n-2, ..., bit_0) ;
   DOMAIN_REG_TYPE = plltype ;
    [DOMAIN_REG_VALUES = ( val_0, "meaning_0" ,
                ....., val_n, "meaning_n" ) ; ]
```

Specify the beginning of a block of OPCG program register definitions in the mode definition or ASSIGN file.

DOMAIN_REG - specify the name for the OPCG program register. Use a comma separated list of register names to specify as a netlist attribute.

Within the mode definition or ASSIGN files, the following keywords apply to this register up until the next REG keyword specifies the start of information for a new register:

DOMAIN_REG_BITS - specify a list of latches or flops that comprise the OPCG register, listed from the high-order bit to the low-order bit. The bit ordering is important since values will be assigned to the registers as binary values.

DOMAIN_REG_TYPE - specify the type of the OPCG register. The following are valid register types:

- DOWN_COUNTER counts down from the active edge of the GO signal for the launching of pulses down the associated clock domain.
- DOWN_COUNTER2 is a counter whose clock runs at double-speed (the counter's clock is twice the frequency of the domain input clock).
- DOWN_COUNTER_HALF is a counter whose clock runs at half the domain's input clock frequency.
- PULSE_WIDTH is an n bit binary register that is 1-hot such that exactly 1 of its bits must be 1 if the domain is to be used.
- PULSE_WIDTH2 is used when the pulse generation logic runs at twice the domain's input clock frequency. the earliest possible trailing edge is the cycle after the leading edge of the domain's pulse.
- PULSE_SELECT statically defines when additional pulses beyond the first for a domain should appear.
- PULSE_SELECT2 denotes that the domain OPCG logic is running at twice the frequency of the domain input clock.

- SKEW_SELECT is an n bit register whose binary value selects from a set of available delays to add into the clock domain when in OPCG mode. The skew amount added will delay the start of all clock edges. If there are multiple OPCG macros in various locations on a device that use the same clock frequency, adding some skew to one may be useful for dealing with timing between such regions of the same clock domain in OPCG mode.

- CLOCK_HIGH_GATE_SR contains the string of bits to be used to gate the domain's input clock when that clock is in its high phase. This is a serially shifted register that outputs a new value on the falling edge of the input clock so that it is safe to use the register output to gate the clock high phase. This register can be paired with a CLOCK_LOW_GATE_SR register that gates the clock on the low phase. This is used when the domain clock output is generated by use of muxed clock gating on opposite phases of the clock.

- CLOCK_LOW_GATE_SR contains the string of bits to be used to gate the domain's input clock when that clock is in its low phase. This is a serially shifted register that outputs a new value on the rising edge of the input clock so that it is safe to use the register output to gate the clock low phase. This register can be paired with a CLOCK_HIGH_GATE_SR register that gates the clock on the high phase. This is used when the domain clock output is generated by use of muxed clock gating on opposite phases of the clock. This register is otherwise functionally equivalent to the CLOCK_HIGH_GATE_SR.

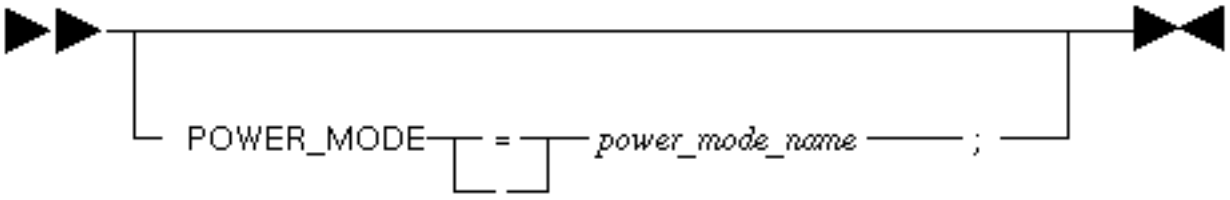
- BLOCK_DOMAIN_INPUTS contains a single bit that when set to a BLOCK value will cause the functional flops of this domain that receive values from other domains to have the paths into them from other domains blocked. Valid value meanings are BLOCK and ENABLE.

- CUSTOM is used to define programming bits that are useful to an OPCG design however are not recognized by Encounter Test. These may be of arbitrary length and Encounter Test will not have any understanding of what values to set into these registers other than to apply the values specified to load into them.

`DOMAIN_REG_VALUES` specifies a list of paired values and meanings. The first value is a binary value representing a value that can be loaded into the bits of the register. Specify binary values (for example, `00101` for a 5-bit register). The second value is a string providing some meaning to the preceding value (for example, `"multiply_by_5"`). Separate the values with commas. Each pair constitutes data for a single value. To specify register values via a netlist attribute, specify the entire list of values and meanings as a single string with commas separating the values from the meanings.

Power_Mode

The following optional mode definition syntax is used to configure a reduced power test mode. Refer to Configuring a Reduced Power Test Mode Definition Statement in *Encounter Test Low Power User Guide*.



Specify the following:

`Power_Mode=`*`powermodename`*

The specified name must match the name of a Power Mode specified with a `create_power_mode` statement in the CPF file specified for the `build_testmode` command (`cpffile=`*`filename`*).

C

Tester Description Rule (TDR) File Syntax

Introduction to Tester Description Rules (TDRs)

In order to ensure that test data generated by Encounter Test will adhere to the constraints of the target tester on which the design is to be tested, Encounter Test takes as input a description of the tester's capabilities. This is called a tester description rule (TDR).

The TDR is a text file containing a simple and easily parsed set of statements that are used to convey the tester attributes. A TDR is usually provided by the company that will be testing the design, which is typically the chip, module or card manufacturer.

The TDR information identifying the owner of the TDR (the person or company who wrote the TDR statements) is passed along with the test data produced by Encounter Test to provide a contact for the receiving manufacturer in case of any concerns about the correctness of the TDR.

The test data produced by Encounter Test points to one and only one TDR. Therefore, if the same test data is to be shared by two or more testers, the person creating the test mode definition must ensure that a TDR exists that accurately describes the characteristics and limitations the test data must comply with to be usable by all the testers. Refer to “Mode Definition Statements” on page 196 for more information on test mode definition statements.

TDR Statement Syntax

The following sections describe the various statements that comprise a TDR. The `TDR_DEFINITION` statement must be the first statement in the TDR, Other statements can appear in any order. A sample TDR that you can use as a template is given in “Sample TDRs” on page 286.

Each statement of the TDR must end with a semicolon.

The comment characters allowed in TDR statements are:

line comments:

- `//`
- `--`
- `#`

block comments:

- `/*block of comments*/` - "/*" to start and "*/" to end block comments.

  The block comments may appear any place white space may appear. The white space is either a blank or a new line character.

Refer to the following for related information:

- "TESTER_DESCRIPTION_RULE" on page 197
- "Resolving Internal and External Logic Values due to Termination" in the *Encounter Test: Guide 5: Test Vectors*

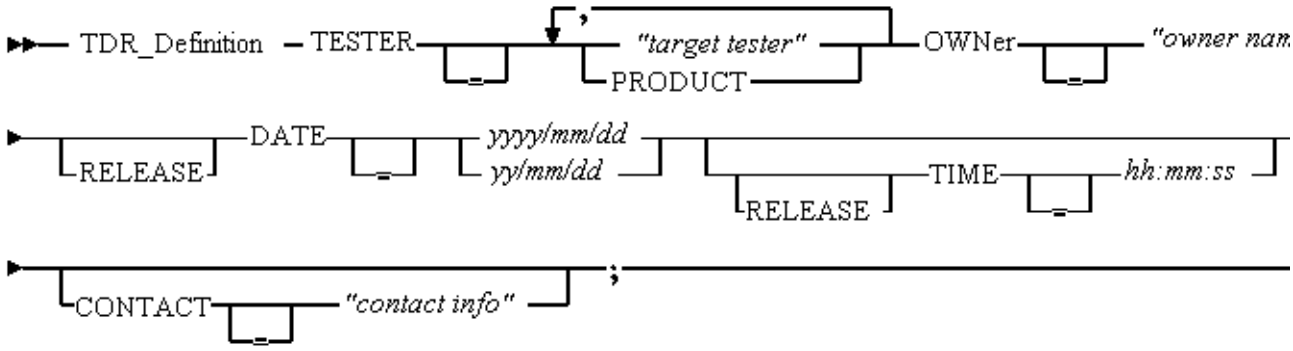
TDR_DEFINITION

The `TDR_DEFINITION` statement is used to identify who wrote the TDR and to which tester(s) the TDR applies. This statement must be the first statement in the file. Up to seven target testers can be specified. The character strings owner, `target_tester`, and `contact_info` must each be less than 32 characters long or they will be truncated to the first 31 characters.

If a TDR is defined that applies to multiple testers, its parameters must be specified such that they are compatible with every tester in the set. It is up to the coder of the TDR to ensure this. One case where this is useful is when the manufacturer wants to balance workload among several different testers capable of testing the design.

The syntax for the TDR DEFINITION statement is shown in the following figure.

Figure C-1 TDR_DEFINITION Statement Syntax



- TESTER

  Required. It is used to specify one or more manufacturing tester names for which test data generated under the constraints specified in the TDR can be applied. One of the tester names can be the reserved name PRODUCT, which implies that test data generated for this TDR can be applied by the product (and possibly also by actual testers). For example, chip or module self test could be applied under assumptions of both PRODUCT and tester constraints.

  - target_tester

    A text string that identifies the tester this TDR applies to.

  - PRODUCT

    Indicates that the product itself can perform any tests which adhere to the limitations of this TDR. This keyword is intended to be used for built-in self test (BIST), where the design tests itself and there is no need for the tester.

- OWNER

  Required. Identifies the name or some other suitable identification of the owner.

- (RELEASE) DATE

  Required. Specify the date this TDR was last modified.

- (RELEASE) TIME

  Optional. Specify the time this TDR was last modified.

- CONTACT

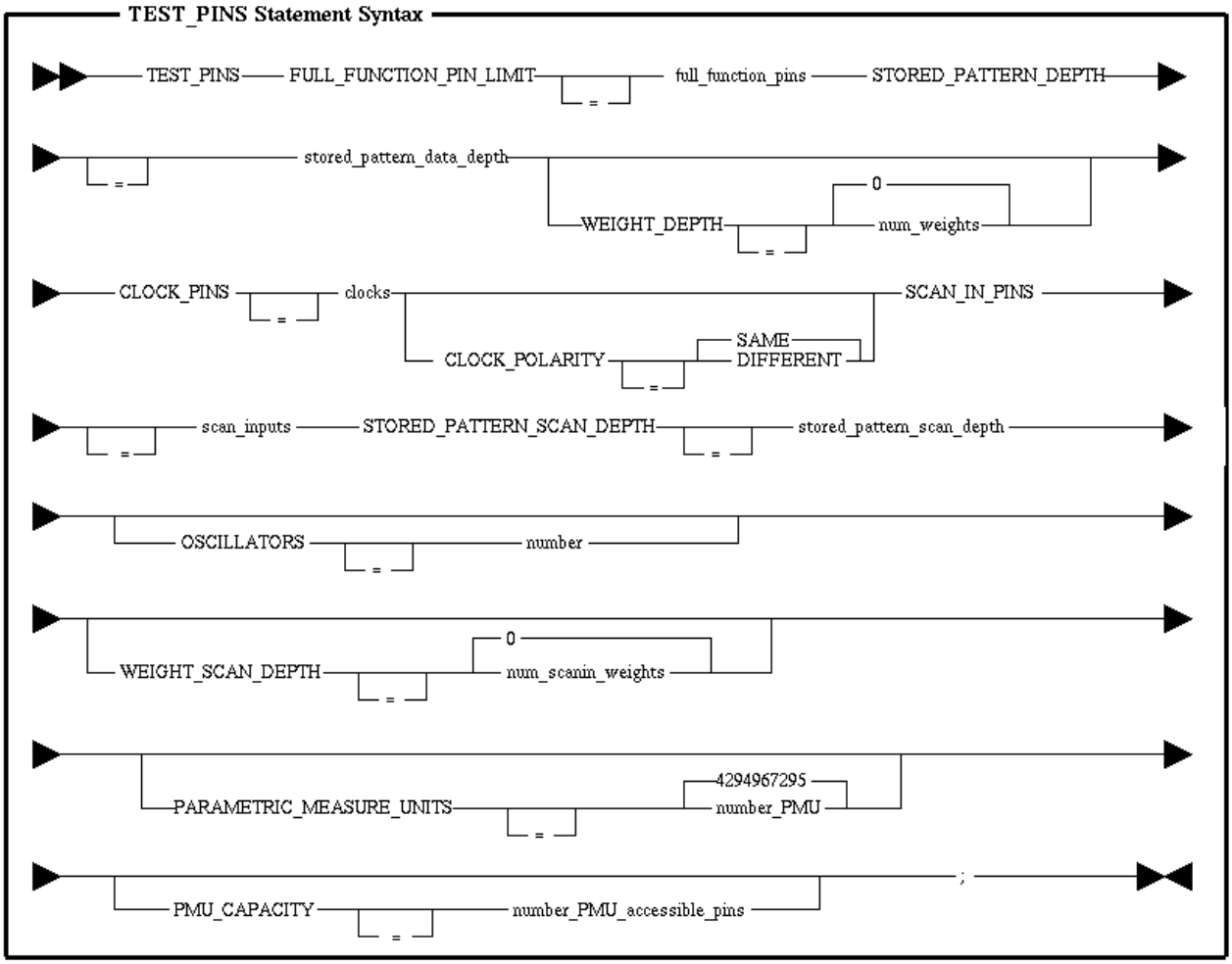
  Optional. Identifies how to contact the owner; for example, address or phone.

TEST_PINS

The TEST_PINS statement is used to specify how many pins are available on the tester for performing specific functions. It is a required statement.

The syntax for the TEST_PINS statement is shown in the following figure.

Figure C-2 TEST_PINS Statement Syntax



- FULL_FUNCTION_PIN_LIMIT

  Required. Specifies how many full function (can be data input, data output, scan input, scan output or a clock) pins the tester supports. This value represents the maximum number of product pins that can be connected to the tester to have either stimulus or measurement operations applied. For example, a 64-pin tester that supports clocks, scan data inputs, scan data outputs or normal stim and response functions on each pin

would specify `FULL_FUNCTION_PIN_LIMIT=64` to indicate that any number of clocks, scan and data assignments can be made as long as the total number of pins contacted is 64 or less. The total number of pins will not exceed 64 due to the assumption that there is overlap.

- STORED_PATTERN_DEPTH

  Required. Specifies how many stimulus or response measurements can be accommodated on any data pin in a single tester buffer load. This count can be used to determine if the test data will fit in a single tester buffer load.

- WEIGHT_DEPTH

  The number of different weight sets that can be stored in the tester. This has meaning only if you are using weighted random pattern testing. The default is zero.

- CLOCK_PINS

  Required. Specifies how many tester pins can deliver pulses to the product.

- CLOCK_POLARITY

  Specifies whether it is permissible for clock pins to change stability (off) polarity between test modes.

  - SAME

    Indicates that all clock pins must have the same stability polarity in all test modes. This is the default.

  - DIFFERENT

    Indicates that clock pins are allowed to have a different stability polarity in different test modes.

- OSCILLATORS

  Specifies the number of free-running oscillators that are supported by the tester.

- SCAN_IN_PINS

  Required. Specifies how many pins can be scan input pins. It is assumed that there can be just as many scan output pins, provided that the *FULL_FUNCTION_PIN_LIMIT* count is not exceeded. Since scan in/out pins usually require substantially more buffer memory than data pins, some testers may place limits on the number of scan strings any product may define. If there is no special consideration for scan I/O pins, simply use the same value specified for *FULL_FUNCTION_PIN_LIMIT*.

- STORED_PATTERN_SCAN_DEPTH

Required. Specifies how many stored pattern scan in or scan out values a single scan pin can support within a single tester buffer load. This count can be used to determine if the test data will fit in a single tester buffer load.

- WEIGHT_SCAN_DEPTH

  This relates to the size of the tester's memory that is usable for storing weights on a scan data input pin. This number, divided by the length of the longest scan chain on the product, gives an upper bound on the number of weight sets that can be stored on the tester. *num_scanin_weights* is another bound on the number of weight sets. These parameters are used only if you are doing weighted random pattern testing. The default is zero.

- PARAMETRIC_MEASURE_UNITS:

  The number of parametric measurement units (PMUs) to be assumed by Encounter Test processing. This parameter is significant only if (a) driver & receiver testing is being performed, or (b) the number of full-function pins is less than the number of active pins on the product and static logic testing is being performed. In the latter case, most testing should be done in a boundary scan (internal) test mode in which only the test function pins need to be contacted. The PARAMETRIC_MEASURE_UNITS parameter limits the number of data pins that can be stimulated or measured within the same test pattern. The default is 4294967295 (2 to the 32nd power minus 1). It should be noted that Encounter Test always assumes that a full-function pin is capable of making parametric measurements, so PARAMETRIC_MEASURE_UNITS should always be equal to or greater than FULL_FUNCTION_PIN_LIMIT.

  Encounter Test has three types of support in relation to PMUs, which apply only to static logic testing and driver/receiver testing. In defining the conditions where these types of support apply, we assume that `PARAMETRIC_MEASURE_UNITS = FULL_FUNCTION_PIN_LIMIT`.

  - If `PARAMETRIC_MEASURE_UNITS >=` the number of product pins, then Encounter Test assumes all product pins can be contacted simultaneously. Otherwise:

    - If `PARAMETRIC_MEASURE_UNITS = FULL_FUNCTION_PIN_LIMIT`, then Encounter Test assumes that non-test function attributed pins cannot be contacted at all; *and*

    - If `PARAMETRIC_MEASURE_UNITS > FULL_FUNCTION_PIN_LIMIT`, then Encounter Test assumes all test function attributed pins can be contacted and only one non-test function attributed pin can be contacted at a time.

- PMU_CAPACITY

The total number of pins that can be contacted by PMUs (see PARAMETRIC_MEASURE_UNITS). These two parameters are different in the tester where some PMUs can be multiplexed to different pins. Note that PMU_CAPACITY is not the number of pins that can be simultaneously contacted by PMUs, but the total number that can be accessed by PMUs. PMU_CAPACITY should be equal to or larger than PARAMETRIC_MEASURE_UNITS. This parameter is optional; it defaults to the value of the PARAMETRIC_MEASURE_UNITS parameter.

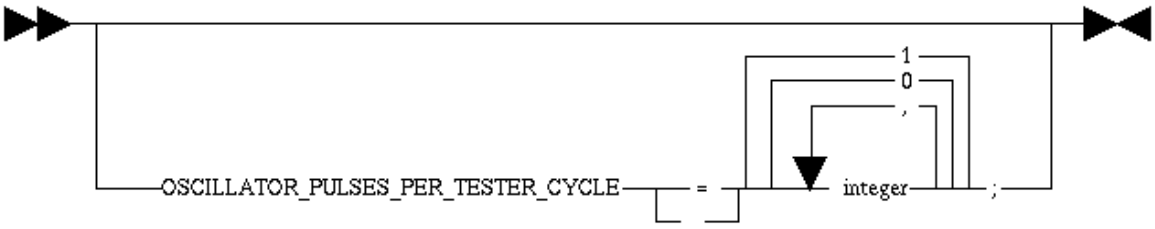
OSCILLATOR_PULSES_PER_TESTER_CYCLE

Some testers are capable of delivering multiple pulses on the same pin during a tester cycle. This capability is useful for driving oscillator inputs to the design under test. Encounter Test exploits this by the oscillator events in TBD (Start_Osc, Wait_Osc, Stop_Osc). This parameter tells how many oscillator pulses the tester is capable of producing on a single pin during each tester cycle.

If the tester (or test compiler) does not support the synchronous oscillators as described above, then code 0 for this parameter. If the tester can support several different numbers of oscillator pulses per tester cycle, enter the list of numbers supported.

The syntax for the OSCILLATOR_PULSES_PER_TESTER_CYCLE statement is shown in the following figure.

Figure C-3 OSCILLATOR_PULSES_PER_TESTER_CYCLE Statement Syntax



- **integer**

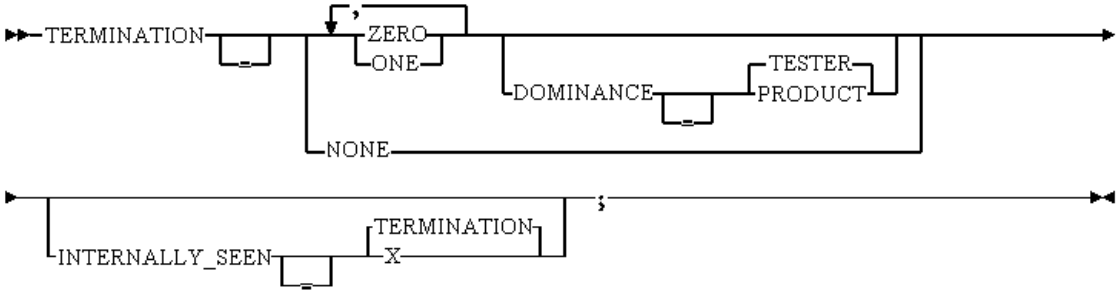
  Specifies the number of pulses the tester can deliver on an oscillator pin in each tester cycle. This must be a non-negative integer. Specify a list of one or more positive integers if the tester has the flexibility to supply a variable number of oscillator pulses per tester cycle. Specify 0 to indicate that the tester or test compiler does not support synchronous

oscillators. The integer list is required if this statement is used, but if the statement is missing the default is 1.

TERMINATION

The syntax for the TERMINATION statement is shown in the following figure. See section Termination Values in *Encounter Test: Guide 1: Models* for related information.

Figure C-4 TERMINATION Statement Syntax



- TERMINATION

  The TERMINATION statement is used to specify the tester's capabilities for terminating three-state (high- impedance) outputs and bidirectional pins. It is also used to specify what value should be propagated to internal nets from a bidirectional pin when the tester is not driving and all three-state product drivers for that pin are at high-impedance. It is a required statement.

  - ZERO

    Indicates that the tester can provide termination to zero.

  - ONE

    Indicates that the tester can provide termination to one.

    Note: If both ZERO and ONE are specified, then the tester can terminate such pins to either ZERO or ONE (but only one at a time).

  - NONE

    Indicates that the tester cannot provide termination to either zero or one.

- DOMINANCE

  Indicates whether tester supplied termination should be applied to pins which already have product termination, and if so, which will dominate. Dominance applies only when the tester can provide termination. It is optional.

  - TESTER

    Indicates that tester-supplied termination is applied to all three-state output pins regardless of any product-supplied value for a pin. This is the default if DOMINANCE is not specified.

  - PRODUCT

    Indicates that tester-supplied termination is applied only to pins without product-supplied termination.

- INTERNALLY_SEEN

  Specifies whether the resolved termination value for an external three-state net should be propagated into the internal (receiver) logic for bidirectional three-state pins, or that an unknown, pessimistic assumption should be made. The default is to use termination if there is any.

  - TERMINATION

    Specifies that internal logic should use the terminated value for a high impedance (Z) on an external three-state net. This is the default if INTERNALLY_SEEN is not specified.

  - **X**

    Specifies that even if an external three-state net has termination on it, a pessimistic assumption should be made that forces a high impedance (Z) to be seen as an unknown (X) value by the internal (receiver) logic.

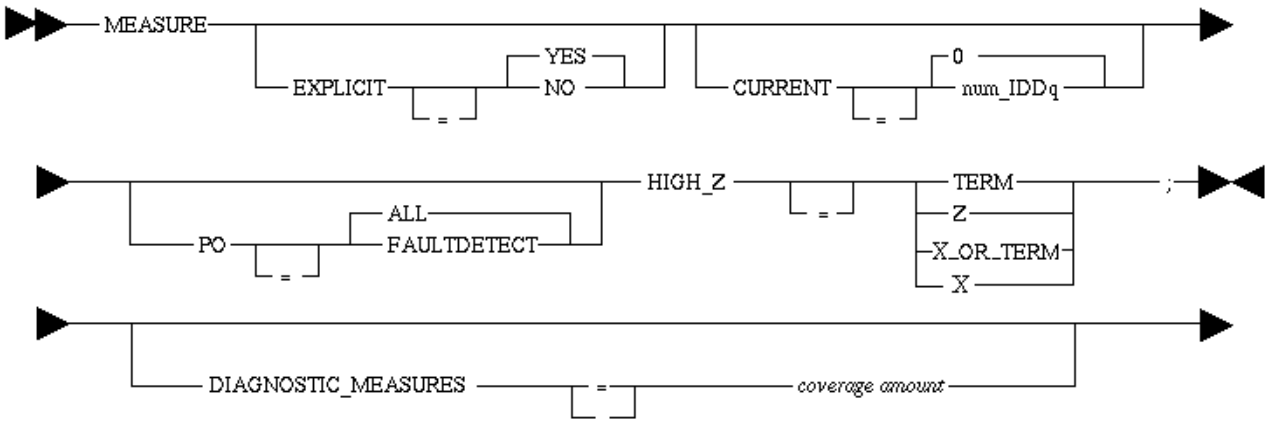
Note: Specifying `DOMINANCE=PRODUCT` will override this behavior. Refer to the description for globalterm in the *Encounter Test: Reference: Commands* for additional information.

MEASURE

The MEASURE statement is used to specify certain measure options for producing the test data. It is a required statement.

The syntax for the MEASURE statement is shown in the following figure.

Figure C-5 MEASURE Statement Syntax



- EXPLICIT

  Optional. This keyword governs the extent to which primary output measure data is written to the test data output of Encounter Test. Primary output measures are not taken at the tester except as requested by the test data. With `EXPLICIT=YES`, the default, measures are written to the test data output only for those patterns for which output measures are explicitly requested by either automatic test generation or by Measure_PO events in manually generated test vectors. With `EXPLICIT=NO`, primary output measures will be written for all patterns containing no scan activity, regardless of whether a primary output was specifically targeted for fault detection. Specifying NO can significantly increase test data volume over YES.

- CURRENT

  Specifies the maximum number of IDDq current measurement test vectors that should be provided. The default value of zero indicates that the tester does not support IDDq measurement of current.

- PO

  Optional.

  Specify which measures to produce on POs during fault simulation. Specify faultdetect to produce measures only on POs that detect faults. Specify all to produce measures on all POs regardless of fault detection.

  The default is all except for:

  - driver/receiver tests

- logic tests where the number of active logic pins is greater than the full function pin limit on the TDR

For the above two cases, the option is ignored and measures are produced only on POs that detect faults.

- HIGH_Z

  Required. This keyword is used to specify how high-impedance (Z) values are to be measured by the tester:

  - TERM

    Specifies that the product or tester termination (see the TERMINATION statement) should be used to convert high_Z values into logic zero or one values.

  - Z

    Specifies that regardless of product or tester termination, if all three-state drivers feeding a product I/O pin are inhibited, the measure response should be Z (high impedance).

  - X_OR_TERM

    Specifies that the product or tester termination (see the TERMINATION statement) should be used to convert high_Z values into logic zero or one values if termination exists on the PO pad. Otherwise, X (unknown) should be measured. In case of no termination, the value of Z will be measured as X (unknown).

  - X

    Specifies that regardless of product or tester termination, if all three-state drivers feeding a product I/O pin are inhibited, the measure response should be X (unknown).

- DIAGNOSTIC_MEASURES

  Specify a fault coverage percentage threshold that, when attained, test generation will discontinue keeping diagnostic Scan_Unload events. The default is 100.

  The default implies the test data will carry precomputed diagnostic expects.

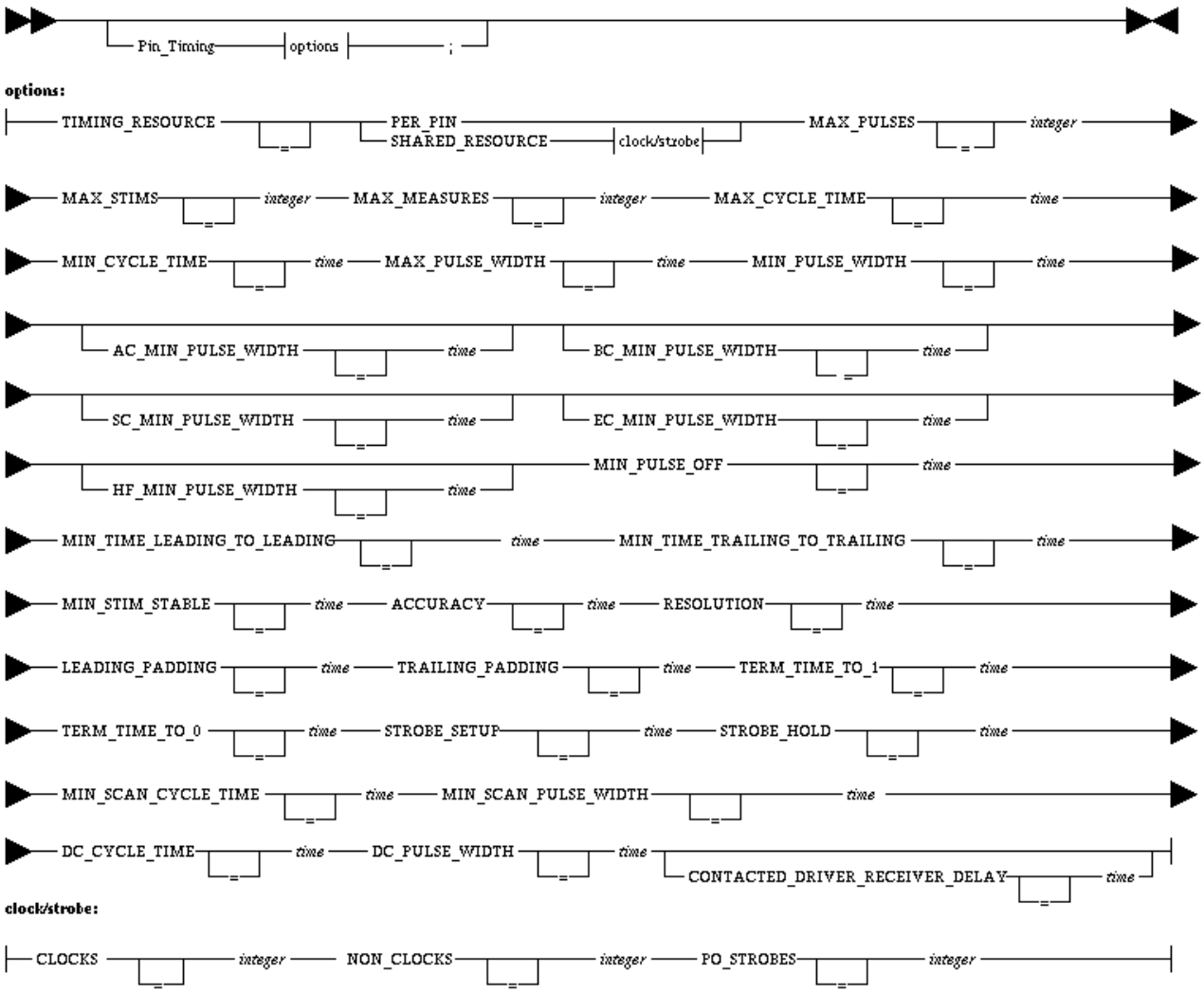
PIN_TIMING

The PIN_TIMING statement is used to specify information about the timing capabilities, both hardware and software, of the manufacturer's testers. All times are specified by a decimal followed by ps, ns, us, ms or s. The smallest resolution is a picosecond (ps). For example, 2.5 ps resolves to 3 ps.

If the PIN_TIMING statement is not included in the TDR, timings can still be generated. However, all the time and number fields are considered to be zero. Therefore, the PIN_TIMING parameters that should be active must be specified at runtime.

The syntax for the PIN_TIMING statement is shown in the following figure.

Figure C-6 PIN_TIMING Statement Syntax



- TIMING_RESOURCE

  Specify either PER_PIN or SHARED_RESOURCE.

  - PER_PIN

Specify this option to indicate that the tester has timing capabilities that allow each of its pins to be timed independently.

- SHARED_RESOURCE.

  Specify this option to indicate that the tester does not have timing capabilities on every pin, so individual timing generators must be shared across pins.

The timing generators are classified by Encounter Test according to the type of product pins they contact. The types are clocks (pins attached to pulse generators), non-clocks, and Primary Outputs (POs). Within each group there is a maximum number of generators allowed.

For example, if there are four product clocks and two timing generators, Encounter Test may relate two clocks to one pulse generator and two to the other or three to one pulse generator and one to the other.

Note: When specifying the shared-resource pin counts, you should take the following into consideration:

- The tester capabilities
- The capabilities of the software that generates the tester program from Encounter Test output

For example, the software-hardware interaction may require that you reserve a certain number of pulse generators for scanning. These generators will not be included in the clock count.

- CLOCKS

  Required.

  Specify the number of independently timed clock groups allowed to be pulsed during the tester cycle.

- NON_CLOCKS

  Required.

  Specify the number of independently timed groups of non-pulsed pins allowed in the tester cycle. This includes non-clock pins, as well as clock pins that are not pulsed.

- PO_STROBES

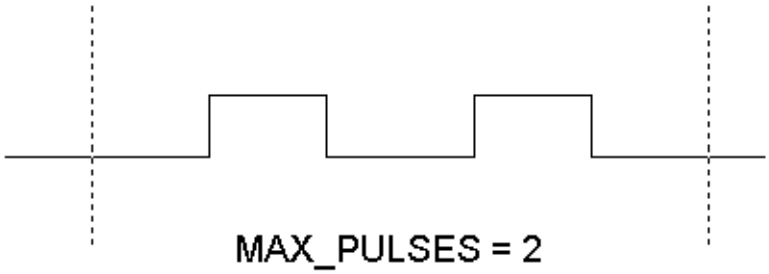
  Required.

  Specify the number of independently timed groups of measures allowed on the output pins of the product.

- MAX_PULSES

  Required.

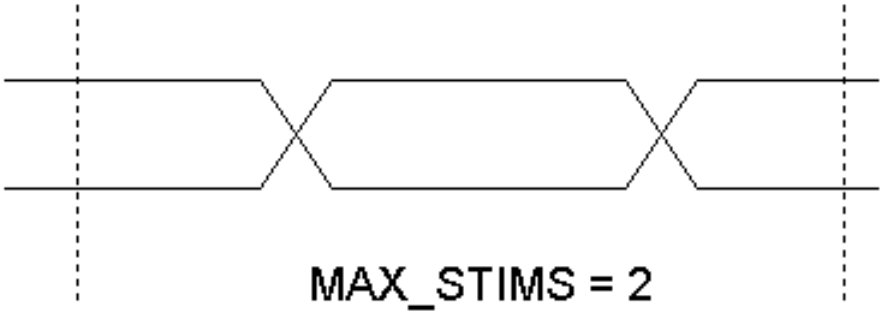
  Specify the maximum number of pulses (pair of edge changes on a product input pin) on a single pin during the tester cycle. Refer to the following example:



- MAX_STIMS

  Required.

  Specify the maximum number of stims (single edge change on a product input pin) on a single pin during the tester cycle. Refer to the following example:



- MAX_MEASURES

Required.

Specify the maximum number of measures on a product output pin during the tester cycle.

- MAX_CYCLE_TIME

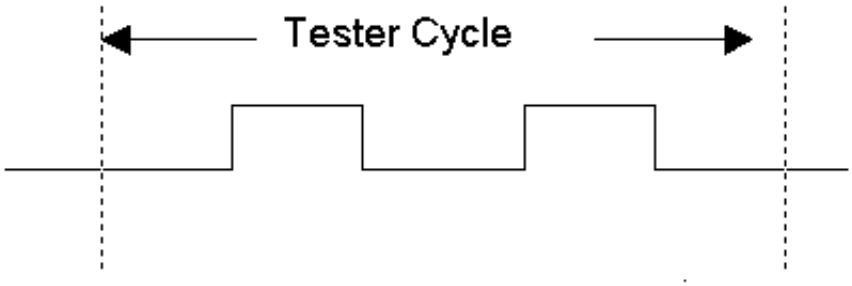
  Required.

  Specify the maximum tester cycle time.

- MIN_CYCLE_TIME

  Required.

  Specify the minimum tester cycle time. Refer to the following example:



- MAX_PULSE_WIDTH

  Required.

  Specify the maximum duration of a tester pulse.

- MIN_PULSE_WIDTH

  Required.

  Specify the minimum duration of a tester pulse. This parameter applies in the absence of an explicit clock or high function receiver min pulse width specifications, e.g, AC_MIN_PULSE_WIDTH. This value should take into account tester and technology capabilities and requirements.

- AC_MIN_PULSE_WIDTH

  Optional.

Specify the minimum duration of A_SHIFT_CLOCK pulses. In the absence of this value the MIN_PULSE_ WIDTH value will be used for all A_SHIFT_CLOCKs. If a PI is both an A_SHIFT_CLOCK and a C-clock, the larger width specification will be used.

- BC_MIN_PULSE_WIDTH

  Optional.

  Specify the minimum duration of B_SHIFT_CLOCK pulses. In the absence of this value the MIN_PULSE_ WIDTH value will be used for all B_SHIFT_CLOCKs. If a PI is both a B_SHIFT_CLOCK and a C-clock, the larger width specification will be used.

- SC_MIN_PULSE_WIDTH

  Optional.

  Specify the minimum duration of C-clock pulses. In the absence of this value the MIN_PULSE_ WIDTH value will be used for all C-clocks. If a PI is both a C-clock and either an A_SHIFT_CLOCK, B_SHIFT_CLOCK, or E-clock, the larger width specification will be used.

- EC_MIN_PULSE_WIDTH

  Optional.

  Specify the minimum duration of E-clock pulses. In the absence of this value the MIN_PULSE_ WIDTH value will be used for all E-clocks. If a PI is both an E-clock and a C-clock, the larger width specification will be used.

- HF_MIN_PULSE_WIDTH

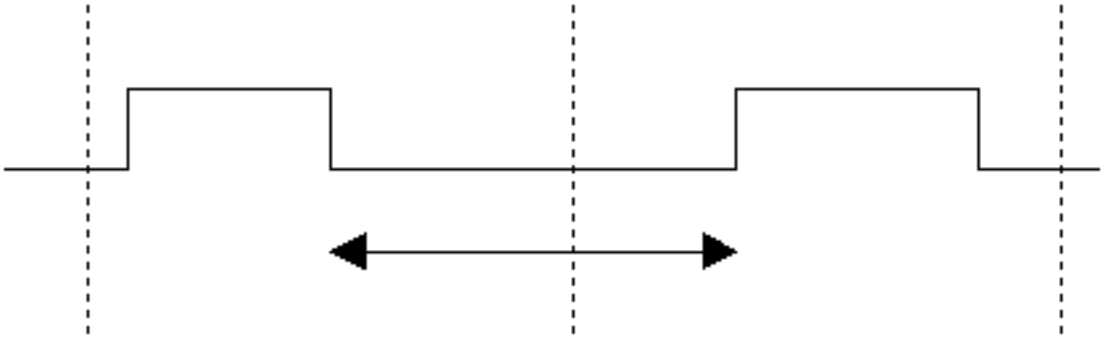
  Optional.

  Specify the minimum duration of a pulse which can be propagated by the high function receivers. High function receivers are identified in the technology rule by the HF_MIN_PULSE_WIDTH attribute on the receiver's input pin. In the absence of this value, the MIN_PULSE_WIDTH value will be used for all high function receivers. If a high function receiver is also a clock (AC, BC, SC, or EC), the clock minimum pulse width value, if specified, will override this value.

- MIN_PULSE_OFF

  Required.

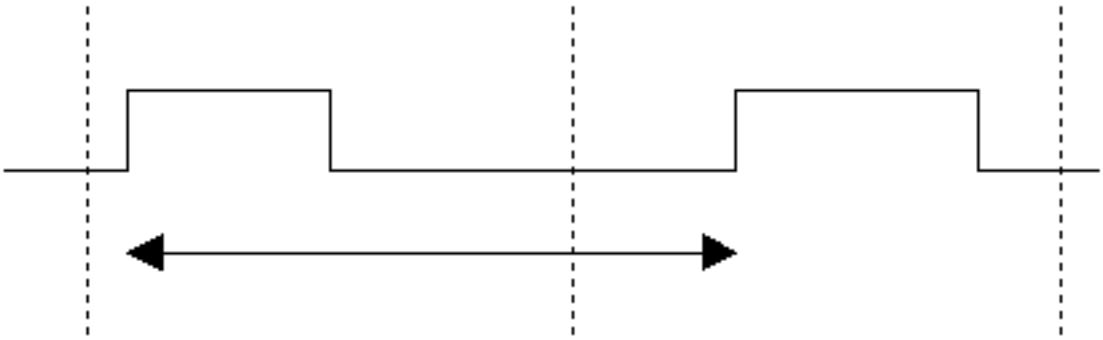
Specify the minimum time between pulses on the same pin. Refer to the following example:



- MIN_TIME_LEADING_TO_LEADING

  Required.

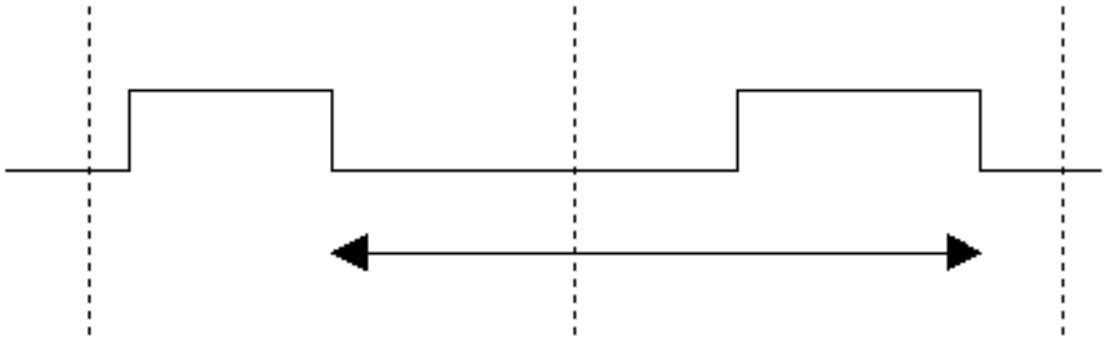
  Specify the minimum time between the leading edge of a pulse and the leading edge of the next pulse on the same pin. Refer to the following example:



- MIN_TIME_TRAILING_TO_TRAILING

  Required.

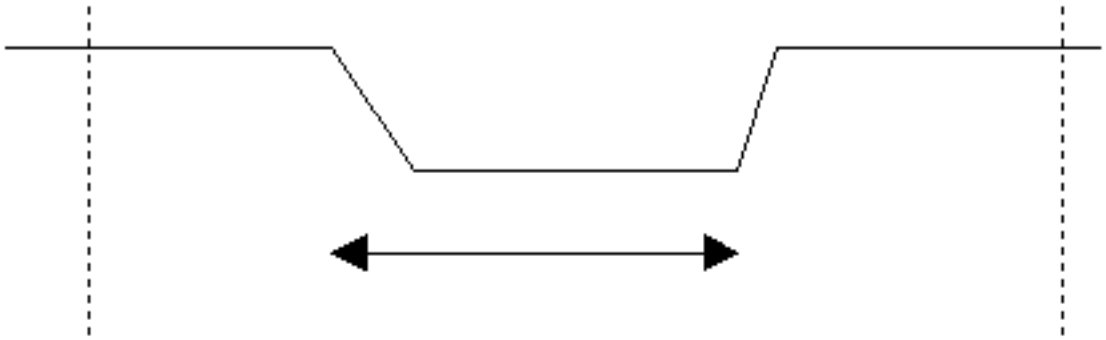
Specify the minimum time between the trailing edge of a pulse and the trailing edge of the next pulse on the same pin. Refer to the following example:



- MIN_STIM_STABLE

  Required.

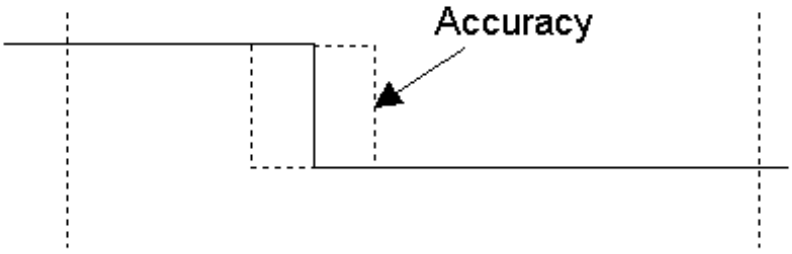
  Specify the minimum time that a non-clock pin must be stable before another event can be applied to that pin. Refer to the following example:



- ACCURACY

  Required.

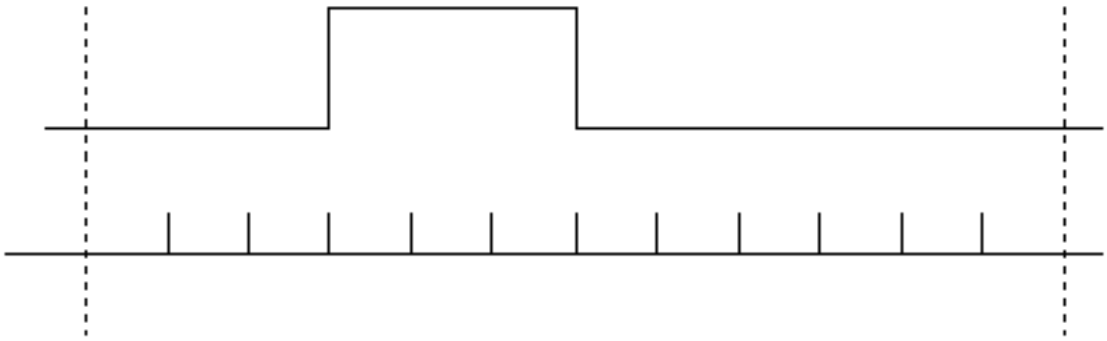
Specify the tester tolerance or difference in time (+ or -) between when an edge is programmed to arrive and when it actually arrives at the product. Refer to the following example:



- RESOLUTION

  Required.

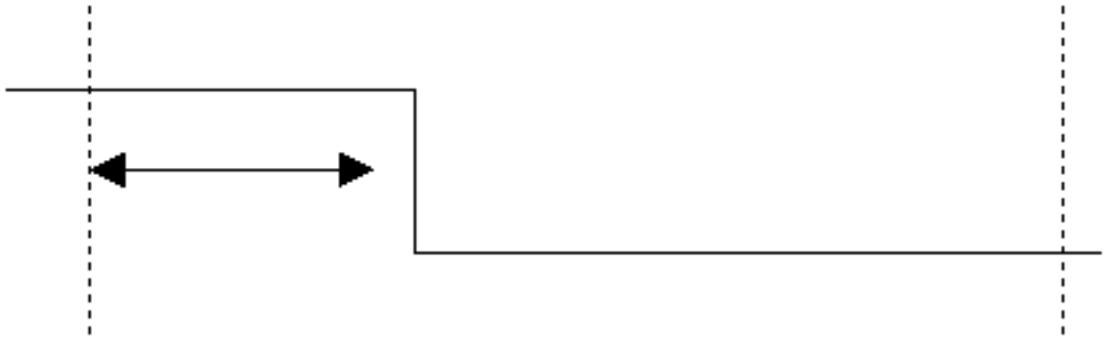
  Specify the smallest increment of time within the tester cycle. All edges must be placed at multiples of this increment. Refer to the following example:



- LEADING_PADDING

  Required.

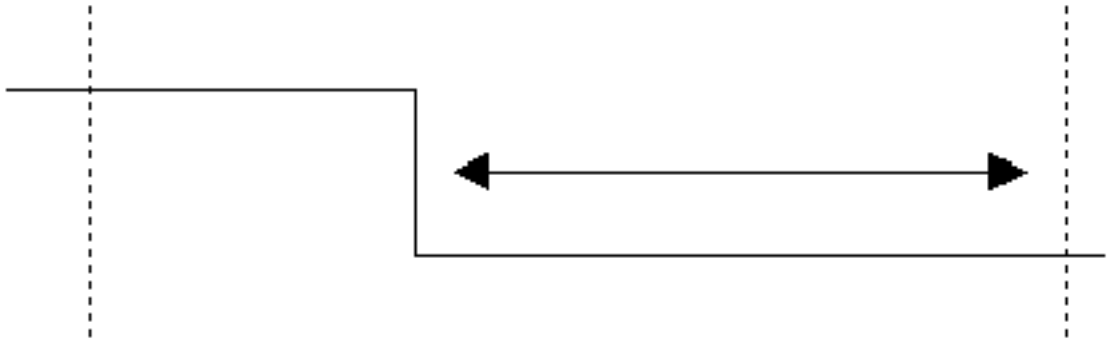
Specify the minimum time required between the beginning of a tester cycle and an edge of a signal. Refer to the following example:



- TRAILING_PADDING

  Required.

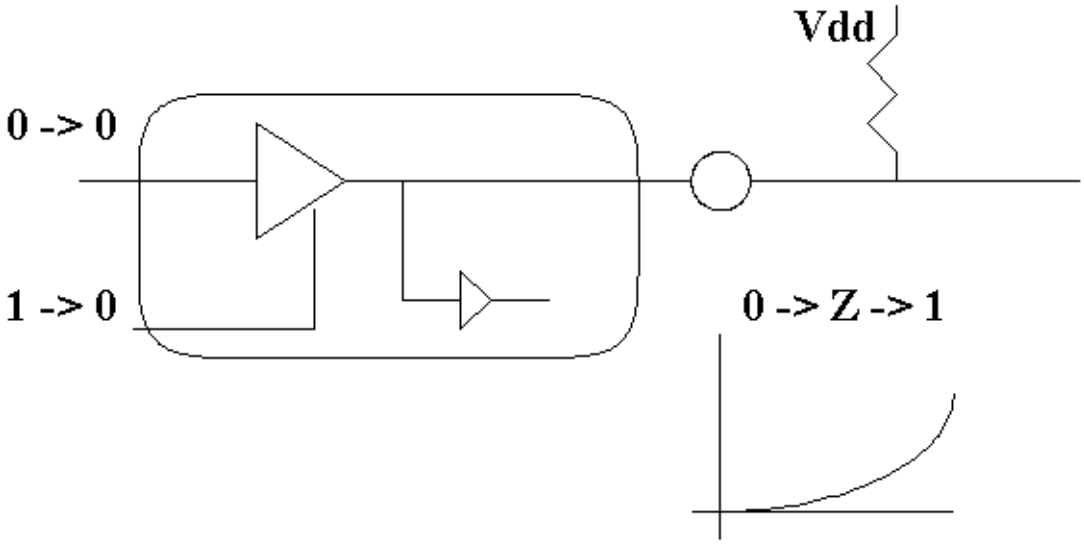
  Specify the minimum time required between the last signal edge and the end of the tester cycle. Refer to the following example:



- TERM_TIME_TO_1

  Required if the tester supports termination.

Specify the longest time it takes for tester termination to pull a signal from high impedance to a stable logic one. Refer to the following example:



- TERM_TIME_TO_0

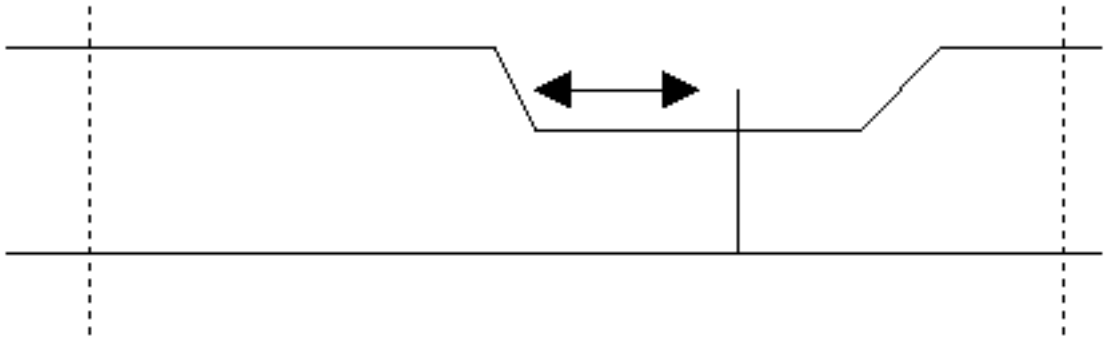
  Required if the tester supports termination.

  Specify the longest time it takes for tester termination to pull a signal from high impedance to a stable logic zero.

- STROBE_SETUP

  Required.

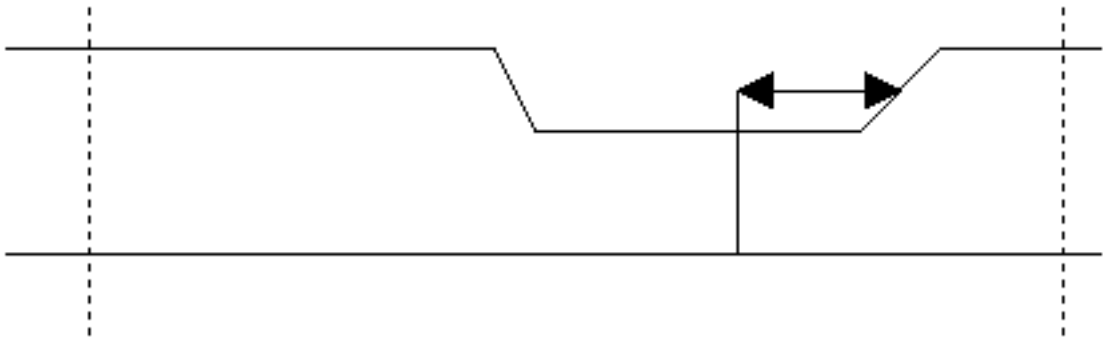
Specify the minimum time data must be stable before the tester PO is strobed. Refer to the following example:



- STROBE_HOLD

  Required.

  Specify the minimum time after a PO is strobed that the data must be kept stable. Refer to the following example:



- MIN_SCAN_CYCLE_TIME

  Required.

  Specify the minimum tester cycle when scanning. This is a complete shift cycle (i.e., A-B for LSSD, A-E-B for GSD).

- MIN_SCAN_PULSE_WIDTH

  Required.

  Specify the minimum pulse duration allowed for the scan clocks.

- DC_CYCLE_TIME

  Required.

  Specify the size of the static cycles when timings are not generated. This is used when there is no timing information on the product to generate timing data and for the generation of timeplates during WGL Export.

- DC_PULSE_WIDTH

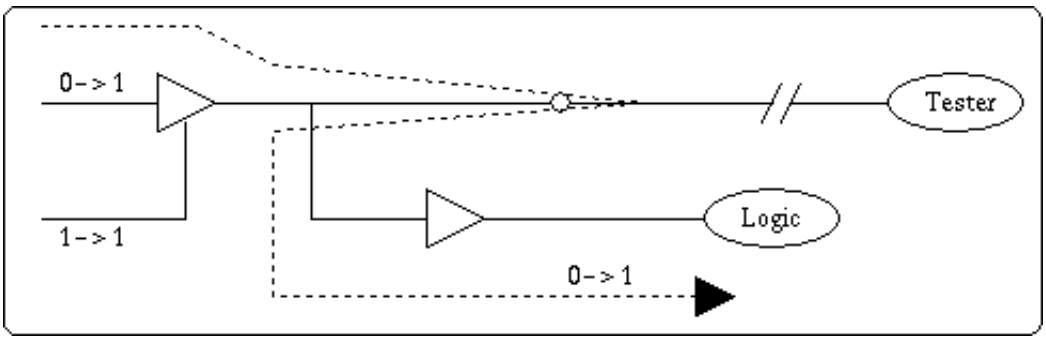
  Required.

  Specify the pulse duration to be used when no timing information exists for the clock and for the generation of timeplates during WGL Export.

- CONTACTED_DRIVER_RECEIVER_DELAY

  Optional.

  Specify a delay duration. This parameter accounts for the additional delay across tester-contacted Common Input/Output (CIO) pins. The extra delay is caused by the additional RC of the probe and tester circuitry. Refer to the following example:



DELAY_PROCESSING

The DELAY_PROCESSING statement is used to specify information about how the delays are to be processed by the delay calculator and the Encounter Test Timing Tool. DELAY_CALC_MODE and TESTER_CONDITIONS affect the delay calculation, and the CLOCK_GATING keywords affect the Encounter Test processing.

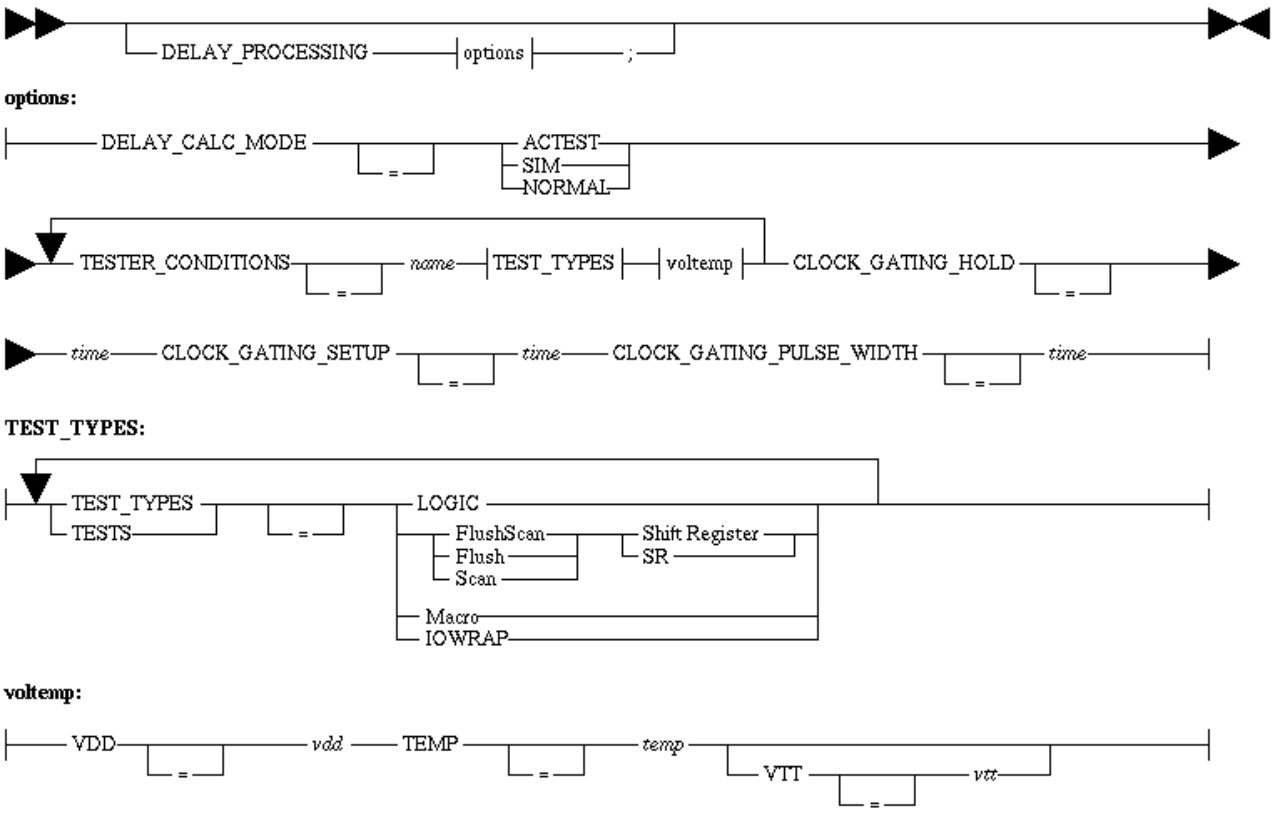
All times are specified by a positive decimal followed by ps, ns, us, ms or s. The VTT and VDD values should be positive integers in terms of volts. The TEMP values should be positive integers in terms of Centigrade.

The character string *name* must be less than 32 characters long or it will be truncated to the first 31 characters.

If the DELAY_PROCESSING statement is not included in the TDR, a delay model can still be generated and timing generation can still be performed. The parameters must be specified at runtime.

The syntax for the DELAY_PROCESSING statement is shown in the following figure.

Figure C-7 DELAY_PROCESSING Statement Syntax



- DELAY_CALC_MODE

  Required.

  This specifies what portion of the delay rules the delay calculator should use to generate the delays.

  Specify one of the following:

- ACTEST

This option tells the delay calculator to use the delay test portion of the delay rules. This is needed since not all modes of the rules specify the paths within technology cells that are used just during test.

- SIM

  If there are special modes within the rules used just for simulation these modes should be used during the calculation.

- NORMAL

  This option specifies to generate the delays for the timing analysis tool. This uses the default mode of the delay rules.

- TESTER_CONDITIONS

  Required.

  This specifies the conditions under which the product will be tested. Each set of conditions is given a name. For example, if scan chain tests and logic tests are run under different conditions (i.e., scan chain test is run with Vdd = 3.0 volts TEMP = 85 C and logic test is run with Vdd = 5.0 volts TEMP = 100 C), there will be a condition name for the scan chain tests (for example, SRT) and a condition name for the logic tests (for example, LOGIC). The condition name can be anything, but it is recommended that it relate to the type of test being run. A DELAY_PROCESSING statement can have a maximum of four TESTER_CONDITIONS attributes specified.

- TEST_TYPES

  This is the type(s) of tests the delays can be used for. There can be multiple conditions that relate to a single TEST_TYPES. For example, if LSSD flush tests are to be run at multiple temperature and voltage settings, each setting would have a different name but the same TEST_TYPES (TEST_TYPES=SR).

- VDD

  Required.

  Specify the voltage at which the product will be tested.

- TEMP

  Required.

  Specify the temperature in Celsius at which the product will be tested.

- VTT

  Optional.

Specify the second power supply voltage at which the product will be tested.

- CLOCK_GATING_HOLD

  Required.

  Specify the length of time that a gating signal must be stable after the trailing edge of a clock pulse has arrived at another input of the same gate.

- CLOCK_GATING_SETUP

  Required.

  Specify the length of time that a gating signal must be stable before the leading edge of a clock pulse arrives at a clock gate.

- CLOCK_GATING_PULSE_WIDTH

  Required.

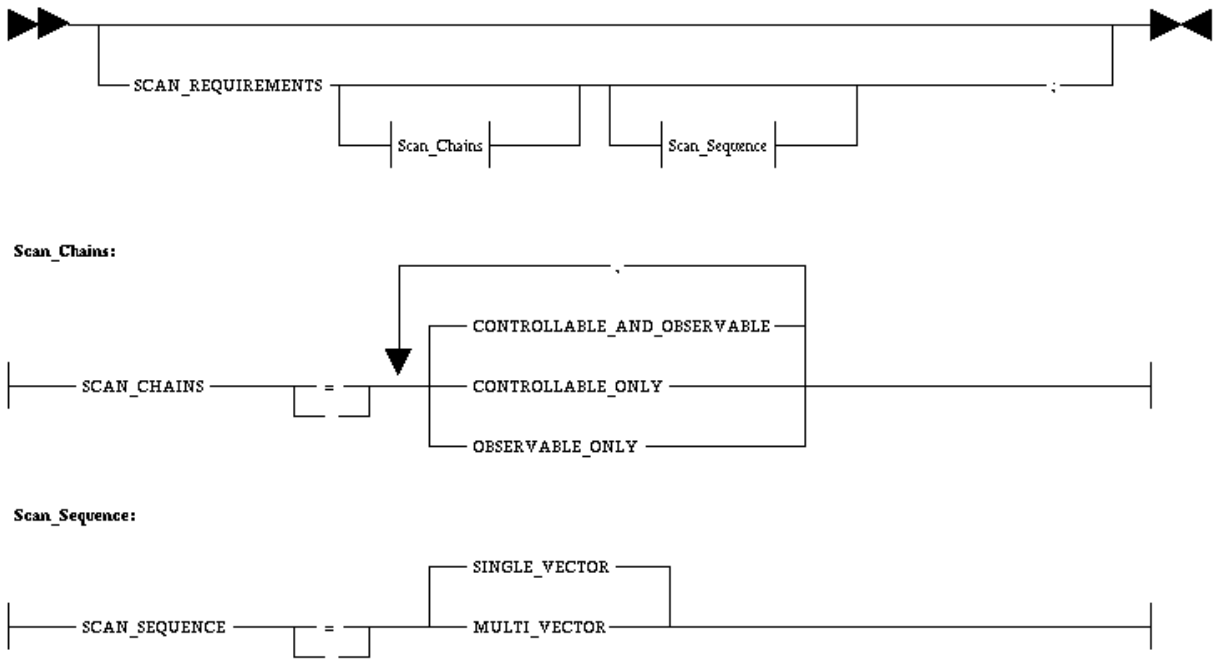
  Specify the minimum duration of a clock pulse to propagate through a gate.

SCAN_REQUIREMENTS

SCAN_REQUIREMENTS is an optional statement present only if there are specific stim and measure capabilities placed on the scan chains of the part.

The syntax for the SCAN_REQUIREMENTS statement is shown in the following figure.

Figure C-8 SCAN_REQUIREMENTS Statement Syntax



Specify the types of SCAN_CHAINS which are acceptable. More than one type can be specified. Any type not specified is unacceptable.

- CONTROLLABLE_AND_OBSERVABLE

  Scan chains which are both controllable and observable are acceptable. This is the default.

- CONTROLLABLE_ONLY

  Scan chains which are only controllable are acceptable.

- OBSERVABLE_ONLY

  Scan chains which are only observable are acceptable.

Specify the type of SCAN_SEQUENCE that is acceptable.

- SINGLE_VECTOR

  A single vector scan sequence is the default.

- MULTI_VECTOR

A multi-vector scan sequence is acceptable.

Sample TDRs

The following are example TDRs that illustrate how the statements might be coded.

```
/*********************************************************/
/* Sample TDR                                            */
/*********************************************************/
TDR_Def
     OWNER = "Owner Name"
     DATE = yyyy/mm/dd
     TESTER = "Dummy"
     CONTACT = "Contact Name"
     ;

/*  The TEST_PINs statement identifies the test pin resources that    */
/* are available for the specified tester. Encounter Test uses this   */
/* information to determine if the circuit uses more resources than   */
/* are available on the target tester.                                */


TEST_PINS
     FULL_FUNCTION_PIN_LIMIT = 4096
     STORED_PATTERN_DEPTH = 1000000
     CLOCK_PINS = 64
     SCAN_IN_PINS = 256
     STORED_PATTERN_SCAN_DEPTH = 16383
     ;

/*  The TERMINATION statement tells Encounter Test whether the tester can  */
/* provide a termination value for 3-state primary output or                */
/* bidirectional pins. It also indicates whether the tester's             */
/* supplied termination will dominate any termination that may exist      */
/* on the product (refer to the description of the globalterm             */
/* parameter in the Encounter Test: Reference: Commands for how to specify tester
*/
/* termination to be used when the patterns are applied).                 */

TERMINATION
     ZERO, ONE
     DOMINANCE = TESTER
     INTERNALLY_SEEN = X
     ;
/*  The MEASURE statement tells Encounter Test how a high impedence (Z)    */
/* should be measured by the tester. Either the tester will measure       */
/* the value as resolved by the application of product and/or tester       */
/* termination, or it will measure Z regardless of the termination.       */

MEASURE
     HIGH_Z = Z

/*********************************************************/
/* Sample TDR for Delay Test                             */
*********************************************************
TDR_Def
     OWNER = "Owner Name"
     DATE = yyyy/mm/yy
```

```
     TESTER = "Dummy"
     CONTACT = "Contact Name"
     ;

TEST_PINS
     FULL_FUNCTION_PIN_LIMIT = 4096
     STORED_PATTERN_DEPTH = 1000000
     CLOCK_PINS = 64
     SCAN_IN_PINS = 256
     STORED_PATTERN_SCAN_DEPTH = 16383
     PARAMETRIC_MEASURE_UNITS = 4096
;
TERMINATION
     ZERO, ONE
     DOMINANCE = TESTER
     INTERNALLY_SEEN = X
;
MEASURE
     HIGH_Z = Z
     EXPLICIT = YES
     CURRENT = 20
;

PIN_TIMING
     TIMING_RESOURCE=SHARED_RESOURCE CLOCKS=2 NON_CLOCKS=2 PO_STROBES=1
     MAX_PULSES = 1
     MAX_STIMS   = 1
     MAX_MEASURES = 1
     MAX_CYCLE_TIME = 1 MS
     MIN_CYCLE_TIME = 10 NS
     MAX_PULSE_WIDTH = 100 NS
     MIN_PULSE_WIDTH = 5 NS
     HF_MIN_PULSE_WIDTH = 2.5 NS
     MIN_PULSE_OFF = 2 NS
     MIN_TIME_LEADING_TO_LEADING  = 10 NS
     MIN_TIME_TRAILING_TO_TRAILING  = 10 NS
     MIN_STIM_STABLE = 2 NS
     ACCURACY = 200 PS
     RESOLUTION = 25 PS
     LEADING_PADDING = 0 S
     TRAILING_PADDING = 0 S
     TERM_TIME_TO_1 = 50 NS
     TERM_TIME_TO_0 = 75 NS
     STROBE_SETUP = 100 PS
     STROBE_HOLD =  300 PS
     MIN_SCAN_CYCLE_TIME = 40 NS
     MIN_SCAN_PULSE_WIDTH = 10 NS
     DC_CYCLE_TIME = 200 NS
     DC_PULSE_WIDTH  = 50 NS
;

DELAY_PROCESSING
     DELAY_CALC_MODE = ACTEST
     TESTER_CONDITIONS = "Scan Chain Test"
     TEST_TYPES=SR VDD = 3.0 TEMP = 20
     TESTER_CONDITIONS = "Logic_&_Array_Test"
     TEST_TYPES=LOGIC TEST_TYPES=MACRO
     VDD = 5.0 TEMP = 60
     CLOCK_GATING_HOLD = 500 PS
     CLOCK_GATING_SETUP = 250 PS
     CLOCK_GATING_PULSE_WIDTH  = 2 NS
```

D

Identifying Inactive Logic

Inactive logic is the logic in the design that is not observable and is not processed by the Encounter Test commands. The logic may be unobservable due to things in the design source such as “dangling logic” (logic that is not connected to anything) or TIE blocks that keep values from propagating.

The use of Test Inhibit (TI) signals (on either primary inputs or fixed value latches) may cause some logic to be inactive in that specific test mode. In addition, the use of Test_Constraint (TC) signals may cause yet more logic to become inactive.

Logic may be classified as inactive for the "test generation state" and active for the "scan state" and vice-versa. Thus, most system-function logic may be inactive in the scan state, resulting in improved application program performance when processing scan tests, for example. Logic that is inactive in both the test generation state and the scan state is displayed as greyed out in the GUI design schematic viewer.

For purposes of identifying inactive logic in the test generation state, `TI` and `TC` signals are held at their specified states as described above. Similarly, for the identification of inactive logic in the scan state, `TI` and `SE` signals are held at their specified states.

Yet another cause of inactive logic is the existence of cut points. Cut points cause the upstream logic to be unobservable as far as Encounter Test programs are concerned. Although the logic is, in fact, observable, Encounter Test marks logic that is observable only through cut points as inactive. This logic is also given a special designation, “OPC”. Any node which is the source of a cut point net is considered active (provided that the cut point net was active to begin with.

In the case of inactive logic unique to a test mode (due to the use of TI or TC signals), faults in the inactive logic are also considered to be inactive in that test mode. A fault is considered inactive in the test mode if it is on logic which is inactive in both the test generation state and the scan state. Fault coverage statistics for a test mode are based on the test mode's active faults only. For globally inactive logic (due to “dangling” or TIE blocks), untestable faults are simply excluded from the fault model.

For Boundary Scan Internal or External test modes, large sections of the design may be considered logically inactive. For a Boundary Scan Internal test mode, the logic feeding only

to non-test function primary output pins will be inactive; also the logic between non-test function primary input pins and boundary scan latches is inactive due to Test Inhibit (TI) test mode controls, but this is not guaranteed because the logic may fan out to observable points through paths not blocked by the TI signals. For a Boundary Scan External test mode, usually only the logic between the I/O pins and the boundary scan latches is considered active; most of the logic internal to the design is deemed to be inactive. An exception to this occurs if it is necessary to supplement the normal external logic with chip PI-to-PO paths to enable the creation of a kind of internal-external hybrid boundary model in support of AC path testing at the MCM level. This is necessary if the MCM paths to be tested are more than just the driver-to-receiver interconnect paths, but must also include some paths that traverse whole chips. To accommodate this need, Encounter Test supports the user specification of a set of chip primary output nets which must be fully traced back to chip primary inputs. These traces will supplement the normal external logic. For any test mode, all logic necessary for correct operation of all defined scan chains is considered to be active. For Boundary Scan External test modes it is advisable to define only those scan chains necessary for scanning just the boundary latches to avoid including too much active logic.

Reporting Inactive Logic

Use the `report_faults` command to report a list of faults that are identified as inactive. A sample command is given below:

```
report_faults inactiveallmodes=yes reportdetails=yes
```

This will create a list of all the inactive faults associated with the inactive logic. You can then determine the modules in the design that have the inactive logic.

Debugging Inactive Logic

You need to debug the inactive logic reported by `build_testmode` to increase the test coverage for a test mode. A potential debug technique is to report the cause of inactive state for each inactive instance and net in the test mode. The following is a sample debug script, which when executed after `build_testmode`, reports wherever the logic has a hard tie or is tied in the inhibit or constraint states:

```
#! /bin/sh
# The following ten lines allow xxx to be executed no matter where perl is in your
path
if [ $TB_PERL ] ; then
   `which $TB_PERL` -x -S $0 $*
else
   echo "Encounter Test environment is not available."
   echo "Use runtb -c to set up the environment, or run from the Encounter Test GUI"
   `which perl` -x -S $0 $*
fi
exit $?
```

```
#! perl
BEGIN { push(@INC,split(':',$ENV{PATH})); }
use TBX;
#
# check for help invocation
#
foreach $argument ( @ARGV )
{
  if ( ( $argument eq "-h" ) or ( $argument eq "-H" ) )
  {
    print "report_inactive_logic.pl workdir=<workdir> testmode=<testmode>\n";
    print "   Reports the inactive logic in a design\n\n";
    exit;
  }
}
#
# extract the parameters
#
if ( @ARGV < 2 )
{
   die "Workdir and testmode must be specified\n";
}
$workdir = "";
$testmode = "";
foreach $argument ( @ARGV )
{
  if ( $argument =~ /workdir=/i )
  {
   $workdir = $argument;
   $workdir =~ s/^.*=//;
  }
  elsif ( $argument =~ /testmode=/i )
  {
   $testmode = $argument;
   $testmode =~ s/^.*=//;
  }
  else
  {
   die "Illegal keyword - ".$argument."\n";
  }
}
#
# select the project, entity and test_mode
#
if ( TBX::selectWorkDir($workdir) )
{
  die TBX::getErrorMsg;
}
if ( TBX::selectTestMode($testmode) )
{
  die TBX::getErrorMsg;
}
#
# print the header
#
print "\n\n";
print "report_inactive_logic\n\n";
print "Workdir".TBX::currentWorkDir()."\n";
print "TestMode".TBX::currentTestMode()."\n\n";
#
# get the number of nodes in the design
```

```
#
$maxNodes = TBX::numNodes();
if (TBX::getReturnCode)
{
  die TBX::getErrorMsg;
}
#
# go through all nodes looking for inactive ones
# store in an associative array
#
%inactive_nodes = ();
$inactive = 0;
for ( $node = 1 ; $node <= $maxNodes ; $node++ )
{
   $active=TBX::isActiveFromNode($node);
   if (TBX::getReturnCode)
   {
    die TBX::getErrorMsg;
   }
   if ( not $active )
   {
    $inactive += 1;
    $inactive_nodes{$node} = 0;
   }
}
#
# check whether the node is at a fixed state in the Test Constraint state
#
TBX::setCircuitState("Test Constraint");
if (TBX::getReturnCode)
{
  die TBX::getErrorMsg;
}
foreach $node ( keys ( %inactive_nodes ) )
{
  $valueStr = TBX::valueFromNode($node);
  if ( $valueStr ne "X" )
  {
    $inactive_nodes{$node} = 1;
  }
}
#
# check whether the node is at a fixed state in the Test Inhibit state
#
TBX::setCircuitState("Test Inhibit");
if (TBX::getReturnCode)
{
  die TBX::getErrorMsg;
}
foreach $node ( keys ( %inactive_nodes ) )
{
   $valueStr = TBX::valueFromNode($node);
   if ( $valueStr ne "X" )
   {
     $inactive_nodes{$node} = 2;
   }
}
#
# check whether the node is at a fixed state in the Tie state
#
TBX::setCircuitState("Tie Only");
```

```
if (TBX::getReturnCode)
{
  die TBX::getErrorMsg;
}
foreach $node ( keys ( %inactive_nodes ) )
{
  $valueStr = TBX::valueFromNode($node);
  if ( $valueStr ne "X" )
  {
    $inactive_nodes{$node} = 3;
  }
}
#
# print the list of nodes and fixed states
#
foreach $node ( sort keys ( %inactive_nodes ) )
{
  #
  # print the node type and name
  #
  $name=TBX::blockNameFromNode($node);
  if ( $name ne "" )
  {
    print "Block ";
    printf ( "%10d" , $node );
    print " ".$name;
  }
  else
  {
    $name=TBX::highestNetNameFromNode($node);
    if ( $name ne "" )
    {
      print "Net   ";
      printf ( "%10d" , $node );
      print " ".$name;
    }
    else
    {
      print "Other ";
      printf ( "%10d" , $node );
    }
  }
  if ( $inactive_nodes{$node} == 3 )
  {
    print "Hard Tie\n";
  }
  elsif ( $inactive_nodes{$node} == 2 )
  {
    print "Tied in Test Inhibit\n";
  }
  elsif ( $inactive_nodes{$node} == 1 )
  {
    print "Tied in Test Constraint\n";
  }
  else
  {
    print "\n";
  }
}
$percent_inactive = (($inactive*100)/$maxNodes) ;
print "\n\n";
```

```
print "Total nodes         - ".$maxNodes."\n";
print "Inactive nodes      - ".$inactive."\n";
print "Percentage inactive - ";
printf ( "%4.2f" , $percent_inactive );
print "%\n\n";
```

Note: The above-given script is just a sample and is not supported formally by Encounter Test.

The following is the sample output of the script:

```
Encounter Test environment is not available.
Use runtb -c to set up the environemnt, or run from the Encounter Test GUI
report_inactive_logic
Workdir .
TestModeFULLSCAN
Net           10 DLX_CHIPTOP_TMS
Block       1248 DLX_CORE.B_REG.CLOCK_GATE.i_3.__i0
Block       1573 DLX_CORE.C_REG.CLOCK_GATE.i_3.__i0
Block       1866 DLX_CORE.INSTR_REG.CLOCK_GATE.i_3.__i0
Block       2287 DLX_CORE.MAR_REG.CLOCK_GATE.i_3.__i0
Block      23845 DLX_CORE.TSV008_CORRECTOR.CLOCK_GATE.i_3.__i0
Block      23847 JTAG_MODULE.BYPASS_REGISTER.TBS_TDO_reg.I0.dff_primitive
Block      23848 JTAG_MODULE.BYPASS_REGISTER.TBS_TDO_reg.I0.dff_primitive
Block      23849 JTAG_MODULE.BYPASS_REGISTER.TBS_TDO_reg.I0.dff_primitive
Block      23850 JTAG_MODULE.BYPASS_REGISTER.TBS_TDO_reg.I0.dff_primitive
Block      23851 JTAG_MODULE.BYPASS_REGISTER.TBS_TDO_reg.I0.udp_0_or Tied in Test
Inhibit
Block      23852 JTAG_MODULE.BYPASS_REGISTER.TBS_TDO_reg.I0.udp_st0_row0_and Tied
in Test Inhibit
Block      23853 JTAG_MODULE.BYPASS_REGISTER.TBS_TDO_reg.I1
Block      23854 JTAG_MODULE.BYPASS_REGISTER.TBS_TDO_reg.I2
Block      23855 JTAG_MODULE.BYPASS_REGISTER.TBS_TDO_reg.IC
Block      23856 JTAG_MODULE.DFT__0.I0Tied in Test Inhibit
Block      23857 JTAG_MODULE.DFT__0.I1Tied in Test Inhibit
Block      23858 JTAG_MODULE.DFT__0.I2Tied in Test Inhibit
Block      23859 JTAG_MODULE.DFT__1.I0
Block      23860 JTAG_MODULE.DFT__1.I1
Block      23861 JTAG_MODULE.DFT__10.__i0Hard Tie
Block      23862 JTAG_MODULE.DFT__11.I0Hard Tie
Block      23863 JTAG_MODULE.DFT__12.I0Hard Tie
.............................
.............................
.............................
Total nodes         - 27191
Inactive nodes      - 2450
Percentage inactive - 9.01%
```

E

Test Mode Design States

Design States

The design states described in this section are those resulting from application of specific PI, pseudo PI, and latch values that have particular significance in the test mode. Encounter Test applications use these design states as starting points for testability analysis and test generation. To aid user analysis, these design states can be set with *Reset circuit state* on the graphical user interface. For more information, refer to Reset Circuit State in the *Encounter Test: Reference: GUI*.

Tie Only

This state is essentially a “reset” of the design. All PIs and memory elements are set to X and values from TIE-blocks are propagated into the logic.

Test Inhibit

In this state all PIs and memory elements are set to X, values from Tie blocks are propagated into the logic, and Test Inhibit signals (on primary inputs, pseudo PIs, and fixed-value latches) are set to the test function value and propagated into the logic. This is the base design state which Encounter Test never violates, either for test generation or for scanning.

Test Constraint

In this state all PIs and memory elements are set to X, values from Tie blocks are propagated into the logic, and Test Inhibit signals (on primary inputs, pseudo PIs, and fixed-value latches) and Test Constraint signals (on primary inputs, pseudo PIs, and latches) are set to the test function value and propagated into the logic. This is the base test generation design state, more confining than the Test Inhibit state (if there are TC primary inputs), which Encounter Test test generators never violate.

Test Constraint and Clocks Off

The test constraint and clocks off state forces the clocks off to all memory elements required at a constant value for test generation, but not necessarily for scanning. To reach this state, Encounter Test sets:

- all TIE Blocks to value
- all TEST_INHIBIT pins and latches to value
- all TEST_CONSTRAINT pins and test constraint latches to value
- all clocks off

Encounter Test will not allow more than one clock at a time to depart from the Test Constraint and Clocks Off state in the process of test generation. (This state does not apply to the scanning operation.)

Mode Initialization

The mode initialization state is the starting state for the mode. To reach this state, Encounter Test sets:

- the design to the Test Constraint and Clocks Off state
- the initial values on all latches as determined by the mode initialization sequence
- the initial values on all RAMs as determined by the mode initialization sequence
- the final stimulus on all primary inputs as determined by the mode initialization sequence.

See SEQUENCE_DEFINITION on page 230 for more information on Encounter Test mode initialization sequences.

Scan States

A scan state allows scan chains to be scanned. If the test mode allows all its scan chains to be scanned in parallel (this is the usual case) then the scan operation for this test mode consist of just one scan section. If it is not possible to scan all chains in parallel then the scan operation consists of multiple scan sections, each of which scans in parallel one or more scan chains. All defined scan sections are executed in sequence in order to scan all the scan chains.

For each scan section there is defined a distinct scan state. When the *Scan* option of *Set circuit state* is selected from the graphical user interface and there is only one scan section then that section's scan state is displayed directly. If, however, there are multiple scan sections then selecting the *Scan* state causes the *Select a Scan Section* window to be displayed. This shows a list of scan sections from which one can be selected for display.

Encounter Test supports scan-in and scan-out design states, differentiated as follows:

- Scan In

  This state requires that any pin defined in the scan in and global categories with polarity must be applied to their values. The design must be at this state before any input scan is performed (channel scan, Scan_Load, etc.)

- Scan Out

  This state requires that any pin defined as a SCAN_OUT_GATE (SOG) must be applied to its value. The design must be at this state before any output scan unload is performed (channel scan, Scan_Unload, etc.).

To reach these states, Encounter Test sets:

- all SCAN_ENABLE pins to the value required for scanning.
- any CLOCK_ISOLATION pins to the value required to select the scan clock function.
- any MISR_ENABLE pins to the value required for scanning (self test only).

For overlapped scan, the scan in and scan out states are identical.

However many scan sections there are, each has the following characteristics:

1. Each latch or scan-in PI is a function of only the single preceding latch or scan-in PI in its scan chain.
2. All clock inputs to non-scan memory elements are held OFF (at test function value), even when clocks required for shifting are manipulated.
3. Any shift clock to a latch may be turned ON (opposite of test function value) and OFF (test function value) by changing the corresponding clock PI.

To reach this state Encounter Test sets:

- all SCAN_ENABLE (scan selects) to the value required for scanning.
- any CLOCK_ISOLATION to the value required to select the scan clock function.
- any MISR_ENABLE pins to the value required for scanning (self test only).

Note: If you use OUTPUT_INHIBIT pins to suppress output switching noise, they will also be set to value in the scan state.

Flush Scan Chain

The flush state makes all the level sensitive latches used in all the scan chains “flushable” and a combinational signal path will exist between the scan in and the scan out pins. Thus, if a value is placed on the scan in, the effects of that value will be seen at the scan out some time later. To reach this state Encounter Test sets:

- the design to scan state (clocks, tie-blocks, test inhibits, clock isolation, scan enables, and output inhibits to value)
- all scan_A and scan_B clocks are turned on

Note: The Flush Scan Chain state is defined only for test modes for which the scan operation is composed of just one scan section - i.e., all scan chains can be scanned in parallel.

Nonscanflush Design State

The nonscanflush design state is produced when a nonscanflush sequence is defined as part of test mode creation. The test mode creation process recognizes and analyzes the sequence, replacing any Stim_PI values for pipelined pins with a Stim_PI for the associated PPIs.

Encounter Test defines this design state as the PI and PPI state in which all of the nonscanflush pipeline clocks are pulsed.

Refer to the following for additional information:

- ASSIGN
- The description of nonscanflush sequence in the “Define_Sequence” section of the *Test Pattern Data Reference*
- “prepare_pipeline_sequence” in the *Encounter Test: Reference: Commands*

MISR Observe

This state is created by applying the values specified on the MISR_READ (MRD) pin(s) while in the Scan In state. Note that the MRD test function may be specified on pins that have other functions in the Scan In state (such as SE). In this case, the MRD value overrides the Scan

In state value. While in this state, the contents of the on-board MISR should be available on the MISR_OBSERVE (MO) pins.

MISR Reset

This state is created by applying the MISR_RESET_ENABLE (MRE) pins to their value while in the Scan In state. Note that the MRE test function may be specified on pins that have other functions in the Scan In state (such as SE). In this case, the MRE value overrides the Scan In state value. While in this state, the MISR latches can be reset to a known state if the `MISR_Reset` clock pulse is done followed by a B clock.

PRPG Load

This state is created by applying the values specified for the PRPG_LOAD (PGE) pin(s) while the design is in the Stability state.

Channel Mask Load

This state is created by applying the `channelmaskprecond` sequence on top of the scan state. The Channel Mask Load design state allows analysis of problems in the loading of channel masks. Channel mask pins denote from where mask bits are loaded, however if no `CML` pins are specified, Encounter Test assumes SI pins serve this function.

OPCG Load State

This state is created if at least one OPCG Load Clock (`OLC`) is present in the design. The OPCG design state allows analysis of problems in the loading of OPCG registers. Refer to OPCG Test Mode in *Encounter Test: Flow: OPCG* for additional information.

F

Boundary Scan Design Language

BSDL is recommended by the IEEE 1149.1 standard to specify boundary scan. Refer to "Boundary Scan Attributes" in *Encounter Test: Reference: Legacy Functions* for details.

This section describes methods for:

- "BSDL Extension - Port Alias" on page 301
- "BSDL Extension for Identifying Image Unwired Ports" on page 303
- "Parsing BSDL" on page 304
- "Writing BSDL" on page 307

Important

Do not use device names with binary patterns in BSDL as it will result in syntax errors when running the verify_11491_boundary command. For example, using "entity x1 is" will cause a syntax error.

BSDL Extension - Port Alias

Encounter Test applications which use BSDL (Boundary Scan Description Language) require that the BSDL port names correlate with the Encounter Test model pin names. The `Test_Port_Alias` BSDL extension allows you to correlate differently named ports that are logically the same, without having to change the port names in the BSDL or the pin names in the design source. For example, if the BSDL port name is CLK and the corresponding pin name in the Encounter Test model is AA100AA0[0], then you can include the following port alias statement in the BSDL to enable Encounter Test 1149.1 applications to make the name correlation:

```
attribute Test_Port_Alias :BSDL_EXTENSION;
attribute Test_Port_Alias of  <entityname>: entity is
     "CLK :AA100AA0[0]";
```

Note: no spaces are allowed in the `Test_Port_Alias` string.

Another source of the Port Alias statement is the `write_bsdl` tool, which generates a `Test_Port_Alias` statement when it determines that a legitimate BSDL port name cannot be created from a Encounter Test pin name.

Sample BSDL File

This sample BSDL has a *Test_Port_Alias* statement near the bottom of the file.

```
entity ttl74bct8374tda is generic (PHYSICAL_PIN_MAP : string := "DW_PACKAGE");

  port (CLK:in bit; Q:out bit_vector(1 to 8); D:in bit_vector(1 to 8);
     GND, VCC:linkage bit; OC_NEG:in bit; TDO:out bit; TMS, TDI, TCK:in bit);

  use STD_1149_1_1990.all;   -- Get Std 1149.1-1990 attributes and definitions

  attribute PIN_MAP of ttl74bct8374tda : entity is PHYSICAL_PIN_MAP;

  constant DW_PACKAGE:PIN_MAP_STRING:="CLK:1, Q:(2,3,4,5,7,8,9,10)," &
           "D:(23,22,21,20,19,17,16,15)," &
           "GND:6, VCC:18, OC_NEG:24," &
           "TDO:11, TMS:12, TCK:13, TDI:14";

  constant FK_PACKAGE:PIN_MAP_STRING:="CLK:9, Q:(10,11,12,13,16,17,18,19)," &
           "D:(6,5,4,3,2,27,26,25)," &
           "GND:14, VCC:28, OC_NEG:7," &
           "TDO:20, TMS:21, TCK:23, TDI:24";

  attribute TAP_SCAN_IN    of TDI : signal is true;
  attribute TAP_SCAN_MODE  of TMS : signal is true;
  attribute TAP_SCAN_OUT   of TDO : signal is true;
  attribute TAP_SCAN_CLOCK of TCK : signal is (20.0e6, BOTH);

  attribute INSTRUCTION_LENGTH of ttl74bct8374tda : entity is 8;

  attribute INSTRUCTION_OPCODE of ttl74bct8374tda : entity is
       "BYPASS (11111111, 10001000, 00000101, 10000100, 00000001),"   &
       "EXTEST (00000000, 10000000),"  &
       "SAMPLE (00000010, 10000010),"  &
       "INTEST (00000011, 10000011),"  &
       "HIGHZ  (00000110, 10000110),"  &     -- Boundary Hi-Z
       "SETBYP (00000111, 10000111),"  &     -- Boundary 1/0
       "CELLTST(00001100, 10001100),"  &     -- Boundary selftest normal
       "SCANCN (00001110, 10001110),"  &     -- BCR Scan normal
       "SCANCT (00001111, 10001111)";        -- BCR Scan test

  attribute INSTRUCTION_CAPTURE of ttl74bct8374tda : entity is "10000001";

  attribute REGISTER_ACCESS of ttl74bct8374tda : entity is
       "BOUNDARY (CELLTST),"   &
       "BYPASS (SETBYP),"   &
       "BCR[2] (SCANCN, SCANCT)";   -- 2-bit Boundary Control Register

  attribute BOUNDARY_REGISTER of ttl74bct8374tda : entity is
-- num cell  port   function safe [ccell disval rslt]
        "17 (BC_1, CLK,    input,   X),"  &
        "16 (BC_1, OC_NEG, input,   X),"  &  -- Merged Input/Control
        "16 (BC_1, *,      control, 1),"  &  -- Merged Input/Control
        "15 (BC_1, D(1),   input,   X),"  &
```

```
        "14 (BC_1, D(2),   input,   X),"  &
        "13 (BC_1, D(3),   input,   X),"  &
        "12 (BC_1, D(4),   input,   X),"  &
        "11 (BC_1, D(5),   input,   X),"  &
        "10 (BC_1, D(6),   input,   X),"  &
        "9  (BC_1, D(7),   input,   X),"  &
        "8  (BC_1, D(8),   input,   X),"  &
        "7  (BC_1, Q(1),   output3, X,  16, 1, Z)," &  -- cell 16 @ 1 -> Hi-Z.
        "6  (BC_1, Q(2),   output3, X,  16, 1, Z)," &
        "5  (BC_1, Q(3),   output3, X,  16, 1, Z)," &
        "4  (BC_1, Q(4),   output3, X,  16, 1, Z)," &
        "3  (BC_1, Q(5),   output3, X,  16, 1, Z)," &
        "2  (BC_1, Q(6),   output3, X,  16, 1, Z)," &
        "1  (BC_1, Q(7),   output3, X,  16, 1, Z)," &
        "0  (BC_1, Q(8),   output3, X,  16, 1, Z)";

  attribute Test_Port_Alias:BSDL_EXTENSION;

  attribute Test_Port_Alias of ttl74bct8374tda: entity is
        "CLK:TestClkPort," &
        "Q(1):TestportQ1," &
        "Q(2):TestportQ2," &
        "Q(3):TestportQ3," &
        "Q(4):TestportQ4," &
        "Q(5):TestportQ5," &
        "Q(6):TestportQ6," &
        "Q(7):TestportQ7," &
        "Q(8):TestportQ8," &
        "D(1):TestportD1," &
        "D(2):TestportD2," &
        "D(3):TestportD3," &
        "D(4):TestportD4," &
        "D(5):TestportD5," &
        "D(6):TestportD6," &
        "D(7):TestportD7," &
        "D(8):TestportD8," &
        "GND:TestportGND," &
        "VCC:TestportVCC," &
        "OC_NEG:TestportOC_NEG," &
        "TDO:TestportTDO," &
        "TMS:TestportTMS," &
        "TDI:TestportTDI," &
        "TCK:TestportTCK";

end ttl74bct8374tda;
```

BSDL Extension for Identifying Image Unwired Ports

The purpose of the Test_Image_Unwired BSDL extension is to identify logical ports which have no connection to any physical pins and to exempt them from Encounter Test's model to BSDL correlation checking. An image unwired port is a primary input/output pin in the top cell of the logic model that has no connection to a physical pin. An image unwired port appears in the logic model but does not appear in the logical port statement nor in any physical pin map statement appearing in a particular BSDL file. The port names appearing in the

Test_Image_Unwired BSDL extension must be Encounter Test logic model pin names in simple (short) name form (consistent with other Encounter Test BSDL extensions).

```
attribute Test_Image_Unwired: BSDL_EXTENSION;

attribute Test_Image_Unwired of  <entity name>: entity is
"<TB pin name>, " &         -- port not required in BSDL
"<TB pin name>, " &
"<TB pin name>, " &
"<TB pin name> ";
```

Parsing BSDL

IEEE 1149.1 *Parse BSDL* reads one or more BSDL (Boundary Scan Design Language) files and checks them for syntax and semantic errors. For a description of the BSDL language and the IEEE 1149.1 Standard, refer to *IEEE Std 1149.1b-1994 (Supplement to IEEE Std 1149.1-1990* and *IEEE Std 1149.1a-1993*).

The primary purpose of *Parse BSDL* is to determine if the BSDL contained in a file is syntactically and semantically correct. The parser reports on all deviations from the BSDL syntax. It also reports on many of the deviations from BSDL semantics. The parser does not have a check for every semantic check defined by the IEEE 1149.1 standard. For a complete list of BSDL Semantic rules that are checked, refer to "IEEE 1149.1 Rules Verification Table" in the *Verification and Analysis Reference*.

To Read BSDL, refer to "read_bsdl" in the *Encounter Test: Reference: Commands*.

Prerequisite Tasks

- Specify one or more BSDL files to parse. See "1149.1 Test Mode Input Files" on page 151.
- Import or open an Encounter Test design in order to perform BSDL model correlation checks. Refer to section Building Test Mode in *Encounter Test: Guide 1: Models.*

Input Files

The suffix `.gz`, indicating file compression, is attached to the files listed below if they were created with the storage versus performance environment variable set to `TB_MODE=COMPRESS`. Refer to "Using Commands to Manage Files and Disk Space" in the *Encounter Test: Reference: Commands* for additional information.

The following are used as input to *Parse BSDL*:

- BSDL File - required

The input BSDL file is an ASCII file containing the boundary scan description language as defined by *IEEE Std 1149.1b-1994 (Supplement to IEEE Std 1149.1-1990)* and *IEEE Std 1149.1a-1993*). The file may be created using a Test Insertion tool, the `write_bsdl` tool, or be manually created. Refer to the following for additional information:

- "Boundary Scan Attributes" in the *Encounter Test: Reference: Legacy Functions*
- "Writing BSDL" on page 307

Parse BSDL can parse multiple BSDL files at a time; you can specify any number of input BSDL input files using read_bsdl.

The parser is capable of parsing an incomplete or "partial BSDL file" provided that the BSDL file contains at least BSDL entity and end statements.

Encounter Test requires that BSDL statements be in the following order:

1. `entity`
2. `generic`
3. *`any other BSDL statements,in any order`*
4. `end`

The `modelchecks` option causes the parser to perform a rudimentary BSDL file to model correlation check. When the `modelchecks` option is requested, the parser checks the BSDL against the Encounter Test model to ensure that the entity name in the BSDL is defined as a cell name in the model. It also checks that each of the logical port names in the BSDL are defined for the cell determined to be the entity cell in the Encounter Test model. These additional checks are known as the "BSDL model correlation checks".

Parse BSDL is case-sensitive. The entity name in the input BSDL must match the block name in the Encounter Test model, and the port name in the BSDL must match the pin name in the model, unless you fold the BSDL entity name and the port names to upper or lower case by use of the `bsdlfold` keyword. The folding will occur before any model correlation checks are performed. The ability to fold the data is provided so that you can attempt to match BSDL data to the model data without having to rewrite the BSDL due to case-related problems. The `bsdlfold` keyword can be toggled (upper/lower) to attempt model matching. The following is an example:

If the input BSDL had the following statement:

```
port (clk :in bit; q :out bit_vector(1 to 8); d :in bit_vector(1 to 8);
      gnd, vcc :linkage bit; oc_neg :in bit; tdo :out bit; tms, tdi, tck :in bit);
```

and you specify `bsdlfold=upper`, then Encounter Test views the BSDL as:

```
port (CLK :in bit; Q :out bit_vector(1 to 8); D :in bit_vector(1 to 8);
      GND, VCC :linkage bit; OC_NEG :in bit; TDO :out bit; TMS, TDI, TCK :in bit);
```

Refer to "read_bsdl" in the *Encounter Test: Reference: Commands* for detail on these command line parameters.

Notes:

1. The port names in the port statement, the TAP attribute statements, the constant package statements, and the Boundary Register statements are all folded.
2. There is no provision for correlating mixed-case data; mixed-case data in the model must match that in the BSDL.
3. The input file is not changed; case-folding occurs only in the data stored by the application.

Refer to "Boundary Scan Attributes" in the *Encounter Test: Reference: Legacy Functions* for additional information.

- Package File - a package file is an ASCII file which describes boundary scan cell definitions. The package file is in the form of a standard VHDL package file definition.

  One package file must be provided for each package reference in the BSDL file "USE" statement. This normally includes an IEEE 1149.1 Standard package file (e.g. STD_1149_1_1994) and zero or more user defined package files (e.g. IBMDFT_1149_1_1998_V5).

  The Standard package files are shipped with Encounter Test (`$Install_Dir/defaults/bsdl`) while the user-defined package file(s) are provided by the technology provider or design team, in the case of custom boundary scan cell designs.

  For an example, refer to the standard package file *STD_1149_1_1994* shipped with Encounter Test in directory `$Install_dir/defaults/bsdl`.

- Encounter Test Model - optional

  You need an Encounter Test model only if you wish to perform BSDL model correlation checks. *Parse BSDL* checks the port and entity names and port directions in the BSDL against the corresponding data in the model.

Sample BSDL files and a sample package file have been shipped with Encounter Test. They reside in `$Install_Dir/defaults/bsdl` where `$Install_dir` is the execution environment set by invoking Encounter Test.

Writing BSDL

The `write_bsdl` tool generates boundary scan description language (BSDL) for an IEEE 1149.1 design. Its purpose is to relieve Encounter Test users from the time-consuming and error prone task of writing and maintaining BSDL for their 1149.1 designs. The completeness of the generated BSDL is bounded by user-provided IEEE 1149.1 design information, the level of compliance of the design, and `write_bsdl` tool limitations. Deficiencies in any of these areas may result in an incomplete file per the *IEEE Std 1149.1b-1994* (supplement to *IEEE Std 1149.1-1990* and *IEEE Std 1149.1a-1993)*.

Note: IBM ASICs customers should not need to edit their models; rather they should point to the appropriate ASICs tech libs.

`write_bsdl` uses two sources of information when producing a BSDL file:

1. The Encounter Test test mode and logic model provided when `write_bsdl` is invoked.
2. The optional BSDL file provided when `write_bsdl` is invoked.

`write_bsdl` uses the test mode and logic model as the primary source of BSDL information. Any information that can be derived from the test mode and logic model takes precedence over information that may exist in an input BSDL file. This typically includes entire statements. For example, `write_bsdl` completely ignores a `BOUNDARY_REGISTER` statement provided via an input BSDL file since it relies on the test mode and logic model to create a current view of the boundary register test logic implementation. Conversely, since `write_bsdl` cannot derive `RUNBIST_EXECUTION` information from the test mode and logic model, it will simply copy the `RUNBIST_EXECUTION` statement from its input BSDL file when it exists.

Note: Cadence RTL Compiler can create IEEE 1149.1 logic and using its `write_bsdl` command line, it can produce a BSDL file that can be imported into Encounter Test.

It is recommended that you run *1149.1 Boundary Scan Verification* or *Parse BSDL* to determine that your BSDL file is syntactically and semantically correct. This is true for any BSDL file, whether it has been created manually, generated by a test insertion tool, or generated by `write_bsdl`. Refer to "Verify 1149.1 Boundary" in the *Verification and Analysis Reference* and "Parsing BSDL" on page 304 for additional information.

To Write BSDL, refer to "write_bsdl" in the *Encounter Test: Reference: Commands*.

Restrictions

`write_bsdl` produces an incomplete BSDL description. It is also possible that the BSDL description produced by `write_bsdl` does not reflect a functional view of the design's primary input/output pins. Therefore, it is highly recommended that any BSDL file produced

by `write_bsdl` be reviewed by someone familiar with the function of the design to verify that the I/Os and their boundary register cells are described as they would operate in system mode. It is highly recommended that any BSDL file produced by `write_bsdl` be run through "Verify 1149.1 Boundary" so that BSDL deficiencies (e.g. incompleteness) can be identified.

Physical Pin Map Support

`write_bsdl` allows specification of up to four attribute name(s) for identifying up to four disjoint sets of physical pin information by using these options:

Include the `PIN_MAP_STRING` Statement by specifying the `write_bsdl` command option `physicalpinattr=`*`attr name`* `[,`*`attr name`*`]`

where *`attr name`* is a name of an attribute which appears in the design source for a component. There can be at most four attribute names specified. If `write_bsdl` is invoked with the option to include physical pin map string statements, it generates one `PIN_MAP_STRING` statement for each attribute name specified.

The following syntax and semantic checks are performed for each attribute name specified:

- Each name must start with an alpha type character (A-Z, a-z)
- Each name may contain numerics and underscores only
- Each name is folded to uppercase and checked for uniqueness.

The actual attribute searched is done in a case sensitive manner. However, all names that are written to the BSDL will be folded to upper case. The first attribute name passed in will be used in the GENERIC statement at the start of the BSDL file. As an example, if you specify `verify_11491_boundary physicalpinattr=PIN_grid,BallGrid`, the generated BSDL would contain:

```
generic(PHYSICAL_PIN_MAP : string="PIN_GRID");
attribute PIN_MAP of <entity> : entity is PHYSICAL_PIN_MAP;
constant  PIN_GRID : PIN_MAP_STRING := "<generated physical pin map string">;
constant  BALLGRID : PIN_MAP_STRING :="<generated physical pin map string">;
```

`write_bsdl` will only search the attributes associated with the hierModel pins of the top most cell in the Encounter Test model when attempting to find a `physicalpinattr`. Only those attributes which have a complete set of valid physical pin names will be written to the BSDL. In cases where only a partial set of physical pin names is found, or no physical pin names are found, the corresponding `PIN_MAP_STRING` statement will be written as a comment with missing names being replaced by 3 underscores. However, in either case, a request to generate a `PIN_MAP_STRING` would need to be specified via `physicalpinattr` for any `PIN_MAP_STRING` to be produced.

Sample EDIF:

```
(cell &TDAJTAG2 (celltype generic)
     (view nl (viewtype netlist)
        (interface
          (port SYS_ACLK (direction input)
            (property K_FLAG (string "-AC"))
            (property PAD (string "AA01"))
            (property APAD (string "AA01"))
            (property BPAD (string "AA01"))
            (property CPAD (string "AA01"))
          )
          (port SYS_BCLK (direction input)
            (property K_FLAG (string "-BC"))
            (property PAD (string "AA02"))
            (property APAD (string "AA02"))
            (property BPAD (string "AA02"))
            (property CPAD (string "AA02"))
          )
            (port SYSCLK (direction input)
            (property K_FLAG (string "-SC"))
            (property PAD (string "AA03"))
            (property APAD (string "AA03"))
            (property BPAD (string "AA03"))
            (property CPAD (string "AA03"))
          )
```

On the `verify_11491_boundary` command line, specifying *`physicalpinattr`*`=PAD,APAD,BPAD,CPAD` would produce four `PIN_MAP_STRING` statements in the generated BSDL. One such statement would appear as follows:

```
constant PAD: PIN_MAP_STRING:=
      "SYSCLK:AA03, "    &
      "SYS_ACLK:AA01, "    &
      "SYS_BCLK:AA02";
```

Prerequisite Tasks

Perform the following before running `write_bsdl`:

1. Import a gate-level model of the design. Refer to section Building Test Mode in *Encounter Test: Guide 1: Models* for additional information.
2. Using the Build Test Mode window, create a test mode for the design so that the SCAN TYPE and test function pins are known to Encounter Test. The SCAN TYPE in the Mode Definition File must always be 1149.1. Refer to "Logic Test Structure Verification (TSV)" in the *Verification and Analysis Reference* for information on the available scan types.

   `write_bsdl` also accepts user defined initialization sequences as input. Refer to "Mode Initialization Sequences (Advanced)" on page 66 and "Coding an Externally Specified Initialization Sequence" on page 68 for additional information.

   Optionally, a rudimentary BSDL input file may be specified to *Create a Test Mode*. The purpose of this is to identify the 1149.1 test function pins (TCK, TMS, TDI, TDO, TRST)

if not already correctly identified in the test mode. 1149.1 compliance enable patterns may also be specified by the BSDL, if the associated primary inputs have not already been assigned the TI test function pin attribute in the mode definition. If it is desired that 1149.1 test function pins and compliance enable pins be identified from BSDL rather than from the test mode, then an input BSDL file is specified to `write_bsdl` and the option to use a test function pin Assignment from BSDL is specified. In this manner a fast path for test function pin information is supported that doesn't require rerunning the Build Test Mode every time an adjustment to their specification is made.

Creating Initial BSDL File for Custom 1149.1 Design

Perform the following steps to create an initial BSDL file for a custom 1149.1 macro design:

1. Build a logic model. Refer to section Building Test Mode in *Encounter Test: Guide 1: Models* for more information.
2. Modify the assignfile to declare `TCK`, `TMS`, `TDI`, and `TDO`, as shown below:

   ```
   assign pin=mychip_tck test_function= -TCK;
   assign pin=mychip_tms test_function= TMS;
   assign pin=mychip_tdi test_funtion= TDI;
   assign pin=mychip_tdo test_function= TDO
   ```

   Refer to "ASSIGN" on page 210 and "Assignfile Examples" on page 32 for more information.
3. Run the `build_testmode` command as follows:

   ```
   Run build_testmode WORKDIR=<directory> TESTMODE=tap_custom MODEDEF=1149
   ASSIGNFILE=<assignfile with TAP test_functions declared>
   ```

   Refer to build_testmode in the *Encounter Test: Reference: Commands* for more information on `build_testmode` options.
4. Write an initial BSDL file using the following syntax:

   ```
   write_bsdl WORKDIR=<my work directory> TESTMODE=tap_custom bsdloutput=<my
   initial bsdl out file>
   ```

   Refer to "write_bsdl" in the *Encounter Test: Reference: Commands* for more information on the `write_bsdl` command.

The preceding procedure assumes that the custom 1149.1 macro design is fully compliant with the IEEE specification.

Most of the standard instructions and instruction register length will be automatically configured; however you must add any other user-defined instructions to the BSDL file. After modifying this initial BSDL file, use it to rebuild the 1149.1 test mode so that you can perform 1149.1 compliance checking and Boundary Scan Test Structure Verification.

Index

Numerics

1149.1
  test function pins 50
1149.1 Tap Controller State Diagram 78
1149.1, keyword of SCAN syntax 199
1149.6 boundary scan
  output boundary scan cell changes 159
  tap controller changes 155
  test receiver 152
  verifying 172

A

A_SHIFT_SYSTEM_CLOCK (AS) 35
AS (A_SHIFT_SYSTEM_CLOCK) 35
ASSIGN statement syntax
  BLOCK 221
  diagram and keywords 210
  DOMAIN 222
  PIN 222
  PIPELINE_CLOCK 222
  PIPELINE_DEPTH 221
  PPI 221
  TEST_FUNCTION 222

B

B_SHIFT_CLOCK (BC) 35
B_SHIFT_SYSTEM_CLOCK (BS) 35
BC (B_SHIFT_CLOCK) 35
BDY (BOUNDARY_DATA_PIN) 48
BI (BIDI_INHIBIT) 42
BIDI_INHIBIT (BI) 42
block level compression 141
  commands 144
Boundary Register
  implementation by Test Synthesis 149
boundary scan
  Boundary Register 149
  IEEE 1149.1 architecture 146
boundary scan controls, IEEE 1149.1 49, 51
boundary scan controls, RPCT 48
BOUNDARY_DATA_PIN (BDY) 48
BOUNDARY, keyword of SCAN syntax 201
BS (B_SHIFT_SYSTEM_CLOCK) 35
BSDL - physical pin map support 308
BSDL extension - identifying image unwired ports 303
BSDL file
bsdl package file 306
BSDL Parser 304
  prerequisite tasks 304

C

CELL_BOUNDARY 207
CHANNEL_INPUT (CHI) 33
CHANNEL_OUTPUT (CHO) 33
CI (CLOCK_ISOLATION) 38
circuit states
  description 295
  flush scan chain 298
  misr observe 298
  misr reset 299
  mode initialization 296
  nonscanflush 298
  prpg load 299
  test constraint 295
  test constraint and clocks off 296
  test inhibit 295
  tie only 295
clock gates 38
CLOCK_ISOLATION (CI) 38
CLOCKDOMAIN syntax 238
clocks, system and scan 34
COMET statement syntax
  comet_name 235
  diagram and keywords 233
comet_name, keyword of COMET syntax 235
comments - mode definition 195
comments - TDR syntax 259
compression, block level 141
CONTROL_PIN (CTL) 48
create vector correspondence 68
creating BSDL

concepts 307
input files 151
prerequisite tasks 309
restrictions 307
CTL (CONTROL_PIN) 48
custom scan protocol
1149.1 example 77
defining 71
load suffix 75
multiple scan sections 73
restrictions 83
scanentry 79
scanexit 80
scanlastbit 80
scanop 75
scanop structure, advanced 76
scanop structure, basic 75
scanprecond 79
scansectionexit 80
scansequence 79
customer service, contacting 15
CUTPOINTS statement syntax
diagram and keywords 236
CUTPOINTS, keyword of DELETE
syntax 236

D

DELAY_PROCESSING 281
DELAY_PROCESSING statement
syntax 282
DELETE statement syntax
CUTPOINTS 236
diagram and keywords 236
PPIS 236
DELETE, keyword of ON_BOARD_MISR
syntax 233
DELETE, keyword of
ON_BOARD_PRPG 232
DRIVER_RECEIVER, keyword of
TEST_TYPES syntax 207
DYNAMIC, keyword of FAULTS
syntax 208
DYNAMIC, keyword of TEST_TYPES
syntax 204

E

E_SHIFT_CLOCK (EC) 35
E_SHIFT_SYSTEM_CLOCK (ES) 36
EC (E_SHIFT_CLOCK) 35
ECID, keyword of TEST_TYPES
syntax 206
embedded devices, power-up controls 51
ES (E_SHIFT_SYSTEM_CLOCK) 36

F

FAST_FORWARD, keyword of ON_BOARD
PRPG syntax 232
FAULTS syntax
diagram and keywords 208
DYNAMIC 208
NONE 208
STATIC 208
filename, keyword of
SEQUENCE_DEFINITION
syntax 231
FINITE__STATE_MACHINE (FSM) 41
FIXED_VALUE_DEFAULT syntax
diagram and keywords 210
flush scan chain circuit state 298
FORCE_B_CLOCK, keyword of SCAN
syntax 199
FORCE_B_CLOCK 199
FSM (FINITE__STATE_MACHINE) 41

G

GO test function pin 47
GSD, keyword of SCAN syntax 199

H

help, accessing 15
hier_net_name, keyword of
ON_BOARD_MISR syntax 232
hier_net_name 232
hier_net_name, keyword of
ON_BOARD_PRPG 231
hier_net_name 231

I

I/O wrap, keyword of TEST_TYPES
syntax 204

IDDq PROPAGATE, keyword of TEST_TYPES syntax 207
IEEE 1149.1
BSDL Parser 304
image unwired ports, identifying 303
IMPLICIT CHOPPERS statement syntax diagram and keywords 241
IN, keyword of SCAN syntax 200
INITIAL_STATE (IS) 52
initialization sequence
automatically generated 67
initialization sequence in TBDpatt format testmode example 69
initialization sequence requirements 66
initialization sequences 66
automatically generated 67
coding externally specified 68
overview 67
requirements 66
Instruction Decode Logic
use of 149

L

latches, test function pins 41
LENGTH, keyword of SCAN syntax 199
LFSR (linear feedback shift register) 99
LH (LINEHOLD) 40
LINEHOLD (LH) 40
load suffix 75
LOGIC, keyword of TEST_TYPES syntax 204
LSSD, keyword of SCAN syntax 198

M

MACRO_MODE statement syntax diagram and keywords
MACRO, keyword of TEST_TYPES syntax 204
MARKOFF, keyword of COMETS syntax 235
MEASURE 267
MEASURE Statement Syntax 268
misr observe circuit state 298
misr reset circuit state 299
MISR_OBSERVE (MO) 44
MISR_READ (MRD) 44
MISR_RESET_ENABLE (MRE) 44
MO (MISR_OBSERVE) 44
mode definition file
statements 196
syntax 195
use of 195
mode definition statement syntax
ASSIGN 210
COMET 233
CORRELATE 239
CUTPOINTS 236
DELETE 236
DIAGNOSTIC_MODE 241
FAULTS 208
FIXED_VALUE_DEFAULT 210
IMPLICIT CHOPPERS 241
MACRO_MODE 239
ON_BOARD_MISR 232
ON_BOARD_PRPG 231
RAM_INITIALIZE_VALUE 239
SCAN 197
SEQUENCE_DEFINITION 230
SIGNATURE_OBSERVATION_MODE 237
statements 196
TEST_TYPES 202
TESTER_DESCRIPTION_RULE TYPE 198
mode initialization circuit state 296
model_attribute_name, keyword of TEST_FUNCTION_PIN_ATTRIBUTES
model_attribute_name 209
MRD (MISR_READ) 44
MRE (MISR_RESET_ENABLE) 44
multiple test modes 99

N

NET, keyword of ASSIGN syntax 220
NET 220
NIC NO_INTERCONNECT 48
1149.1 boundary scan controls 49, 51
NO_INTERCONNECT (NIC) 48
NONE, keyword of FAULTS syntax 208
NONE 208
NONE, keyword of TEST_TYPES syntax 204
NONE 204
nonscanflush circuit state 298

O

OI (OUTPUT_INHIBIT) 42
OLC (OPCG Load_Clock) pin 47
OLC_A pin 47
OLC_B test function pin 47
OLE (OPCG_Load_Enable) pin 47
OLI (OPCG_Load_Input) pin 47
ON_BOARD 201
ON_BOARD_MISR statement syntax
  diagram and syntax 232
  hier_net_name 232
ON_BOARD_PRPG statement syntax
  diagram and keywords
on-product misr test function pins 44
OPCBIST, keyword of TEST_TYPES
    syntax 206
OPMISRplus 117
OSC (OSCILLATOR) 37
OSCILLATOR (OSC) 37
OSCILLATOR_PULSES_PER_CYCLE 26
    5
OSCILLATOR_PULSES_PER_TESTER_C
    YCLE statement syntax 265
oscillators 37
oTI 37
OUT, keyword of SCAN syntax 201
OUTPUT_INHIBIT (OI) 42

P

P_SHIFT_CLOCK (PC) 35
P_SHIFT_SYSTEM_CLOCK (PS) 35
package file, bsdl 306
padding, x-mask 128
parse BSDL
  concepts 304
  input files 304
partition file 26
PC (P_SHIFT_CLOCK) 35
PGE (PRPG_LOAD_ENABLE) 43
physical pin map support in creating
    BSDL 308
PIN_TIMING statement syntax 270
PIN_TIMINGS 269
PIN, keyword of ASSIGN syntax 219
PLD (PRPG_LOAD) 43
port alias extension 301
PPIS, keyword of DELETE syntax 236
PR (PRPG_RESTORE) 43
PRIMITIVE_BOUNDARY, keyword of
    TEST_TYPES syntax 207
prpg load circuit state 299
PRPG_LOAD (PLD) 43
PRPG_LOAD_ENABLE (PGE) 43
PRPG_RESTORE (PR) 43
PRPG_SAVE (PV) 43
PS (P_SHIFT_SYSTEM_CLOCK) 35
PV (PRPG_SAVE) 43

R

RAM_INITIALIZE_VALUE statement syntax
  diagram and keywords
removes all committed test 34
REQUIRED_STATE (RS) 51

S

sample bsdl file 302
scan chain sequence order 33
scan clock sequence number 36
scan sequences, automatic
  description of
SCAN statement syntax
  1149.1 199
  BOUNDARY 201
  GSD 199
  IN 200
  LENGTH 199
  LSSD 198
  TYPE 198
SCAN_ENABLE (SE) 38
SCAN_GATE (SG) 38
SCAN_IN (SI) 32
SCAN_IN (SI) test function 32
SCAN_IN_GATE (SIG) 44
SCAN_OUT (SO) 32
SCAN_OUT_FILL (SOF) 45
SCAN_OUT_GATE (SOG) 45
SCAN_REQUIREMENTS 284
SCAN_REQUIREMENTS statement
    syntax 285
scanentry 79
scanexit 80
scan-in/scan-out circuit states 296
scanlastbit 80
scanop 70, 71

scanop structure, advanced 76
scanop structure, basic 75
scanprecond 79
scansectionexit 80
scansequence 79
sequence definition 66
SEQUENCE_DEFINITION statement syntax
  diagram and keywords 230
  filename 231
  hier_net_name 231
SG (SCAN_GATE) 38
SHIFT_REGISTER, keyword of TEST_TYPES syntax 208
SI (SCAN_IN) test function 32
SIG (SCAN_IN_GATE) 44
SIGNATURE_OBSERVATION_MODE statement syntax
  diagram and keywords
  testmodename 238
SIGNATURES, keyword of TEST_TYPES syntax 206
skewed load and unload 75
SOF (SCAN_OUT_FILL) 45
SOG (SCAN_OUT_GATE) 45
State Diagram for 1149.1 TAP Controller 78
STATIC, keyword of FAULTS syntax 208
STATIC, keyword of TEST_TYPES syntax 204
STATS_ONLY, keyword of COMETS syntax 235
syntax, tdr
  DELAY_PROCESSING 281
  MEASURE 267
  OSCILLATOR_PULSES_PER_TESTER_CYCLE 265
  PIN_TIMING 269
  PRPG_DEFINITION 269
  SCAN_REQUIREMENTS 284
  TDR_DEFINITION 260
  TERMINATION 266
  TEST_PINS 261
system and scan clocks
  clock functions 34
  scan clock sequence number 36
SYSTEM_CLOCK (SC) 34

T

TAP Controller
  inputs and output 149
TAP Controller State Diagram 78
TC (TEST_CONSTRAINT) 40
TCK (TEST_CLOCK) 49
TD_MIGRATION, keyword of TEST_TYPES syntax 204
TDI (TEST_DATA_INPUT) 49
TDO (TEST_DATA_OUTPUT) 49
TDR (tester description rule)
  introduction 259
  sample TDRs 286
  statement syntax 259
TDR statement syntax
  diagram and keywords
  tdrname 197
TDR_DEFINITION 260
TDR_DEFINITION statement syntax 261
tdrname, keyword of TDR syntax 197
TERMINATION 266
TERMINATION statement syntax 266
test constraint and clocks off circuit state 296
test constraint circuit state 295
test constraint sequencing 40
test function attributes
  definition
  scan inputs and outputs 32
  system and scan clocks 34
test function pins
  1149.1 50
  1149.1 boundary scan controls 49, 51
  attributes
  circuit states 295
  clock gates 38
  determining usage 52
  latches 41
  LBIST controls 42
  on-product misr 44
  OPCG controls 46
  oscillators 37
  RPCT boundary scan controls 48
  simultaneous output switching control 42
  test constraints 40
  three-state driver control 42
  valid combinations 54
test inhibit circuit state 295

test mode
custom scan protocol 70
multiple test modes 99
overview 17
test mode, removing
performing
test mode, resetting
performing 103
TEST_CLOCK (TCK) 49
TEST_CONSTRAINT (TC) 40
TEST_DATA_INPUT (TDI) 49
TEST_DATA_OUTPUT (TDO) 49
TEST_FUNCTION_PIN_ATTRIBUTES syntax
diagram and keywords 209
TEST_INHIBIT (TI) 39
TEST_MODE_SELECT (TMS) 49
TEST_PINS 262
TEST_PINS statement syntax 262
TEST_RESET (TRST) 49
TEST_TYPES syntax 202
CELL_BOUNDARY 207
diagram and keywords 202
DRIVER_RECEIVER 207
DYNAMIC 204, 208
ECID 206
IDDq PROPAGATE 207
LOGIC 204
NONE 204, 208
OPCBIST 206
SHIFT_REGISTER 208
SIGNATURES 206
STATIC 204, 208
TD_MIGRATION 204
TIMED_TEST 205
tester description rule (TDR)
introduction 259
sample 286
statement syntax 259
testmodename, keyword of SIGNATURE_OBSERVATION_MODE syntax 238
TI (TEST_INHIBIT) 39
tie only circuit state 295
TIMED_TEST, keyword of TEST_TYPES syntax 205
TMS (TEST_MODE_SELECT) 49
TRST (TEST_RESET) 49
TYPE, keyword of SCAN syntax 198

U

UNCORRELATE 240
using Encounter Test
online help 15

X

x-mask padding 128

Z

ZMRE test function 44